SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

AN EFFICIENT TEST FOR CIRCULAR-ARC GRAPHS*

ALAN TUCKER†

Abstract. An undirected graph G is called a circular-arc graph if there exists a family of arcs on a circle and a 1–1 correspondence between vertices and arcs such that two distinct vertices are adjacent if and only if the corresponding arcs overlap. Such a family is called a circular-arc model for G. In this paper we present an $O(n^3)$-step algorithm for testing whether an n-vertex graph is a circular-arc graph, and if it is, constructing a circular-arc model. Unfortunately the algorithm, its proof, and its efficient implementation are all quite involved.

Key words. circular-arc graph, efficient test, circular one's, consecutive one's

1. A graph $G=(\mathcal{V},\mathcal{A})$ consists of a finite set $\mathcal{V}$ of vertices and a symmetric, reflexive adjacency relation $\mathcal{A}$. The complement $\bar{G}=(\mathcal{V},\bar{\mathcal{A}})$ of G has for distinct vertices x and y, $x\bar{\mathcal{A}}y \Leftrightarrow$ not $x\mathcal{A}y$. A graph G is called an *intersection graph* for a family $\mathcal{F}$ of sets if there exists a 1–1 correspondence between the vertices of G and the sets of $\mathcal{F}$ such that two distinct vertices are adjacent if and only if the associated sets intersect. Conversely, such a family $\mathcal{F}$ is called an *intersection model* for G. If $\mathcal{F}$ is a family of intervals on a line, G is called an *interval graph*. If $\mathcal{F}$ is a family of arcs[1] on a circle, G is called a *circular-arc graph*. Interest in interval graphs dates from a paper by biologist S. Benzer [1] in which he showed that overlap data involving fragments of a certain viral gene could be modeled by a family of intervals—this finding confirmed the hypothesis that DNA has a linear structure within genes (and helped earn Benzer a Nobel prize). Interval graphs have since found application elsewhere in the biological and social sciences (see the survey on interval graph uses in Roberts [10, Chap. IV]). Several authors have characterized interval graphs (Lekkerkerker and Boland [8], Gilmore and Hoffman [6], and Fulkerson and Gross [4]). Fulkerson and Gross [4] gave an $O(n^4)$ algorithm for testing whether an n-vertex graph is an interval graph. Recently, Lueker and Booth [2], [3], [9] (independently) improved the Fulkerson–Gross test with special data structures and a fast clique enumeration algorithm to run in $O(n^2)$ time (linear in the number of edges—the best possible time). This test is based on a test for the conseuctive 1's property in a certain (0, 1)-matrix, specifically, the graph's vertex-clique incidence matrix. A (0, 1)-matrix M has the conseuctive 1's property if the rows of M can be permuted so that the 1's in each column occur in consecutive rows.

Circular-arc graphs also have a potential role in genetic research (Stahl [11]). Circular-arc graphs have recently been applied to problems in multidimensional scaling (Hubert [7]), traffic control (Stouffers [12]), computer compiler design (Tucker [16]), and the characterization of a certain class of lattices (Trotter and Moore [13]). This author [14] has given a matrix characterization of circular-arc graphs and an efficient algorithm for recognizing proper circular-arc graphs (graphs with a circular-arc model in which no arc properly contains another arc). Gavril [5] has given efficient algorithms for some other special classes of circular-arc graphs. This author [15] has obtained a Kuratowski-type (forbidden-subgraph) structure theorem for proper circular-arc graphs and unit-length circular-arc graphs. Trotter and Moore [13] have a structure

* Received by the editors July 8, 1977, and in final revised form December 22, 1978.

† Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, New York 11784. This research was partially supported by the National Science Foundation under research Grant MPS-7505268.

[1] Without loss of generality, we can assume that all intervals and arcs are closed, i.e., that they contain in their endpoints, and that the $2n$ endpoints of the n intervals or arcs in $\mathcal{F}$ are distinct.

theorem for circular-arc graphs which partition into two cliques (i.e., those whose complement is bipartite). The matrix characterization of circular-arc graphs is based on a variant of the circular 1's property (similar to the consecutive 1's property except now the 1's in each column must occur in cyclicly consecutive rows). In our circular-arc graph algorithm we will require testing certain (0, 1)-matrices for the circular 1's property. Such a test can be converted to the problem of testing a related matrix for the consecutive 1's property (Booth and Lueker [3] can do a consecutive 1's test in linear time with respect to the number of 1's in the matrix). The conversion is based on the following argument. If the rows of a (0, 1)-matrix M can be arranged so that the 1's in each column occur in a circular (cyclicly consecutive) set of rows, then the 0's in each column also will occur in a circular set of rows. It follows that if M' is obtained from M by complementing a subset of columns of M, i.e., interchanging 1's and 0's in these columns, then M has the circular 1's property if and only if M' does. Now suppose that the subset of columns of M complemented are the columns with a 1 in the last row (so M' has all 0's in its last row). Then an arrangement of M' with circular 1's and (by cyclicly permuting the rows, if necessary) still all 0's in the last row actually has consecutive 1's. *Thus M' has the consecutive* 1'*s property if and only if M has the circular* 1'*s property.*

In the next section we describe an algorithm for constructing, if possible, a circular-arc model for a given n-vertex graph G. In that presentation, we simply state four major propositions needed in the algorithm's construction. The proofs of these propositions are given in the last section. Section 3 discusses efficient $O(n^3)$ implementation of the algorithm ($O(n^2)$ storage locations are required). Section 4 consists of a 14-vertex example of the algorithm.

It is important to note that the cyclic order of arc endpoints in a circular-arc model for a circular-arc graph is, in general, far from unique. For example, the complete n-partite graph $H_{2,n}$ with two vertices in each part (the two vertices in each part are not mutually adjacent themselves but are adjacent to all other vertices) is a $2n$-vertex circular-arc graph with $(n-1)!2^n$ different possible cyclic orders of arcs in a circular-arc model. Interval graphs can also have exponential numbers of different models but the following two facts greatly simplify interval modeling: (i) maximal cliques in interval graphs correspond to points of maximal overlap on the line; and (ii) an n-vertex interval graph has at most $n-1$ maximal cliques. Thus it suffices to find an ordering of maximal cliques such that each vertex (interval) is in a consecutive set of maximal cliques (this is where the consecutive 1's property is involved). In circular-arc graphs, a clique can be modeled by a set of arcs that, by reaching around the circle, have no common point of overlap; $H_{2,n}$ has 2^n maximal cliques. These observations about the complexity of arc models and cliques hopefully convey a sense of the rich combinatorial structure possible in circular-arc graphs and of why complicated ad hoc methods, instead of general principles, are the basis of almost all results in the theory of circular-arc graphs. It had even been conjectured (page 3 of Booth [2]) that testing for circular-arc graphs might be an NP-complete problem.

2. The circular-arc algorithm.

In this section we present an algorithm for constructing a circular-arc model (one of possibly many models) for a given n-vertex graph G, assuming such a model exists. If G is not a circular-arc graph, either the construction may be forced to terminate or the family of circular arcs produced by the algorithm will be found not to be a model for G. As in this author's characterization and recognition test of proper circular-arc graphs, the circular-arc test splits into two cases, depending on whether $\bar{G}$, the complement of G, is bipartite; if not bipartite, $\bar{G}$ must contain a primitive (chordless) odd-length circuit. In addition, the nonbipartite case has two major subcases (the two subcases are not exclusive):

Case I. $\bar{G}$ bipartite; equivalently, G partitions into two cliques;

Case II. $\bar{G}$ has an odd-length chordless primitive circuit.

Subcase IIa. $\bar{G}$ has a triangle; equivalently, G has 3 (or more) mutually nonadjacent vertices.

Subcase IIb. $\bar{G}$ has an odd hole (an odd-length primitive circuit of length $\geqq 5$).

The algorithm in each case consists of three major stages. In Stage One, a special subset of m vertices is found whose associated arcs can be physically determined (i.e., positioned on a circle). The endpoints of these m arcs divide the circle up into $2m$ sections such that no remaining arc could possibly have both its endpoints in the same section. In Stage Two, the endpoints of the remaining arcs are located in appropriate sections. In Stage Three, the set of arc endpoints in each section is arranged appropriately. Instead of referring to vertices and their associated arcs, we shall henceforth consider G to consist of n arcs which overlap prescribed other arcs, and eliminate any reference to vertices. Initially, before the circular-arc model is built, the overlap of arcs is just an abstract relation. We denote the overlap of arcs A_1, A_2 by $A_1 \cap A_2$ (or nonoverlap, $A_1 \not\cap A_2$) and the set of arcs which A overlaps by $\mathcal{N}(A)$ (note $A \in \mathcal{N}(A)$). In a circular-arc model, let $\mathrm{Cc}(A)$ and $\mathrm{Cl}(A)$ denote the counterclockwise and clockwise endpoints, respectively, of arc A. Finally, we assume that the graph is preprocessed to eliminate equivalent arcs (arcs with the same $\mathcal{N}(A)$'s) and arcs overlapping all other arcs. These equivalent and all-overlapping arcs are trivially incorporated into a model if G is a circular-arc graph.

Stage One. Find a special subset of arcs to be positioned on the circle so as to divide the circle into sections such that no remaining arc has its two endpoints in the same section.

Case I. *G partitions into two cliques.* First we note that if G is a circular-arc graph which partitions into two cliques, then in *any* circular-arc model for G, there are two points such that every arc contains (at least) one of the two points. Let G' be a minimal graph with a circular-arc model for which this assertion is false. By the minimality of G', if any arc A_1 in a model for G' is excluded (i.e., the associated vertex in G' is deleted), the remaining arcs all contain one of some two points. But if $A_2 \subset A_1$, then A_1 contains the point in A_2. Thus G' must be a proper circular-arc graph. However in [15], this author showed that this assertion is true for any proper circular-arc graph. So we can assume that our model has a common point p_1 at the top of the circle contained in all arcs in a clique $\mathcal{B}$ and a common point p_2 at the bottom of the circle contained in all arcs in the other clique $\mathcal{D}$ (note that the choice of $\mathcal{B}$ and $\mathcal{D}$ is not always unique, but that does not matter). We shall use B to denote a typical arc in $\mathcal{B}$ and D a typical arc in $\mathcal{D}$.

Pick any (set-theoretically) maximal arc B in $\mathcal{B}$ (i.e., $\mathcal{N}(B)$ maximal) and call it B_0^*. We inductively define A_i^* (a $\mathcal{B}$ or $\mathcal{D}$ arc) as follows. Suppose $A_{i-1}^* = B_{i-1}^*$ (A_{i-1}^* is a B-arc, initially use B_0^*) and let B_i' be a maximal arc B such that B_{i-1}^* contains B, i.e., $\mathcal{N}(B) \subset \mathcal{N}(B_{i-1}^*)$. If there exists an arc D with $D \cap B_{i-1}^*$, $D \not\cap B_i'$, and $D \cap B$ for all B such that $\mathcal{N}(B) \not\subset \mathcal{N}(B_{i-1}^*)$, then let $A_i^* = D_i^*$ be a minimal such D. Further, $A_{i+1}^* = D_{i+1}^*$ is a minimal such D which additionally contains D_i^*, and so on with $A_{i+2}^* = D_{i+2}^*$ a minimal such D containing D_{i+1}^*, etc. until there is no such D containing D_{i+k-1}^*, and then $A_{i+k}^* = B_{i+k}^*$ is the arc B_i' (if no such D's exist at all, $A_i^* = B_i^* = B_i'$).

Next let D_{-1}^* be the maximal D with $D \not\cap B_0^*$, and recursively define D_{-i-1}^* as the maximal D contained in D_{-i}^*. The A^* arcs range from A_{-q}^* to A_r^*. (Note: A_{-q}^* must be a $\mathcal{D}$ arc and $-q \leqq -1$ since B_0^* does not overlap all arcs; similarly A_r^* must be a $\mathcal{B}$ arc.) Since $\mathcal{N}(B_j^*) \subset \mathcal{N}(B_i^*)$ for $i < j$, we may assume in our model that B_i^* physically contains

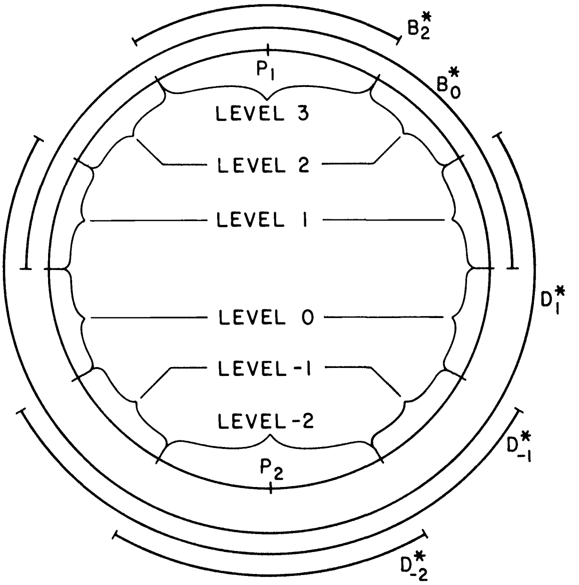


FIG. 1. *Positions of B^* and D^* arcs in Stage One, Case* I *of algorithm.*

B_j^* and similarly for D_i^* and D_j^*. Further for $i<j$, B_i^* and D_j^* each contain all arcs not properly contained in the other and so we can also assume that B_i^* and D_j^* physically overlap at both ends. Thus the A_i^*'s can be positioned as shown in the example in Fig. 1. We call the sections on either side of the circle contained between the endpoints of A_{i-1}^* and A_i^* *level i* (see Fig. 1), the section within B_r^* level $r+1$, and the section within D_{-q}^* level $-q$. In Stage Two, we shall divide each level into its left and right sections. It is straightforward to check that any B or D with both endpoints at the same level would have been incorporated in the set of A^*'s.

Subcase IIa. *G contains at least* 3 *mutually nonoverlapping arcs.* Find a (set-theoretically) maximal set $\mathscr{I}$ of independent (i.e., mutually nonoverlapping) arcs such that no arc in $\mathscr{I}$ properly contains any other arc of G and such that for no arc $A_i^* \in \mathscr{I}$, do there exist two independent arcs A_1, A_2 which overlap only A_i^* in $\mathscr{I}$(in the latter case, replace A_i^* by A_1 and A_2 in $\mathscr{I}$). Let $A_1^*, \cdots, A_m^*$ be such a set $\mathscr{I}$. The key step in Subcase IIa of the algorithm is finding an appropriate cyclic order for $\mathscr{I}$ consistent with its arcs' overlap relation with the other arcs (by a cyclic order, we mean a linear order or any cyclic permutation and/or inversion of that order). We form an $m \times 2(n-m)(0, 1)$ incidence matrix M_0 with a row for each A_i^* and two columns, $2j-1$ and $2j$, for each remaining arc A_j. Let $\mathcal{N}^*(A_j)$ denote the set of A_i^*'s that A_j overlaps. For each A_j there are either zero, one, or two A_i^*'s such that $A_i^* \in \mathcal{N}^*(A_j)$ and $\mathcal{N}(A_i^*) \not\subset \mathcal{N}(A_j)$; these are A_i^*'s within which A_j has an endpoint and we call them the *ends of* A_j. (If more than two such A_i*'s exist, G clearly cannot be a circular-arc graph and the algorithm terminates.) If there is zero or one such A_i^*, let column $2j$ of M_0 have a 1 in row i if $A_i^* \in \mathcal{N}^*(A_j)$. If $\mathcal{N}^*(A_j)$ has no ends, then column $2j-1$ is all 0's. If A_j has one end, call it $A_{i_j}^*$, then we obtain column $2j-1$ from column $2j$ by converting the 1 in row i_j to a 0. If A_j has two ends, call them $A_{i_j}^*$ and $A_{i_{j'}}^*$, then let columns $2j-1$ and $2j$ have a 1 in row i if $A_i^* \in \mathcal{N}^*(A_j)$ except that entries $(i_j, 2j-1)$ and $(i_{j'}, 2j)$ are 0; however if $\mathcal{N}^*(A_j)=2$, then treat A_j as if it had no ends. See Fig. 2. (For ease in visualizing this example, the arcs are pictured in a valid circular-arc model; columns of all 0's are omitted in Fig. 2b.) A valid cyclic order for $\mathscr{I}$ (compatible with a circular-arc model for G) will induce

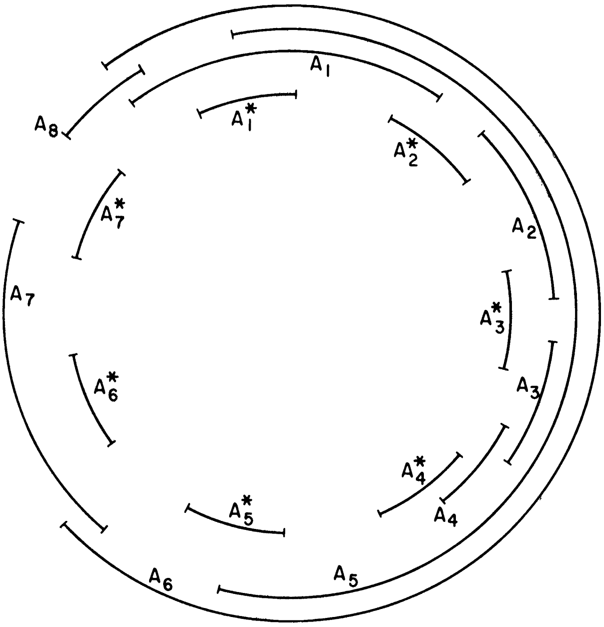


FIG. 2a. *Family of arcs $G = G_0$.*

	2	4	6	8	10	12	14	16
1*	1	0	0	0	1	1	0	0
2*	1	1	0	0	1	1	0	0
3*	0	1	1	0	1	1	0	0
4*	0	0	0	1	1	1	0	0
5*	0	0	0	0	1	1	0	0
6*	0	0	0	0	0	0	1	0
7*	0	0	0	0	0	0	1	1

FIG. 2b. *M_0 matrix for family of arcs in Fig. 2a. Odd numbered columns are all 0's and are omitted.*

	4	8	9	10	12	14	16
1′*	1	0	0	1	1	0	1
2′*	1	1	1	1	1	0	0
4*	0	1	1	1	1	0	0
5*	0	0	1	1	1	0	0
6*	0	0	0	0	0	1	0
7*	0	0	0	0	0	1	1

FIG. 2c. *M_2 matrix with unique cyclic order.*

circular 1's in M_0. Observe that by deleting the 1's in entries $(i_j, 2j-1)$ and $(i_{j'}, 2j)$, we force $A^*_{i_j}$ and $A^*_{i_{j'}}$ to be at the ends of the cyclicly consecutive set $\mathcal{N}^*(A_j)$ in any cyclic order of $\mathscr{I}$ which induces circular 1's in M_0.

Unfortunately, there will in general be several cyclic orderings of $\mathscr{I}$ which induce circular 1's in M_0. It may be possible to invert or arbitrarily permute the order of a consecutive set of A^*_i's or of a consecutive set of subsets of (ordered) A^*_i's (these are the only possibilities, see Fulkerson and Gross [4] or Booth and Leuker [3]). For example, in Fig. 2 the order of A^*_1, A^*_2, A^*_3 could be inverted or the subsets $\{A^*_1, A^*_2, A^*_3\}$, $\{A^*_4\}$, $\{A^*_5\}$ could be permuted without destroying circular 1's in the arrangement of M_0 shown in Fig. 2b (although these new orders would no longer be induced by valid circular-arc models for this graph). If there is nonuniqueness, we must worry about the overlaps which occur in between consecutive A^*_i's such as the overlap of A_1 and A_8 in Fig. 2a. (This overlap prohibits inverting the order of A^*_1, A^*_2, A^*_3.) To handle this problem, we recursively define a sequence of graphs, related matrices, and cyclicly ordered independent sets, $G'_i, M_i, \mathscr{I}_i$ where $G'_0 = G$, $\mathscr{I}_0 = \mathscr{I}$, and M_0 is as just defined. Given $G'_{i-1}, M_{i-1}, \mathscr{I}_{i-1}$, pick an arc A'_i in G'_{i-1} such that $|\mathcal{N}^*(A_i)|$ is minimum among all arcs (not in $\mathscr{I}_{i-1}$) which overlap an $A^* \in \mathscr{I}_{i-1}$ which is part of a set, or a subset in a set of subsets, that may be inverted or cyclicly permuted without destroying the circular 1's in M_{i-1} (induced by the cyclic order of $\mathscr{I}_{i-1}$). Now delete A'_i and the A^*'s in $\mathscr{I}_{i-1}$ which A'_i overlaps, and replace these arcs with a new arc A'^*_i which overlaps any arc overlapped by one or more of the deleted arcs. G'_i is this new graph; M_i is defined for G'_i just as M_0 was defined for G; and $\mathscr{I}_i$ is A'^*_i plus the remaining arcs in $\mathscr{I}_{i-1}$. Now find a cyclic order for $\mathscr{I}_i$ which induces circular 1's in M_i. We continue this sequence $\{G'_i, M_i, \mathscr{I}_i\}$ until for $i = s$, M_s has a unique cyclic order. In Fig. 2a, the first arc chosen A'_1 would be either A_3 or A_4, say it is A_3; the next arc A'_2 would be A_1 or A_2 or A_4 (note that A_4 now overlaps A'^*_1 and A^*_4 in $\mathscr{I}_1$). If A_1 were chosen next, M_2 would be as shown in Fig. 2c ($M_2 = M_s$ in this case).

We call a cyclic order of $\mathscr{I}_i$ which induces circular 1's in M_i an *extension* of a given cyclic order of $\mathscr{I}_{i+1}$ (which also induces circular 1's in M_{i+1}) if: (a) after the reduction process from G'_i to G'_{i+1}, $\mathscr{I}_{i+1}$ has the given cyclic order when A'^*_{i+1} simply replaces $\mathcal{N}^*(A_{i+1})$ in the cyclic order of $\mathscr{I}_i$; and (b) the order of $\mathcal{N}^*(A'_{i+1})$ in $\mathscr{I}_i$ satisfies the following condition, called the A'_{i+1}-extension condition: if $|\mathcal{N}^*(A'_{i+1})| \geqq 2$, if A'_{i+1} has exactly one end in $\mathscr{I}_i$ (see the definition above of an end), and if there exists $A^0 \in \mathcal{N}(A'_{i+1})$ with $A^0 \not\cap \mathcal{N}^*(A'_{i+1})$, then the end of A'_{i+1} is not beside the arcs of $\mathcal{N}^*(A^0)$ in the cyclic order of $\mathscr{I}_i$.

PROPOSITION 1. *Let G be a circular-arc graph with $\{G'_i, M_i, \mathscr{I}_i\}_{i=0}^{=s}$ as described above. If, starting with a cyclic order θ_s of $\mathscr{I}_s$ which induces circular* 1'*s in M_s, one recursively picks θ_i an extension of θ_{i+1}, then the final extension θ_0 is a valid cyclic order for $\mathscr{I}$ in some circular-arc model of G.*

From Proposition 1 we get a valid cyclic order for $\mathscr{I}$. Assume $A^*_1, A^*_2, \cdots, A^*_m$ are indexed in such an order. Position these arcs in this order clockwise around the circle. We define sections of the circle in this nascent model for G as follows. Section S_{2i} is the segment of the circle between $\mathrm{Cc}(A^*_i)$ and $\mathrm{Cl}(A^*_i)$ and S_{2i-1} is the segment between $\mathrm{Cl}(A^*_{i-1})$ and $\mathrm{Cc}(A^*_i)$. By the choice of $\mathscr{I}$, no other arc is contained in a single section. Since every arc does not overlap at least one other arc, no arc can reach all the way around the circle to have both endpoints in the same section. Thus no arc's two endpoints are in the same section.

Subcase IIb. *$\bar{G}$ has an odd hole.* Let $A^*_1, \cdots, A^*_{2r+1}$, $r \geqq 2$, be a set of arcs corresponding to an odd hole in $\bar{G}$ such that no other arc A is contained in any of the A^*_i's, i.e., $\mathcal{N}(A) \not\subset \mathcal{N}(A^*_i)$. Any odd hole can be reduced either to such a proper odd hole

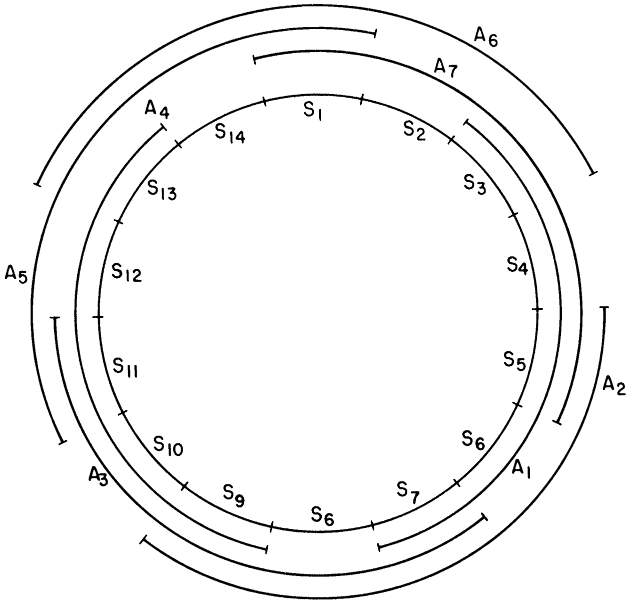


FIG. 3. *Positions of A^*'s when $2r+1=7$ in Stage One, Subcase* IIb *of algorithm.*

or to a triangle (Subcase IIa). Let the A^*'s be indexed so that A_i^* overlaps all other A^*'s except A_{i+r}^* and A_{i+r+1}^* (subscript addition is mod $2r+1$; note that $i+r+1=i-r$). It is a straightforward matter to check that the $2(2r+1)$ endpoints of these A^*'s have a unique cyclic order of the form $\cdots$ Cl(A_{i-r}^*), Cc(A_i^*), Cl(A_{i-r+1}^*), Cc(A_{i+1}^*), $\cdots$. See the example in Fig. 3. Let section S_{2i} be the segment of the circle between Cl(A_{i-r}^*) and Cc(A_i^*) and S_{2i+1} the segment between Cc(A_i^*) and Cl(A_{i-r+1}^*). Since no other arc is contained inside one of the A^*'s then as in Subcase IIa, we have that no arc's two endpoints are in the same section.

Stage Two. Determine the two (distinct) sections to which each arc's endpoints belong.

Case I. *G partitions into the cliques $\mathcal{B}$ and $\mathcal{D}$.* From Stage One of Case I we have the set $A_{-q}^*, \cdots, A_0^*, \cdots, A_r^*$ and levels $-q$ through $r+1$ located on the circle as typified by the example in Fig. 1 and such that the two endpoints of any other arc must be in different levels. Our first task is to determine the levels of the two endpoints of the other arcs. By the *higher end* and *lower end* of an arc in $\mathcal{B}$ (or in $\mathcal{D}$) we mean the endpoint at the higher (greater) and lower level, respectively. Let $h(A)$ and $l(A)$ denote the levels of the higher and lower ends of A, respectively. For $B\in\mathcal{B}$, $l(B)$ must equal to the minimum i such that $\mathcal{N}(B)\not\subset\mathcal{N}(B_i^*)$ or $B\cap D_i^*$, and $h(B)$ can be set equal to the minimum i such that $\mathcal{N}(B_i^*)\subset\mathcal{N}(B)$ or such that B overlaps D_i^* and all D with $\mathcal{N}(D)\not\subset\mathcal{N}(D_i^*)$ (as argued in Stage One, in these situations we can assume B contains B_i^* or overlaps both ends of D_i^*). Similarly for $D\in\mathcal{D}$, $h(D)$ must be equal to the maximum i such that $D\cap B_{i-1}^*$ or $\mathcal{N}(D)\not\subset\mathcal{N}(D_{i-1}^*)$ and $l(D)$ can be set equal to the maximum i such that D overlaps B_{i-1}^* and all B with $\mathcal{N}(B)\not\subset\mathcal{N}(B_{i-1}^*)$ or such that $\mathcal{N}(D_{i-1}^*)\subset\mathcal{N}(D)$.

Next we must determine whether an arc has its higher end on the left or right side of the circle or, more critically, on the same or opposite side as the higher ends of other arcs. To this end, we define a pair of relations S (higher ends on same side) and O (higher ends on opposite sides) on the other arcs (i.e., excluding the A^*'s) as follows: (By the converse of a condition, we mean the condition obtained by interchanging the roles of $\mathscr{B}$ arcs and $\mathscr{D}$ arcs, of higher and lower ends ($h(\cdot)$ and $l(\cdot)$) and of $O(B_1, B_2)$ and $O(D_1, D_2)$.)

(i) $O(B, D)$ is $B \cap D$, $l(B) > l(D)$ and $h(B) > h(D)$.

(ii) $S(B, D)$ if $B \not\cap D$, $l(B) < h(D)$.

(iii) $O(B_1, B_2)$ if there exist $B_1, B_2, B_3, D_1, D_2, D_3$ with all higher ends in level k_2 and all lower ends in level k_1, except $k_1 < h(D_3) \leqq l(B_3) < k_2$, $B_i \not\cap D_i$, $i = 1, 2, 3$, $D_1 \not\cap B_3$, $B_2 \not\cap D_3$, all other pairs overlap.

(iv) $O(B_1, B_2)$ if there exist $B_1, B_2, D_1, D_2, B_i \cap D_i, B_i \not\cap D_j$, $i = 1, 2$, $j \neq i$, all arcs with same upper end level, $l(D_i) < l(B_i)$, $i = 1, 2$; or the converse of this condition.

(v) $O(B_1, B_2)$ if there exist B_1, B_2, D_1, D_2, $l(B_i) = h(D_i)$, $B_i \cap D_i$, $i = 1$, 2, $l(B_i) = l(B_2)$, $B_i \not\cap D_j$, $i \neq j$, $i = 1, 2$.

(vi) $O(B_1, B_2)$ if there exist B_1, B_2, D_1, D_2, $B_i \cap D_i$, $B_i \not\cap D_j$, $i = 1, 2, j \neq i$, $l(B_1) = h(D_1) = l(B_2) = l(D_2) = k$ and either $h(D_2) < l(B_2)$ or else $h(D_2) = l(B_2)$ and there exist D_3 with $l(D_3) < k$, $D_3 \cap B_1$, $D_3 \not\cap B_2$; or the converse of this condition.

(vii) $O(B_1, B_2)$ if there exist B_1, D_1, A_i, $i = 1, 2, \cdots, m$ $(m \leqq 3)$, $A_m = B_2$, all higher ends in level k_2 and all lower ends in level k$_1$, except one end of A_1 is between k_1 and k_2 and one end of A_m is above k_2, $B_1 \not\cap D_1$, B_1, D_1 overlap all A_i's, A_i overlaps all other A_j's except A_{i-1} and A_{i+1}; if $l(A_m) = k_2$, A_m overlaps only A_{m-1} among the D arcs. Or the converse of this condition.

It is not difficult to check that these same and opposite side relations must hold for arcs in a circular-arc model. These relations partition the remaining arcs of G into components $\mathscr{C}_1, \mathscr{C}_2, \cdots, \mathscr{C}_h$ such that the relative sides of the higher end of arcs within each $\mathscr{C}_i$ are fixed with respect to each other, while arcs in different components are unrelated.

PROPOSITION 2. *If G is a circular-arc graph, there are circular-arc models for G with each of the 2^h possible left side-right side orientations of the h different components defined by relations S and O.*

By Proposition 2, we can arbitrarily pick one of the two sides in each component to be the arcs with higher ends on the left side. (We note that Proposition 2 is the most crucial step of the algorithm and its proof is necessarily quite complex.) Now we have a side and level of the higher endpoint and the lower endpoint of each remaining arc. We convert these positions into section numbers with the following scheme of section numbering based on the positions of the $2(q+r+1)$ endpoints of the A^*'s discussed in Stage One (and typified by the example in Fig. 1). Section S_1 is between $\mathrm{Cc}(A^*_{-q})$ and $\mathrm{Cl}(A^*_{-q})$ and section S_{q+r+2} is between $\mathrm{Cc}(A^*_r)$ and $\mathrm{Cl}(A^*_r)$. Let $p_i = \mathrm{Cc}(B^*_i)$ or $\mathrm{Cl}(D^*_i)$ depending on whether A^*_i is an $\mathscr{B}$ or $\mathscr{D}$ arc, and similarly let $t_i = \mathrm{Cl}(B^*_i)$ or $\mathrm{Cc}(D^*_i)$. Then for $2 \leqq i \leqq q+r+1$, S_i is between p_{i-q-2} and p_{i-q-1} and for $q+r+3 \leqq i \leqq 2(q+r+1)$, S_i is between $t_{q+2r+3-i}$ and $t_{q+2r+2-i}$.

Subcase IIa. *G contains at least* 3 *independent arcs.* From Stage One, we have a set $\mathscr{I}$ of independent arcs $A^*_1, A^*_2, \cdots, A^*_m$ indexed in an order compatible with a circular-arc model for G (assuming one exists) with S_{2i} contained in A^*_i and S_{2i-1} the segment between A^*_{i-1} and A^*_i. Moreover each remaining arc must have its endpoints in different sections. Let $A^*_{i_1}$ thru $A^*_{i_2}$ be the cyclicly consecutive set of A^*'s in $\mathscr{N}^*(A_i)$ (the A^*'s that overlap A_i). If $i_1 \neq i_2$ and A_i does not overlap all A^*'s, then $\mathrm{Cc}(A_i)$ can be assumed to be in section S_{2i_1-1} if $\mathscr{N}(A^*_{i_1}) \subset \mathscr{N}(A_i)$ and must be in S_{2i_1} otherwise, and $\mathrm{Cl}(A_i)$ can be assumed to be in S_{2i_2+1} if $\mathscr{N}(A^*_{i_2}) \subset \mathscr{N}(A_i)$ and must be in S_{2i_2} otherwise. If $\mathrm{Cc}(A_i)$ is in S_{2j+1} and $\mathrm{Cl}(A_i)$ is in S_{2j-1}, we call A_i an A^*_j-complementary arc. If $i_1 = i_2 = j$ and $\mathscr{N}(A^*_j) \subset \mathscr{N}(A_i)$, then we can assume $\mathrm{Cc}(A_i)$ is in S_{2j-1} and $\mathrm{Cl}(A_i)$ is in

S_{2j+1}—we call such A_i an A_j^*-equivalent arc. If A_i overlaps all A^*'s and there are two cyclicly consecutive A^*'s, call them $A_{i_1}^*$, $A_{i_1+1}^*$, such that $\mathcal{N}(A_j^*) \not\subset \mathcal{N}(A_i)$, $j = i_1, i_1+1$, then $\mathrm{Cc}(A_i)$ must be in S_{2i_1+2} and $\mathrm{Cl}(A_i)$ must be in S_{2i_1}. We define the set $\mathcal{P}_j$ to consist of those arcs A_i such that either: (a) $\mathcal{N}^*(A_i) = \mathcal{I}$ and only $\mathcal{N}(A_j^*)$ of all the A^*'s is not contained in $\mathcal{N}(A_i)$ or else (b) $\mathcal{N}^*(A_i) = A_j^*$ and $\mathcal{N}(A_j^*) \not\subset \mathcal{N}(A_i)$. The subset of case (b) arcs is called $\mathcal{P}_j^0$. Note that by choice of $\mathcal{I}$ all $\mathcal{N}(A^*)$ cannot be contained in $\mathcal{N}(A_i)$, and if G is a circular-arc graph and $\mathcal{N}^*(A_i) = \mathcal{I}$, there cannot be two nonconsecutive A^*'s with $\mathcal{N}(A^*) \not\subset \mathcal{N}(A_i)$. Each arc in $\mathcal{P}_j$ has one endpoint in S_{2j} and the other in S_{2j-1} or S_{2j+1}.

We convert the problem of determining section ends of arcs in $\mathcal{P}_j$ to a circular-arc modeling problem for the following family (graph) G_j of arcs. We let A_0^* represent the combination of all A^*'s except A_j^* into one big arc (containing all sections except S_{2j-1}, S_{2j}, S_{2j+1} in the model of G we were building) which overlaps all arcs overlapped by the A^*'s it replaces. Besides A_0^*, A_j^* and $\mathcal{P}_j$, G_j also contains all A_j^*-complementary and A_j^*-equivalent arcs and the set $\mathcal{Q}_j$ of all other arcs A overlapping some but not all arcs in $\mathcal{P}'_j = \mathcal{P}_j^0 \cup A_j^* \cup \{A_j^*\text{-equivalent arcs}\}$. $\mathcal{P}'_j$ is the set of arcs contained in S_{2j-1}, S_{2j}, S_{2j+1}. Thus if $A \in \mathcal{Q}_j$, either $\mathrm{Cl}(A)$ is in S_{2j-1} or S_{2j} or else $\mathrm{Cc}(A)$ is in S_{2j} or S_{2j+1}. We eliminate all remaining arcs and also only let a pair of arcs A, A' in $\mathcal{Q}_j$ overlap if $\mathrm{Cl}(A)$ and $\mathrm{Cl}(A')$ are both in $\{S_{2j-1}, S_{2j}\}$ or if $\mathrm{Cc}(A)$ and $\mathrm{Cc}(A')$ are both in $\{S_{2j}, S_{2j+1}\}$. Note that no two arcs $A_1, A_2 \in \mathcal{Q}_j$ whose overlap has been destroyed could overlap inside A_j^* (in S_{2j}); for if so, by the definition of $\mathcal{Q}_j$ there must exist A'_1, A'_2 in $\mathcal{P}_j^0$ such

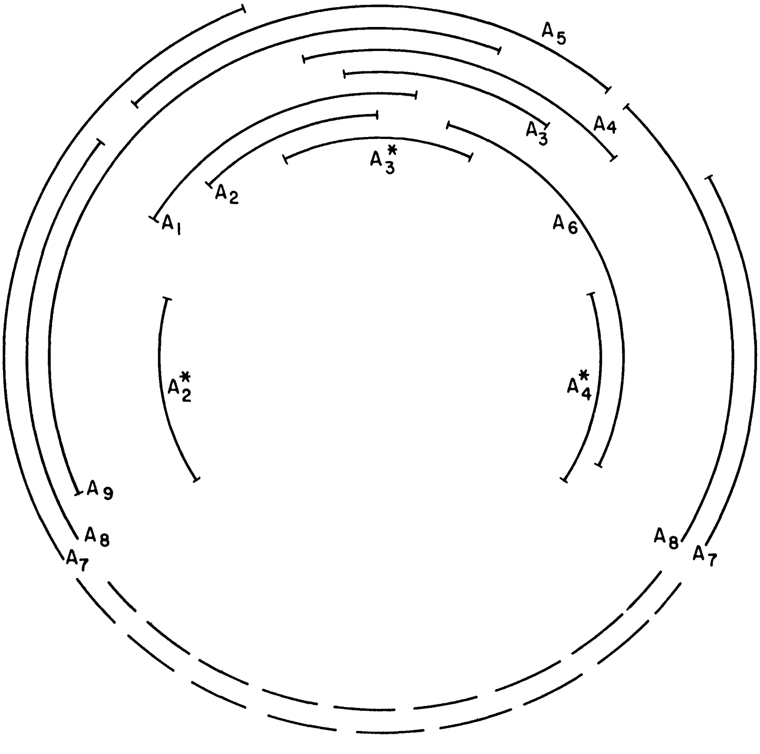


FIG. 4. *Arcs involved in a G_3 construction. All A^*'s except A_3^* are fused together to form A_0^*. Arcs A_1, A_2, A_3, A_4 are in $\mathcal{P}_3^0$ (and $\mathcal{P}_3$), A_7 is other arc in $\mathcal{P}_3$. Arc A_5 is A_3^*-equivalent and A_8 is A_3^*-complementary. Arc A_6 is in $\mathcal{Q}_3$ while A_9 will be deleted to obtain G_3.*

that $A_i \not\cap A'_i$, but this implies $A'_1 \not\cap A'_2$—contradicting the choice of $\mathscr{I}$ (see beginning of Stage One, Subcase IIa). See Fig. 4 for an example of the G_j construction. The graph G_j has all the relevant overlaps that G had in sections S_{2j-1}, S_{2j}, S_{2j+1} and nothing else.

PROPOSITION 3. *Any circular-arc model for G_j can be expanded to some circular-arc model for G (provided one exists) with the only possible change in the order of endpoints in S_{2j-1}, S_{2j}, S_{2j+1} in G_j's model being for pairs of arcs which overlap the same subset of $\mathscr{P}_j^0$.*

By Proposition 3, a recursive call of our algorithm for G_j (a smaller graph) yields a circular-arc model for G_j (if none exists, then G also has no model) which can be used to determine the two end sections of each arc of $\mathscr{P}_j$.

Subcase IIb. *$\bar{G}$ has an odd hole.* From Stage One we have the circle divided into $4r+2$ sections by the endpoints of the $2r+1$ A^*'s which correspond in $\bar{G}$ to an odd hole. Section S_{2i} is between $\mathrm{Cl}(A^*_{i-r})$ and $\mathrm{Cc}(A^*_i)$ and section S_{2i+1} between $\mathrm{Cc}(A^*_i)$ and $\mathrm{Cl}(A^*_{i-r+1})$. No arc's two endpoints are in the same section nor for any A, A^* can we have $\mathscr{N}(A) \subset \mathscr{N}(A^*)$. Suppose arc A_i does not overlap the cyclicly consecutive set of A^* arcs, $A^*_{i_1}$ through $A^*_{i_2}$. Then $\mathrm{Cl}(A_i)$ must be in S_{2i_1-1} or S_{2i_1} so that A_i overlaps $A^*_{i_1-1}$ but not $A^*_{i_1}$, and similarly $\mathrm{Cc}(A_i)$ must be in $S_{2(i_2+r)}$ or $S_{2(i_2+r)+1}$ to overlap A_{i_2+1} but not A_{i_2} (subscript arithmetic is mod $4r+2$). The point dividing S_{2i_1-1} and S_{2i_1} is $\mathrm{Cl}(A^*_{i_1-r})$. If $A^*_{i_1-r}$ contains $S_{2(i_2+r)}$ then $\mathrm{Cl}(A_i)$ must be in S_{2i_1} (or else A_i would be contained inside $A^*_{i_1-r}$). If $A^*_{i_1-r}$ contains neither $S_{2(i_2+r)}$ nor $S_{2(i_2+r)+1}$, then $\mathrm{Cl}(A_i)$ can be assumed to be in S_{2i_1} if $\mathscr{N}(A^*_{i_1-r}) \subset \mathscr{N}(A_i)$ and $\mathrm{Cl}(A_i)$ must be in S_{2i_1-1} if $\mathscr{N}(A^*_{i_1-r}) \not\subset \mathscr{N}(A_i)$. The point dividing $S_{2(i_2+r)}$ and $S_{2(i_2+r)+1}$ is $\mathrm{Cc}(A_{i_2+r})$. If $A^*_{i_2+r}$ contains S_{2i_1-1} and S_{2i_1}, then $\mathrm{Cc}(A_i)$ must be in $S_{2(i_2+r)}$ (or else A_i would be contained inside $A^*_{i_2+r}$). If $A^*_{i_2+r}$ contains neither $S_{2i_{1-1}}$ nor S_{2i_1}, then we put $\mathrm{Cc}(A_i)$ in $S_{2(i_2+r)}$ if $\mathscr{N}(A^*_{i_2+r}) \subset \mathscr{N}(A_i)$ and in $S_{2(i_2+r)+1}$ if not. If $i_1 = i_2$ and also $\mathrm{Cl}(A_i) \in S_{2i_1}$ and $\mathrm{Cc}(A_i) \in S_{2(i_1+r)}$, we call A_i an $A^*_{i_1}$-complementary arc. There remains the situation where either $A^*_{i_1-r}$ contains $S_{2(i_2+r)+1}$ but not $S_{2(i_2+r)}$ or $A^*_{i_2+r}$ contains S_{2i_1-1} but not S_{2i_1}. Either situation is equivalent to $i_2 = i_1+1$ where $A^*_{i_1-r} = A^*_{i_2+r} = \text{some } A^*_j$. If $\mathscr{N}(A^*_j) \subset \mathscr{N}(A_i)$, then we can assume $\mathrm{Cc}(A_i) \in S_{2j}$ and $\mathrm{Cl}(A_i) \in S_{2(j+r)}$ and we call A_i an A^*_j-equivalent arc. If $\mathscr{N}(A^*_j) \not\subset \mathscr{N}(A_i)$, then we put A_i in the set $\mathscr{P}_j^0$ (these arcs will be handled as in Subcase IIa).

Assume all A_i not overlapping all A^*'s have been processed (or put in some $\mathscr{P}_j^0$). Next suppose A_i overlaps all A^*'s. Consider the set of arcs $\mathscr{E}_i$ which A_i does not overlap. Since A_i overlaps all A^*'s, each $A \in \mathscr{E}_i$ cannot overlap all A^*'s. So, except for possible arcs in the $\mathscr{P}_j^0$'s, the arcs in $\mathscr{E}_i$ already have their end sections determined. To overlap all A^*'s, A_i must physically cover at least half the $4r+2$ sections. So the arcs in $\mathscr{E}_i$ range over at most half the sections of the circle. Let S_{i_1} and S_{i_2} be the counterclockwise and clockwise, respectively, section ends of the range of $\mathscr{E}_i$ in some hypothetical completed model for G. If these range ends to not depend on the placement of arcs in any $\mathscr{P}_j^0$'s, then we can put $\mathrm{Cl}(A_i)$ in S_{i_1} and $\mathrm{Cc}(A_i)$ in S_{i_2} (A_i will be the complement on the circle of the union of arcs in $\mathscr{E}_i$). Suppose just one of the ends of the range depends on an arc in a $\mathscr{P}_j^0$. Then this end section of the range is just inside or outside one of the ends of A^*_j and the other end of the range is fixed beyond the other end of A^*_j. Then to assure that A_i overlaps A^*_j, the end section dependent on the $\mathscr{P}_j^0$ arc must be inside A^*_j. For example if $i_1 = 2j$ or $2j+1$ because of a $\mathscr{P}_j$ arc and i_2 is determined, then $i_1 = 2j+1$ and so $\mathrm{Cl}(A_i)$ is in S_{2j+1} and $\mathrm{Cc}(A_i)$ is in S_{i_2}. If S_{i_1} and S_{i_2} are each dependent on a different $\mathscr{P}_j^0$, $\mathscr{P}_{j_1}^0$ and $\mathscr{P}_{j_2}^0$, respectively, then the same argument applies to assure A_i overlaps $A^*_{j_1}$ and $A^*_{j_2}$ and so $\mathrm{Cl}(A_i)$ is in S_{2j_1+1} and $\mathrm{Cc}(A_i)$ is in $S_{2(j_2+r)-1}$. If S_{i_1} and S_{i_2} are dependent on the same $\mathscr{P}_j^0$ (and now $\mathscr{E}_i \subseteq \mathscr{P}_j^0$), we put all such A_i's along with the set $\mathscr{P}_j^0$ in the set $\mathscr{P}_j$.

We find end sections for the arcs in $\mathscr{P}_j$ just as in Subcase IIa. We define G_j as follows. Arcs A^*_{j+r} and A^*_{j-r} (the two A^*'s not overlapping A^*_j) are combined into the arc A_0^*. G_j contains, A_j^*, A_0^*, $\mathcal{P}_j$, the A_j^*-equivalent and A_j^*-complementary arcs, the set $\mathcal{R}_j$ of arcs A with: (a) $\mathrm{Cc}(A) \in S_{2j}$, $\mathrm{Cl}(A) \in S_{2(j+r)-1}$ or $\mathrm{Cc}(A) \in S_{2j+1}$, $\mathrm{Cl}(A) \in S_{2(j+r)}$ and (b) $\mathrm{Cl}(A) \in S_{2j}$, $\mathrm{Cc}(A) \in S_{2(j+r)-1}$ or $\mathrm{Cl}(A) \in S_{2j+1}$, $\mathrm{Cc}(A) \in S_{2(j+r)}$—call case (a) arcs the set $\mathcal{R}_j^0$ (in Subcase IIa, the $\mathcal{R}_j$ arcs were part of $\mathcal{P}_j$ but in Subcase IIb, because A_j^* contains several sections, we were able to determine these arcs' end sections by other methods), and the set $\mathcal{Q}_j$ of other arcs which overlap some but not all of $\mathcal{P}_j' = \mathcal{P}_j^0 \cup A_j^* \cup \mathcal{R}_j^0 \cup \{A_j^*\text{-equivalent arcs}\}$. Again we only let a pair A, A' in $\mathcal{Q}_j$ overlap if $\mathrm{Cl}(A_1)$, $\mathrm{Cl}(A_2)$ are both in S_{2j}, S_{2j+1} or $\mathrm{Cc}(A_1)$, $\mathrm{Cc}(A_2)$ both in $S_{2(j+r)-1}$, $S_{2(j+r)}$. (A $\mathcal{Q}_j$ arc cannot reach into the region of sections S_{2j+2} through $S_{2(j+r)-2}$ since all arcs in $\mathcal{P}_j'$ contain this region.) The remaining arcs, including the other A^*'s, are deleted. By the same argument as in Proposition 3, a circular-arc model of G_j, obtained by a recursive call of our algorithm, can be used to determine the two end sections of each arc in $\mathcal{P}_j$.

Stage Three. Arrange endpoints in each section. In this final stage, Case I, Subcase IIa, and Subcase IIb are identical. We are given a list of $2m$ sections determined by pairs of consecutive endpoints of the m A^*'s determined in Stage One. We know the (distinct) sections in which each arc's two endpoints are located from Stage Two. Let us consider the endpoint of the A^* that marks the clockwise end of section S_i to be in S_i. As endpoints within sections are positioned, we shall refine the sections, dividing a section into several subsections. To make subscripts available to fit the new subsections into the cyclic order of the sections, we renumber section S_i as section S_{n_i}, where n_i equals the number of endpoints in old S_1 through old S_i. Put all the newly numbered sections containing more than one endpoint, i.e. all S_{n_i} such that $n_i > n_{i-1} + 1$, in the list $\mathcal{L}$ (in any order). Repeatedly go through the list $\mathcal{L}$ (in order) and perform the following analysis of each section in $\mathcal{L}$. Stop processing $\mathcal{L}$ when $\mathcal{L}$ is empty or when no new sections were formed on the previous pass through $\mathcal{L}$.

Let S_j be the next section in $\mathcal{L}$ to be analyzed. If all arcs with endpoints in S_j have their other endpoints in the same other section (as sections are currently constituted), then skip S_j and go to the next section in $\mathcal{L}$. Otherwise remove S_j from $\mathcal{L}$ and proceed as follows. Let $\mathcal{B}$ be the set of arcs B with $\mathrm{Cl}(B) \in S$ and $\mathcal{D}$ the set of arcs D with $\mathrm{Cc}(D) \in S_j$. We define an order $\leqq$ on $\mathcal{B} \cup \mathcal{D}$, where $A \leqq A'$ means that the endpoint of A in S_j must be on the counterclockwise side of the endpoint of A' in S_j:

$B_1 \leqq B_2$ if $\mathcal{N}(B_1) \subset \mathcal{N}(B_2)$ and $\mathrm{Cc}(B_1)$, $\mathrm{Cc}(B_2)$ are in different sections;
$D_1 \leqq D_2$ if $\mathcal{N}(D_2) \subset \mathcal{N}(D_1)$ and $\mathrm{Cl}(D_1)$, $\mathrm{Cl}(D_2)$ are in different sections;
$B \leqq D$ if $B \not\cap D$ and $\mathrm{Cc}(B)$, $\mathrm{Cl}(D)$ are in different sections.
$D \leqq B$ if $B \cap D$ and either: (i) for all $D' \in \mathcal{D} - \mathcal{N}(B)$, $\mathcal{N}(D') \subseteq \mathcal{N}(D)$ and $\mathrm{Cc}(B)$, $\mathrm{Cl}(D)$ are in different sections, or (ii) B and D do not extend far enough to be able to overlap at their other ends (going clockwise around the circle from S_j, the other end section of D comes before the other end section of B).

Note that in case (i) of $D \leqq B$ we assume in our model that D physically contains all $D' \in D - \mathcal{N}(B)$ since $\mathcal{N}(D') \subset \mathcal{N}(D)$ and then we can let B safely overlap D in S_j. All other conditions for $A \leqq A'$ must make the endpoint of A in S_j be counterclockwise to that of A' in a circular-arc model. Now we define the transitive closure $\overset{*}{\leqq}$ of $\leqq$ by: for $A, A' \in \mathcal{B} \cup \mathcal{D}$, $A \overset{*}{\leqq} A'$ if there exists a chain $A = A_1 \leqq A_2 \cdots \leqq A_k = A'$.

PROPOSITION 4. *If G is a circular-arc graph, then any pair of arcs A, A' in $\mathcal{B} \cup \mathcal{D}$ with other endpoints in different sections are comparable in $\overset{*}{\leqq}$, i.e., $A \overset{*}{\leqq} A'$ or $A \overset{*}{\leqq} A$.*

So Proposition 4 says that $\overset{*}{\leqq}$ yields a total linear ordering of $\mathcal{B} \cup \mathcal{D}$ except that some elements in the ordering are clusters of arcs with the same other end section. Put the endpoint(s) of each single arc or cluster in this ordering of S_j in its own section with the endpoint of the first arc in the ordering in section S_{j-n_j+1}, where n_j is the number of arcs in S_j, the endpoint of the second arc in S_{j-n_j+2}, etc., or if the first element in the ordering is a cluster of k arcs, their endpoints go in section S_{j-n_j+k}, etc. Put each new section with more than one arc at the end of $\mathscr{L}$. Note that by assumption not all arcs in S_j have the same other end section and so S_j is subdivided into at least two new sections.

If the algorithm goes through list $\mathscr{L}$ without creating new sections (but $\mathscr{L}$ is not empty), it is because all the arcs with endpoints in one section have their other endpoints in the same other section. Let S_p and S_q be two such sections in $\mathscr{L}$. The arcs ending in S_p and S_q form a bipartite graph, call it $G_{p,q}$. We now recursively call our algorithm to build a circular-arc model for each $G_{p,q}$ (if the algorithm fails for $G_{p,q}$ then clearly G is not a circular-arc graph). It is straightforward to deform the model for $G_{p,q}$ so that all endpoints occur in regions corresponding to where S_p and S_q belong in the model for G.

The cyclic ordering of the endpoints in the model of $G_{p,q}$ provides the desired total ordering of endpoints in S_p and S_q which permits us to subdivide S_p and S_q into sections, each with one arc.

Now each endpoint of an arc is in a different section and by listing endpoints by the index of their sections, we have the cyclic order of the $2n$ endpoints (of the n arcs) in a circular-arc model for G. That is, we have a circular-arc model for G (if one exists). Since every step in the construction of the model was based on required properties of a circular-arc model or on a choice which we proved (or will prove in § 5) was consistent with some circular-arc model for G, this model we built must be a valid circular-arc model for G if one exists. Now we check the physical overlaps in the model (determined by the endpoints order). If they correspond to the abstract overlap (adjacencies), this is indeed a circular-arc model for G. If not, G is not a circular-arc graph.

3. Efficient implementation of the algorithm. In this section, we shall describe how the algorithms presented in the previous section for testing whether an n-vectex graph G is a circular-arc graph can be implemented to run in $O(n^3)$ time and with $O(n^2)$ storage locations. The original algorithm of Fulkerson and Gross [4] for testing interval graphs ran in $O(n^4)$ time and tested for the consecutive 1's property in an $n \times n$ matrix in $O(n^3)$ time. With special data structures and a new fast clique generation algorithm, Booth and Lueker [3] were able to implement both interval graph and consecutive 1's tests in $O(n^2)$ time (which is the best possible result). We shall indicate at the end of this section what is required to make our algorithm run in $O(n^2)$ time. Of course, the real difficulty in our algorithm is not with its speed but its length. (For this reason, we do not go into great detail in discussing the implementation.)

As in section two, we will speak of the n arcs rather than n vectices in the graph G and of (abstract) overlaps rather than adjacencies. Let $A(G)$ denote the overlap (adjacency) matrix of G—entry (i, j) is $1 \Leftrightarrow A_i \cap A_j$. Our algorithm's source of information about G will come from the matrix M: $m_{ij} = |\mathcal{N}(A_i)|$, $m_{ij} = 1$ if $\mathcal{N}(A_i) \subseteq \mathcal{N}(A_j)$, $= -1$ if $\mathcal{N}(A_j) \subseteq \mathcal{N}(A_i)$, $= 0$ if $A_i \not\cap A_j$, $= \infty$ otherwise. M is easily obtained from $M^* = A(G) \cdot A(G)$. It takes $O(n^3)$ steps to compute M^* from $A(G)$ (or $O(n^{2.81})$ with Strassen's fast matrix multiplication) and $O(n^2)$ steps to obtain M from M^* and $A(G)$.

First the algorithm preprocesses G to eliminate equivalent vertices and vertices overlapping all other vertices. Actually m_{ij} is ill-defined if $N(A_i) = N(A_j)$ and so we naturally pick up equivalent vertices in building M from M^*. It takes $O(n^2)$ steps to do this preprocessing (including reindexing of the remaining arcs so that they are consecutively indexed from 1 through some number n'; rows and columns of M should be indirectly indexed so that in preprocessing and during recursive calls of the algorithm for a subgraph of G, no entries in M need be moved). Next the algorithm must decide which of Case I, Subcase IIa, or Subcase IIb applies to G. There is a standard way to test $\bar{G}$ for bipartiteness in $O(n^2)$ steps. If $\bar{G}$ is bipartite, the test yields a bipartition $\mathscr{B}$, $\mathscr{D}$ to be used in Case I. Further using M, we easily obtain in $O(n^2)$ steps the set of arcs $A^*_{-q}, \cdots, A^*_0, \cdots, A^*_r$ defined in Stage One of Case I. (Note that if D is a candidate for A^*_i, i.e., $D \cap B^*_{i-1}$ and $D \not\cap B'_i$, then D is not a candidate for any other A^*_j, $j \neq i$.)

If $\bar{G}$ is not bipartite, the test yields a triangle or odd hole of $\bar{G}$. A triangle means Subcase IIa applies. The arcs A^*_1, A^*_2, A^*_3 corresponding to the triangle in $\bar{G}$ are an independent set $\mathscr{I}'$ of G. We first replace any of these A^*_i by $A^{*\prime}_i$ if $\mathcal{N}(A^{*\prime}_i) \subset \mathcal{N}(A^*_i)$ and $A^{*\prime}_i$ is the minimal such arc. Now we successively augment $\mathscr{I}'$ with arcs A^*_k such that $A^*_k \not\cap A^*_j$, all $j < k$ and A^*_k is the minimal such arc. Finally we obtain a maximal independent set $\mathscr{I}''$ such that $A \notin \mathscr{I}'' \Rightarrow$ there exists $A^*_i \in \mathscr{I}''$ with $A^*_i \cap A$. Now for each $A \notin \mathscr{I}''$, we put A in the set $\mathscr{V}_i$ if A^*_i is the only arc in $\mathscr{I}''$ which A overlaps. Then check in each $\mathscr{V}_i$ to see if there exist (minimal) A, $A' \in V_i$ with $A \not\cap A'$; if such a pair is found, we replace A^*_i by A and A' in $\mathscr{I}''$. The result is the set $\mathscr{I}$ which has been obtained from the original triangle in $O(n^2)$ steps. This is the r-arc independent set $\mathscr{I}$ required in Subcase IIa. In Stage One of Subcase IIa, we form matrix M_0 and test it for the circular 1's property in $O(n^2)$ steps by reducing the circular 1's test to a test for consecutive 1's (as described in § 1), and by using the Lueker–Booth consecutive 1's test. Finding the arc A'_i needed to obtain $G'_{i+1}, M_{i+1}, \mathscr{I}_{i+1}$ from $G'_i, M_i, \mathscr{I}_i$ is not hard with the Booth–Lueker data structure for representing all the valid row arrangements of M_i; arc A'_i is found and $G'_{i+1}, M_{i+1}, \mathscr{I}_{i+1}$ formed in $O(n^2)$ steps. The Booth–Lueker data structure requires $O(n)$ storage locations to represent the possible row arrangements of M_i, equivalently, possible cyclic orders of $\mathscr{I}_i$. We need to save all r structures (for each M_i) but the matrices M_i need not be saved. So only $O(n^2)$ extra locations are needed in Stage One for ordering the set $\mathscr{I}$. The total number of steps in building and testing the M_i is $O(rn^2)$, or at worst $O(n^3)$. Obtaining a cyclic order for $\mathscr{I}$ recursively from extensions of the $\mathscr{I}_i$, as described in Proposition 1, is readily performed in $O(n^2)$ steps with the Booth–Lueker data structures.

If the bipartite test for $\bar{G}$ yields an odd hole, then we must first be sure no other arc is contained in one of the arcs corresponding to this odd hole (similar to the inclusion test in Subcase IIa). Any time containment is found with $\mathcal{N}(A^{*\prime}) \subset \mathcal{N}(A^*_i)$, then $A^{*\prime}_i$ replaces A^*_i and we look in $\bar{G}$ to see if the vertex corresponding to $A^{*\prime}_i$ forms an odd hole with a proper subset of the remaining vertices of the original odd hole. If so, we now use this new odd hole. If a triangle is formed, we go to Subcase IIa. This processing requires $O(n^2)$ steps. This completes Stage One for all cases.

The first step in Stage Two, Case I is determining the levels of the higher and lower ends of each arc by the given formulas. First the sets of arcs A, $\mathcal{N}(A) \not\subset \mathcal{N}(A^*_i)$ need to be determined for each B^*_i and D^*_i. Now the end levels can be determined for an arc from entries in the arc's row in M in $O(n)$ steps; for all arcs in $O(n^2)$ steps. Next we must apply the tests of relations S and O to build the components determined by these relations. We check all pairs B, D for conditions (i) and (ii). These two build a bipartite graph with $O(A, A')$ like a normal graph edge and $S(A, A')$ a special nonstandard edge which forces A and A' to be on the same side of the bipartition and in the same component of the graph (one could think of $S(A, A')$ as representing a path of length 2 from A to A'). Denote the components generated by just condition (i) and condition (ii) $\mathscr{C}_1, \mathscr{C}_2, \cdots, \mathscr{C}_q$. They are determined in $O(n^2)$ steps. Store copies of the components for use in checking condition (vii) later. Now for condition (v), we check each $\mathscr{C}_i$ for pairs B, D with $l(B) = h(D)$ and $B \cap D$. Record in sets $\mathscr{B}_0$ and $\mathscr{D}_0$ each B and D involved in such pairs for later use in condition (vi). All components with such pairs are joined by condition (v) into one component. Next we apply condition (vi). We omit the details but D_2 in (vi) is the upper barrier of B_1 (defined in the proof of Proposition 2) and the possible B_1, D_1 pairs were recorded in checking (v). From these facts, an $O(n^2)$ test of (vi) is easy to build.

To apply condition (iv) (to combine various pairs of the current set of components), we need a $p \times p$ matrix W, where p is the number of levels. For each $\mathscr{C}_i$ in the current set of components we perform the following "W-processing." For each $B \in \mathscr{C}_i$, we set $w_1(B) = \min\{l(D) : D \in \mathscr{C}_i,\ h(D) = h(B),\ l(D) < l(B)$, and $D \not\cap B\}$ and we set $w_2(B) = \min\,(h(D) : D \in \mathscr{C}_1,\ l(D) = l(B),\ h(D) < h(B)$, and $D \not\cap B\}$; $w_1(B)$ or $w_2(B)$ is undefined if none of the required D's exist. To speed this step we first should make lists of D's with $l(D) = j$ for each level j and lists with $h(D) = j$ for each j. Let B_k have the smallest $w_1(B)$ among all B's with $h(B) = k$ (and $w_1(B)$ defined) and B^k have the smallest $w_2(B)$ among all B's with $l(B) = k$ (and $w_2(B)$ defined). The pairs B_k, $w_1(B_k)$ and B^k, $w_2(B^k)$ for all k for which such pairs exist can be found in $O(n_i^2)$ steps, where $n_i = |\mathscr{C}_i|$. Using these pairs, we can efficiently determine for every pair of levels k, k' whether there exist $B, D \in \mathscr{C}_i$ with $B \not\cap D$, $h(B) = h(D) = k$ and $l(D) < k' \leqq l(B)$, or there exist $B, D \in \mathscr{C}_i$ with $B \not\cap D$, $l(B) = l(D) = k$ and $h(D) < k' \leqq h(B)$. If so, set $w_{k,k'} = i$. Further set a flag at $w_{k,k'}$ if $k' = l(B)$ or $h(B)$. If we try to set some $w_{k,k'}$ equal to i when it was already set to i', then we do not change $w_{k,k'}$ but set a (second) flag and remove the first flag if it was set by $\mathscr{C}_{i'}$ but would not be set by $\mathscr{C}_i$. In addition if $\mathscr{C}_{i'}$ and $\mathscr{C}_i$ both set $w_{k,k'}$, then condition (iv) is satisfied and components $\mathscr{C}_{i'}$ and $\mathscr{C}_i$ are combined. This combination will be done after all the $\mathscr{C}$'s have been W-processed. After noting that $\mathscr{C}_{i'}$ and $\mathscr{C}_i$ are to be combined (and where they are to be combined), we continue the W-processing of $\mathscr{C}_i$. If G is a circular-arc graph, no more than two components can try to set any particular $w_{k,k'}$ (the second flag will allow us to detect a third component's attempt to set a $w_{k,k'}$ at which time the algorithm terminates). Since each $\mathscr{C}_i$ is processed in $O(n_i^2)$ steps except for setting $w_{k,k'}$'s and since each $w_{k,k'}$ is accessed at most twice (or the algorithm terminates), all $\mathscr{C}$'s can be W-processed in $O(n^2)$ steps. At the end of the W-processing the appropriate pairs of components are combined (possibly concatenating many components into a new single component). We go through W setting each (nonzero) entry to the correct new component number.

Next using W and the fact that B_2 and D_2 will be lower and upper barriers of D_1 and B_1, resp., in condition (iii), we get an $O(n^2)$ test for (iii) (details are omitted)). Now we set to 0 any entry of W for which the first flag is set. We call the remaining nonzero entries in W type 1 entries. They will be used to complete the test for condition (iv), now against pairs B, D with $l(B) = l(D) = k$, $h(D) = h(B) = k'$, $B \not\cap D$. However, such B, D pairs are also part of condition (vii), and so we will check these conditions at once. For each component $\mathscr{C}_i$, we look for sequences A_i, $i = 1, \cdots, m$, as in condition (vii). Starting with a candidate for A_2, we do depth-first searches in the A_2's original component generated by (i) and (ii) (copies were saved) for possible A_1 and A_m to form the sequences. If we first make lists of B's and D's with a given higher or lower end, then in all searches no pair of arcs need be checked for overlap (or nonoverlap) more than twice. If such a sequence is found, we set w_{k_1,k_2} equal to i (if it is not already set to i). We call this a type 2 entry (to be used for condition (vii)). It is easy to check that if G is a circular-arc graph it is impossible for w_{k_1,k_2} already to have a different (nonzero) value as a type 1 or type 2 entry. Each $\mathscr{C}_i$ can be processed for type 2 entries in $O(n_i^2)$ steps (since no pair is checked more than two times). Now we can go through each $\mathscr{C}_i$ looking for pairs $B, D \in \mathscr{C}_i$ with $l(B) = l(D) = k$ and $h(D) = h(B) = k'$, and if entry $w_{k,k'}$ or entry $w_{k',k}$ equals j, $0 < j \neq i$, then the components should be combined by condition (iv) if the entry is type 1 or by (vii) if the entry is type 2. This testing can be done in $O(n^2)$ steps.

This completes the formation of the components generated by the relations S and 0. A total of $O(n^2)$ steps was required. By giving arbitrary left-right orientation to the higher ends of arcs in each component, we can now assign clockwise and counterclockwise end sections for each arc as described in the algorithm. This completes Stage Two, Case I.

The location of end sections of arcs in Subcase IIa and Subcase IIb of Stage Two is straightforward except in the case of the arcs in $\mathscr{P}_j$ sets, for which we must resort to a recursive call of the algorithm to try to build a circular-arc model for G_j. We want to show that $M(G_j)$, the M matrix for G_j, can be set up and the whole algorithm for G_j (excluding recursive calls during the processing of G_j) run in only $O(k_j^2)$ steps, where $k_j = |G_j|$. $M(G_j)$ is the same as M for all arcs in G_j except for main diagonal entries and for arcs $\mathcal{Q}_i$ (and of course the new arc A_0^* whose row and column in $M(G_j)$ is easily determined). The $\mathcal{Q}_j$ arcs split into two (mutually nonoverlapping) cliques as described in the algorithm. The only problem then is to determine possible containment for pairs of $\mathcal{Q}_j$ arcs in the same clique. Such $\mathcal{Q}_j$ arcs can differ only in number of $\mathscr{P}'_j$ arcs they overlap (if G is a circular arc graph, the subsets of $\mathscr{P}'_j$ arcs overlapped by the various $\mathcal{Q}_j$ arcs in the same clique must be ordered by inclusion; if the number of $\mathscr{P}'_j$ arcs overlapped by two such $\mathcal{Q}_j$ arcs is the same, the two arcs are equivalent in G_j and one should be deleted in preprocessing). Then $M(G_j)$ can be obtained from M in $O(k_j^2)$ steps. (Further, if these $\mathcal{Q}_j$ arcs are involved in recursive calls for some (G_j), their M entries will not change again.) The only part of the circular-arc algorithm after initially computing M that does (or will) require $O(n^3)$ steps is in Stage One, Subcase II when we had to compute $\{G'_i, M_i, \mathscr{S}_i\}$, but since G_i can contain at most three independent arcs, which have only one cyclic order, G_j (and any $(G_j)_{j'}$) will not need to compute $\{G'_i, M_i, \mathscr{S}_i\}$. While the algorithm needs only $O(k_j^2)$ steps to process G_j without further recursion, with further recursion it could take $O(k_j^3)$ steps (each recursion call involves a smaller graph). Since an arc of G can be involved in at most two different G_j's, the total processing of all G_j's requires at most $O(n^3)$ steps. This completes Stage Two.

Now we come to Stage Three. We first re-number the sections and make the list $\mathscr{L}$ as described in the algorithm. On the first pass through $\mathscr{L}$, a cyclicly ordered (doubly-linked) list $\mathscr{L}_j$ is made of the other endsections for the arcs in S_j; and for each such other endsection S_i, a (doubly-linked) sublist $\mathscr{L}_{ji}$ is made of the arcs with one end in S_j and the other end in S_i ($\mathscr{L}_{ji}$ is actually two sublists, one for B arcs of S_j and one for D arcs pf S_j). Suppose S_j is the next section chosen from $\mathscr{L}$ for processing. Then we take the first B (if any exists) from each sublist $\mathscr{L}_{ji}$ and order these B's (in a doubly-linked list) as follows. Suppose B_1's counterclockwise end section is farther (in the counterclockwise direction) from S_j than B_2's. Then B_2 goes before B_1 in this order if $B_2 \leqq B_1$ and otherwise B_1 goes before B_2 (for $B_1 \overset{*}{\leqq} B_2$, i.e., there must exist D in S_j with $B_1 \leqq D \leqq B_2$). With $m-1$ of these B's ordered, we go through the current order (starting with the first arc in the order) applying the preceding test for the mth B with each successive B in the order until we find a B' before which B must go, in which case we insert the mth B just before that B' in the order. When these B's are ordered, we then take all the remaining B's from each of the sublists $\mathscr{L}_{ji}$ and do the following with each B. Let B^0 be the B now to be inserted in the order. Suppose B^0 came from sublist L_{ji}; B_i was the first $\mathscr{B}$ arc in $\mathscr{L}_{ji}$ (which has been placed in the order as just described); and B'_i, B''_i are the B's just before and after, resp., B_i in the current order. If B'_i goes before B^0 and B^0 goes before B''_i (by the tests described above), then B^0 joins B_i as part of an (unordered) cluster in the order. If B^0 goes before B'_i, we start moving backwards through the order looking for a position at which to insert B^0. If B''_i goes before B^0, we move forward. If at any time, including tests with B'_i or B''_i, we are testing B^0 against a cluster in the order and the cluster consists of B's from a different sublist then B^0's, then we test B^0 against all B's in the cluster. If the test results are not all the same, we then break up the cluster and put B in between the arcs that go before it and the arcs that go after it. If in moving backwards (forwards) we come to test B^0 against a cluster of arcs from its own sublist, then we skip that cluster and move on backwards (forwards) to the next arc or cluster in the order. If tests show that B^0 goes after (before) this arc or cluster, we put B^0 in the cluster of arcs in its own sublist which we skipped.

It is readily checked that this order is consistent with $\overset{*}{\leqq}$ (the only difference is that some clusters of this order can be split up in $\overset{*}{\leqq}$ by D arcs). We analogously order the D's in S_j. Finally we merge the two orders of the B's and D's as follows: Let $\mathscr{B}_1 \overset{*}{\leqq} \mathscr{B}_2 \overset{*}{\leqq} \cdots \overset{*}{\leqq} \mathscr{B}_p$ and $\mathscr{D}_1 \overset{*}{\leqq} \mathscr{D}_2 \overset{*}{\leqq} \cdots \overset{*}{\leqq} \mathscr{D}_q$ be our order of the B's and D's resp., where $\mathscr{B}_i$ and $\mathscr{D}_j$ are either single arcs or clusters. We write $\mathscr{B}_i \leqq \mathscr{D}_j$ (or $\mathscr{D}_j \leqq \mathscr{B}_i$) if $D \leqq B$ (or $D \leqq B$) for all $B \in \mathscr{B}_i$, $D \in \mathscr{D}_j$. We shall successively position $\mathscr{D}_1$, $\mathscr{D}_2$; etc. in the order of the $\mathscr{B}$'s. First test $\mathscr{D}_1$ with $\mathscr{B}_1$. If $\mathscr{D}_1 \leqq \mathscr{B}_1$, insert $\mathscr{D}_1$ before $\mathscr{B}_1$ in the order. If $\mathscr{B}_1 \leqq \mathscr{D}_1$, test $\mathscr{D}_1$ with $\mathscr{B}_2$, then $\mathscr{B}_3$, etc. until $\mathscr{D}_1 \leqq \mathscr{B}_i$ and then insert $\mathscr{D}_1$ just before $\mathscr{B}_i$ in the ordering. If there exists $D \in \mathscr{D}$, $B, B' \in \mathscr{B}_i$ with $B \leqq D \leqq B'$, then break $\mathscr{B}_i$ into two clusters $\mathscr{B}_i^{(1)}$, $\mathscr{B}_i^{(2)}$ with $\mathscr{B}_i^{(1)} \leqq D \leqq \mathscr{B}_1^{(2)}$ and continue testing the rest of $\mathscr{D}_1$ on $\mathscr{B}_i^{(1)}$. If there exist $D, D' \in \mathscr{D}_1$, $B \in \mathscr{B}_1$ with $D \leqq B \leqq D'$, then break $\mathscr{D}_1$ into two clusters, a new $\mathscr{D}_1$ and a new $\mathscr{D}_2$ (preceding the old $\mathscr{D}_2$ in the D's order) with $\mathscr{D}_1 \leqq B \leqq \mathscr{D}_2$ and continue testing this new $\mathscr{D}_1$ with $\mathscr{B}_i$. Implicit in this breaking procedure is that we test $\mathscr{D}_1$ with $\mathscr{B}_i$ by testing one D in $\mathscr{D}_1$ against all B's in $\mathscr{B}_i$, then a B in $\mathscr{B}_i$ against all other D's in $\mathscr{D}_1$, then another D against all other B's etc. If $\mathscr{D}_1$ and $\mathscr{B}_i$ are from the same sublist $\mathscr{L}_{ji}$, then we skip $\mathscr{B}_i$ and test $\mathscr{D}_1$ against $\mathscr{B}_{i+1}$. If $\mathscr{D}_1 \leqq \mathscr{B}_{i+1}$, then $\mathscr{D}_i$ is merged with the cluster for $\mathscr{B}_i$. After $\mathscr{D}_1$ is inserted in the order we repeat the procedure with $\mathscr{D}_2$. Since $\mathscr{D}_1 \overset{*}{\leqq} \mathscr{D}_2$, we can start testing $\mathscr{D}_2$ with the $\mathscr{B}$ immediately after the position of $\mathscr{D}_1$ in the order, or if $\mathscr{D}_1$ was merged with $\mathscr{B}_i$, we start testing $\mathscr{D}_2$ with $\mathscr{B}_i$ (and if any $D \in \mathscr{D}_2$ causes $\mathscr{B}_i$ to be broken up, the D arcs of $\mathscr{D}_1$ go into $\mathscr{B}_i^{(1)}$; if $\mathscr{D}_2 \leqq \mathscr{B}_i$, then $\mathscr{D}_2$ is inserted between $\mathscr{D}_1$ and $\mathscr{B}_i$ which are split apart). Next we place $\mathscr{D}_3$, $\mathscr{D}_4$, and so on in the order. This completes construction of the $\overset{*}{\leqq}$ order.

Each single arc or cluster in the order now becomes its own section. This requires updating the $\mathscr{L}_j$ and $\mathscr{L}_{ji}$ lists for the other endsections of arcs in S_j. The procedure to build the $\overset{*}{\leqq}$ order in S_j along with this updating (for which we require the position of each arc in the two sublists in which the arc occurs) requires $O(k_j^2)$ steps, where $k_j = |S_j|$. However, arcs of S_j which are in the same cluster (section) are never compared, and so in the repeated processing of all S_j's on successive passes through $\mathscr{L}$, arcs are only compared when they will be put into different sections. It follows that the total processing of all S_j's on all passes through L, takes only $O(n^2)$ steps (to make the searches through $\mathscr{L}$ to find S_j's to be processed efficient, we need to maintain a sublist L' of S_j's in $\mathscr{L}$ with $|\mathscr{L}_j| > 1$). If searches of $\mathscr{L}$ stop with $\mathscr{L}$ nonempty, then we must recursively call the algorithm to process a bipartite subgraph $G_{p,q}$ (described in the algorithm). The submatrix of M corresponding to the arcs in $G_{p,q}$ is $M(G_{p,q})$ except that the m_{ii} entries in the B and D parts of $G_{p,q}$ are each off by a constant factor. Since no other part of our algorithm for Case I graphs, besides determining M, requires more than $O(n^2)$ steps, all the recursion involved in $G_{p,q}$ (recursions in Stage Three for $G_{p,q}$, etc.) will involve at most $O(n^3)$ steps.

From these $G_{p,q}$ subgraphs we complete the cyclic order of the $2n$ endpoints of the arcs in G. Finally, we check (in $O(n^2)$ steps) whether the physical overlaps dictated by the cyclic order of endpoints correspond to the adjacencies (abstract overlaps) of G. If so, G is a circular-arc graph; if not, G is not. Note that there are many places in the algorithm where we could detect if G is not a circular-arc graph, but we only are interested in this possibility when it naturally forces the algorithm to terminate, or when, as in Stage Two, Case I with checking for three accesses to a W-entry during W-processing, the algorithm would run too long.

The preceding implementation requires $O(n^3)$ steps and $O(n^2)$ storage locations (only a small constant number of matrices are used at any one time). There are two steps in the algorithm that directly require $O(n^3)$ steps: making M, and developing the sequence $\{G_i', M_i, \mathcal{I}_i\}$ in Stage One. Subcase IIa. In addition there are two steps that implicitly require $O(n^3)$ steps: the recursion for G_j's to determine end sections of arcs in $\mathcal{P}_j$'s in Stage Two, Subcases IIa, IIb, and the recursion for $G_{p,q}$ subgraphs in Stage Three. By careful preprocessing of $G_{p,q}$ (using appropriate shortcuts), one can show (by a lengthy analysis) that all this recursion can be done in $O(n^2)$ steps. It also appears that with careful analysis, the sequence $\{G_j, M_i, \mathcal{I}_i\}$ and recursion to find end sections of arcs in $\mathcal{P}_j$'s can be determined in $O(n^2)$ steps. However, we know of no way to compute M other than from $M' = A(G)^2$ which requires $O(n^3)$ steps (or $O(n^{2.81})$ with Strassen's fast multiplication; actually Pan recently reduced the number of steps to $O(n^{2.795})$).

As mentioned earlier, simplification rather than greater speed is what is really needed, since the constant in $O(n^3)$ is likely to be horrendous.

4. An example. Figure 5 presents the M matrix for a graph G to be tested by the algorithm.' G is not bipartite. When the bipartite test fails, suppose it yields the triangle in $\bar{G}$ (independent set in G) of A_2, A_5, A_6 ($\bar{G}$ has no odd holes). So Subcase IIa applies. However, $\mathcal{N}(A_1) \subset \mathcal{N}(A_5)$ and $\mathcal{N}(A_3) \subset \mathcal{N}(A_6)$; so our new independent set is A_1, A_2, A_3, to which we can add A_4. So A_1, A_2, A_3, A_4 are $A_1^*, A_2^*, A_3^*, A_4^*$, respectively, for this graph. In seeking a cyclic order for $A_1^*, A_2^*, A_3^*, A_4^*$, we build M_0. The fragment of M_0 shown in Fig. 6 shows that arcs A_5 and A_6 determine the cyclic order uniquely (up to inversion). Let the space between A_4^* and A_1^* be section S_1, A_1^* contain section S_2, the space between A_1^* and A_2^* be section S_3, and so on up to arc A_4^* being S_8. This completes Stage One.

In Stage Two, we readily find that A_5 goes from S_8 to S_3, A_6 from S_6 to S_8, and A_7 goes from S_3 to S_5 and is an A_2-equivalent arc. However, A_i, $8 \leqq i \leqq 14$, are all in $\mathcal{P}_2$. Now $\mathcal{Q}_2 = A_5$, A_0^* is the combination of A_3, A_4, A_1, while A_6 is dropped to form G_2'. Since A_8, A_9, A_{10} overlap A_5, they must go from section S_3 to S_4. Then $A_{11}, A_{12}, A_{13}, A_{14}$ which overlap all A^*'s but not A_8 must go from S_4 to S_3 (we will not follow the algorithm through G_2' to confirm this; however, we note that G_2' is a Case I graph with cliques $\mathcal{D} = \{A_2^*, A_8, A_9, A_{10}\}$ and $\mathcal{B} = \{A_5, A_{11}, A_{12}, A_{13}, A_{14}, A_0^*\}$). Figure 7 depicts our knowledge of arc endsections (not overlaps within sections). Note that we formally consider the counterclockwise endpoint of A_i^* to be in section S_{2i-1}.

Now we move to Stage Three. Recall Cc(A) and Cl(A) denote the counterclockwise and clockwise endpoints of arc A. We renumber the sections by the cardinality of arc endpoints in each section to get: S_1 stays S_1 containing just Cc(A_1); S_2 stays S_2 containing just Cl(A_1); S_3 becomes S_{12} containing Cc(A_2), Cl(A_5), Cc(A_i), $7 \leqq i \leqq 10$, Cl(A_j), $11 \leqq j \leqq 14$; S_4 becomes S_{20} containing Cl(A_2), Cl(A_i), $8 \leqq i \leqq 10$, Cc(A_j), $11 \leqq j \leqq 14$; S_5 becomes S_{22} containing Cc(A_3), Cl(A_7); S_6 becomes S_{24} containing Cl(A_3), Cc(A_6); S_7 becomes S_{25} containing Cc(A_4); and S_8 becomes S_{28} containing Cl(A_4), Cl(A_6), Cc(A_5). In section S_{12}, the order is $A_7 \overset{*}{\leqq} \{A_i, 8 \leqq i \leqq 14\} \overset{*}{\leqq} A_5 \overset{*}{\leqq} A_2$ and in section S_{20} the order is $\{A_i, 8 \leqq i \leqq 14\} \overset{*}{\leqq} A_2$. In section S_{22}, the order of arcs is $A_7 \overset{*}{\leqq} A_3$; in section S_{24}, $A_6 \overset{*}{\leqq} A_3$; in section S_8, $A_6 \overset{*}{\leqq} A_5 \overset{*}{\leqq} A_4$. Now we have the sections listed in Fig. 8. The clusters in S_{10} and S_{19} require us to recursively call the algorithm to process $G_{10,19}$ consisting of A_i, $8 \leqq i \leqq 14$. This is a Case I graph. Let $\mathcal{B} =$

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}
A_1	6	0	0	0	1	0	0	0	0	0	1	1	1	1
A_2	0	9	0	0	0	0	1	2	2	2	2	2	2	2
A_3	0	0	6	0	0	1	0	0	0	0	1	1	1	1
A_4	0	0	0	7	2	2	0	0	0	0	1	1	1	1
A_5	−1	0	0	2	11	0	2	2	2	2	2	2	2	2
A_6	0	0	−1	2	0	7	0	0	0	0	1	1	1	1
A_7	0	−1	0	0	2	0	10	−1	−1	−1	2	2	2	2
A_8	0	2	0	0	2	0	1	6	1	1	0	0	0	0
A_9	0	2	0	0	2	0	1	−1	8	2	2	0	2	0
A_{10}	0	2	0	0	2	0	1	−1	2	8	2	2	0	0
A_{11}	−1	2	−1	−1	2	−1	2	0	2	2	13	1	1	1
A_{12}	−1	2	−1	−1	2	−1	2	0	0	2	−1	11	2	−1
A_{13}	−1	2	−1	−1	2	−1	2	0	2	0	−1	2	11	−1
A_{14}	−1	2	−1	−1	2	−1	2	0	0	0	−1	1	1	11

FIG. 5

	A_5		A_6
	9	10	12
A_1^*	1	1	0
A_2^*	0	0	0
A_3^*	0	0	1
A_4^*	0	1	1

FIG. 6

$A_{11}, A_{12}, A_{13}, A_{14}$ and $\mathscr{D} = A_8, A_9, A_{10}$. Then $B_0^* = A_{11}$, $D_1^* = A_9$ (or A_{10}), $B_2^* = A_{12}$ (or A_{13}), $B_3^* = A_{14}$, $D_{-1}^* = A_8$. This leaves only one more B-arc A_{13} with $l(A_{13}) = 1$, $h(A_{13}) = 3$ and only one D-arc A_9 with $l(A_9) = 3$, $h(A_9) = 1$. The lower ends of A_{10} and A_{13} must be on opposite sides by condition (i) of relation 0 in Stage Two, giving the model shown in Fig. 9 (A_{10} and A_{13} could have sides of the circle reversed). Thus back in S_{10}, we have $A_8 \overset{*}{\leqq} A_{11} \overset{*}{\leqq} A_{10} \overset{*}{\leqq} A_{13} \overset{*}{\leqq} A_9 \overset{*}{\leqq} A_{12} \overset{*}{\leqq} A_{14}$, and in S_{19},

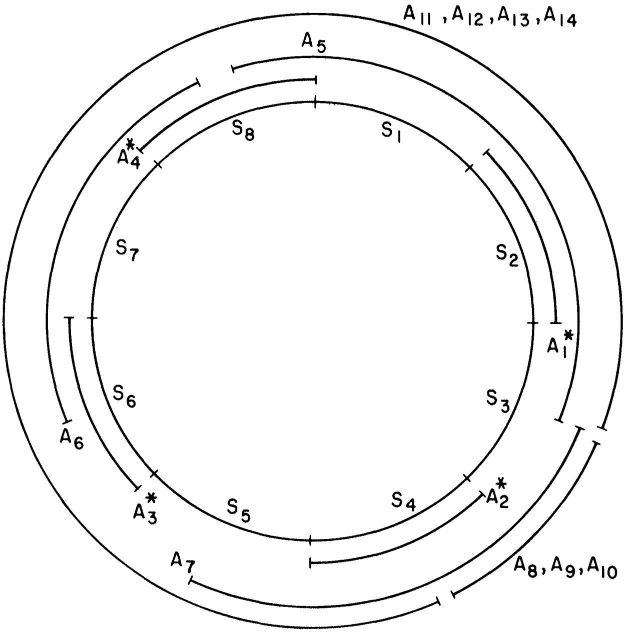


FIG. 7. *General end section information about the arcs.*

Section	Endpoints
S_1	$\mathrm{Cc}(A_1)$
S_2	$\mathrm{Cl}(A_1)$
S_3	$\mathrm{Cc}(A_7)$
S_{10}	$\mathrm{Cc}(A_i)$, $8 \leqq i \leqq 10$ and $\mathrm{Cl}(A_j)$, $11 \leqq j \leqq 14$
S_{11}	$\mathrm{Cl}(A_5)$
S_{12}	$\mathrm{Cc}(A_2)$
S_{19}	$\mathrm{Cl}(A_i)$, $8 \leqq i \leqq 10$ and $\mathrm{Cc}(A_j)$, $11 \leqq j \leqq 14$
S_{20}	$\mathrm{Cl}(A_2)$
S_{21}	$\mathrm{Cl}(A_7)$
S_{22}	$\mathrm{Cc}(A_3)$
S_{23}	$\mathrm{Cc}(A_6)$
S_{24}	$\mathrm{Cl}(A_3)$
S_{25}	$\mathrm{Cc}(A_4)$
S_{26}	$\mathrm{Cl}(A_6)$
S_{27}	$\mathrm{Cc}(A_5)$
S_{28}	$\mathrm{Cl}(A_4)$

FIG. 8

$A_{14} \overset{*}{\leqq} A_{13} \overset{*}{\leqq} A_{10} \overset{*}{\leqq} A_{12} \overset{*}{\leqq} A_9 \overset{*}{\leqq} A_{11} \overset{*}{\leqq} A_8$. We now have a total cyclic order of endpoints of arcs in G. It is readily checked that this order is consistent with the adjacencies of G. So G is a circular-arc graph.

5. Proofs of propositions. In this section we prove the four propositions used in our circular-arc algorithm.

PROPOSITION 1. *Let G be a circular-arc graph with $\{G'_i, M_i, \mathscr{I}_i\}_{i=0}^{=s}$ as described in Stage One, Subcase IIa of the algorithm. If, starting with a cyclic order θ_s of $\mathscr{I}_s$ which*

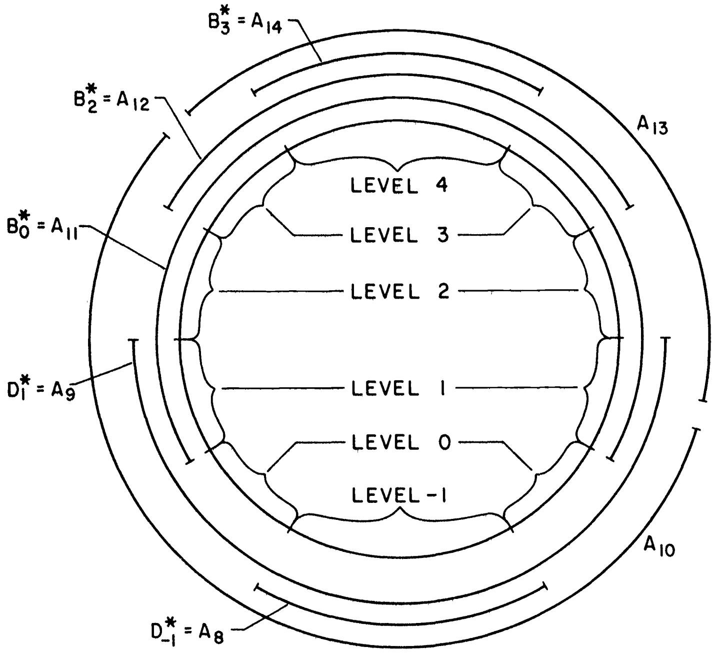


FIG. 9. *Case I model for $G_{10,19}$.*

induces circular 1's in M_s, one recursively picks θ_i an extension of θ_{i+1}, then the final extension θ_0 is a valid cyclic order for $\mathscr{S}$ in some circular-arc model of G.

Proof. Recall that the sequence G_i', M_i, $\mathscr{S}_i$ is found by a reduction process in which we find a minimal A_{i+1}' overlapping A^*'s whose position within a circular-1's-inducing cyclic order of $\mathscr{S}_i$ is not unique, and that we use this A_{i+1}' to obtain G_{i+1}' and $\mathscr{S}_{i+1}$ from G_i' and $\mathscr{S}_i$ by replacing A_{i+1}' and $\mathcal{N}^*(A_{i+1}')$ by the combination arc $A_{i+1}'^*$. If we physically perform this reduction construction, replacing arcs A_{i+1}' and $\mathcal{N}^*(A_{i+1}')$ by an arc $A_{i+1}'^*$ which contains all points that were contained in A_{i+1}' or $\mathcal{N}^*(A_{i+1}')$, then a circular-arc model for G_i' is easily seen to be transformed into a circular-arc model for G_{i+1}'. We call a circular-arc model for G_i' reduced if it can be obtained from $G_0' = G$ by a sequence of such (physical) reduction constructions.

It is not hard to check that if θ_i is any cyclic order of $\mathscr{S}_i$ induced by a reduced model of G_i', then any of the circularity-preserving inversions or cyclic permutations (mentioned in § 2) of θ_i can be induced by other reduced models of G_i'. Since $\mathscr{S}_s$ has a unique cyclic order for inducing circular 1's in M_s, this order must correspond to the cyclic order of $\mathscr{S}_s$ in any reduced circular-arc model for G_s'. We now complete the proof by induction by showing that given a cyclic order θ_{i+1} of $\mathscr{S}_{i+1}$ in some reduced circular-arc model of G_{i+1}', any extension θ_i of θ_{i+1} corresponds to the cyclic order of $\mathscr{S}_i$ in some reduced circular-arc model of G_i.

In θ_i, the arcs in $\mathcal{N}^*(A_{i+1}')$ replace, in some order, the arc A'^* in θ_{i+1}. By the choice of A_{i+1}', there is no arc A in G_i' such that $\mathcal{N}^*(A) \subset \mathcal{N}^*(A_{i+1}')$. If an arc A_j overlaps a subset $\mathcal{D}_j = \mathcal{N}^*(A_j) \cap \mathcal{N}^*(A_{i+1}')$ of $\mathcal{N}^*(A_{i+1}')$, then for columns $2j$ and $2j-1$ of M_i to have circular 1's, the arcs of $\mathcal{D}_j$ must be consecutive in the subset $\mathcal{N}^*(A_{i+1}')$ in θ_i and must be at the end of $\mathcal{N}^*(A_{i+1}')$ in θ_i beside the other arcs in $\mathcal{N}^*(A_j)$. (Or if A_j overlaps all of $\mathscr{S}_i - \mathcal{N}^*(A_{i+1}')$, then $\mathcal{N}^*(A_{i+1}') - \mathcal{D}_j$ must be consecutive in A_i.) For the two columns of M_i corresponding to A_{i+1}' to have circular 1's, each end of A_{i+1}' must be at an end of $\mathcal{N}^*(A_{i+1}')$ in θ_i. The only other constraint on the order of $\mathcal{N}^*(A_{i+1})$ in building the extension of A_i of A_{i+1} is the A_{i+1}'-extension condition, if it applies (which assures that if A'_{i+1} overlaps any arc A^0 not overlapping $\mathcal{N}^*(A'_{i+1})$ and A'_{i+1} has just one end (and $|\mathcal{N}^*(A'_{i+1})| \geqq 2$), then that end of A'_{i+1} is not beside the arcs of $\mathcal{N}^*(A^0)$ in θ_i). A reduced circular-arc model for G'_i in the sequence of reduction constructions (starting from $G'_0 = G$) to get a reduced circular-arc model for G'_{i+1} must have the relative order of the A^*'s in $\mathcal{N}^*(A'_{i+1})$ constrained by exactly the above conditions. Thus the cyclic order θ_i corresponds, up to (permissible) inversion and/or cyclic permutation (described in the algorithm), to the order of $\mathscr{I}_i$ in a reduced circular-arc model for G'_i. Q.E.D.

PROPOSITION 2. *If G is a circular-arc graph, there are circular-arc models for G with each of the* 2^h *possible left-right orientations of the h different components defined by the relations S and O.*

Proof. Let L_i be the set of levels in which the arcs of component $\mathscr{C}_i$ have endpoints. Let $\min L_i$ and $\max L_i$ denote the minimum and maximum levels numbers, respectively, in L_i. We write $L_i \leqq L_j$ (or $L_i < L_j$) if $k \leqq k'$ (or $k < k'$) for all $k \in L_i$, $k' \in L_j$. Then we define the relation on components $\mathscr{C}_i \leqq \mathscr{C}_j$ to mean that there exists a partition of L_j into L_j^l, L_j^h, such that $L_j^l \leqq L_i \leqq L_j^h$ (possibly L_j^l or L_j^h empty). There are three possible cases of $\mathscr{C}_i \leqq \mathscr{C}_j$ to consider. Case (a): L_j^l or L_j^h is empty (i.e., $L_i \leqq L_j$ or $L_j \leqq L_i$ and so also $\mathscr{C}_j \leqq \mathscr{C}_i$) and now $\mathscr{C}_i$ and $\mathscr{C}_j$ are modeled in different levels of the circle; if $\max L_j = k = \min L_i$ then $\mathscr{C}_j$ should use the lower part of level k and $\mathscr{C}_j$ the upper part to assure the required overlaps and nonoverlaps between arcs in different components (we only have to worry here about conditions (i) and (ii) of S and O). Case (b): Neither L_j^l nor L_j^h is empty and $L_j \neq L_i$, and now $\mathscr{C}_i$ is modeled in levels of the circle inside the levels used by $\mathscr{C}_j$; the possibility of $\max L_j^l = \min L_i$ and/or $\min L_j^h = \max L_i$ is treated as in Case (a). Case (c): $L_i = L_j = \{k_1, k_2\}$, $k_1 < k_2$ (and so also $\mathscr{C}_j \leqq \mathscr{C}_i$) and now to assure that arcs in different components are unrelated, in this case, it means each pair of arcs from different components must overlap, one component should use the upper part of level k_1 and the lower part of level k_2—the parts of these two levels closer to each other—and the other component use the farther parts of levels k_1 and k_2. (If there are several $L_i = L_j = L_k = \cdots = \{k_1, k_2\}$, we order them all from closer to farther parts of levels k_1 and k_2.) In each case, we can change the left-right orientation of one component without affecting overlaps and nonoverlaps with other components. It is easy to check that any collection of components in which every pair is related by $\leqq$ can be modeled on the circle with each component's model in a zone of the circle above, below, or within, the zones of each other component's models in the manner described in the above three cases. Then, to prove this proposition, it suffices to show that the set of all components is such a collection, i.e., for every pair of components $\mathscr{C}_i$, $\mathscr{C}_j$, either $\mathscr{C}_i \leqq \mathscr{C}_j$ or $\mathscr{C}_j \leqq \mathscr{C}_i$ (or both).

Suppose components $\mathscr{C}_1$ and $\mathscr{C}_2$ are unrelated in $\leqq$. Then it is easy to check that there must exist k_1^l, $k_1^h \in L_1$ and k_2^l, $k_2^h \in L_2$ with $k_1^l < k_2^l < k_1^h < k_2^h$, or equivalently with subscripts interchanged (or else $L_1 = L_2 = \{k_1, k_2, k_3\}$—we shall omit the proof that in this special 3-level case $\mathscr{C}_1$ and $\mathscr{C}_2$ must be equal). Let A_1 represent a possible arc in $\mathscr{C}_1$ with $l(A_1) = k_1^l$, $h(A_1) = k_1^h$ and A_2 represent a possible arc in $\mathscr{C}_2$ with $l(A_2) = k_2^l$, $h(A_2) = k_2^h$. Then we divide our analysis into three cases: Case (a)—A_1, A_2 both exist; Case (b)—exactly one of A_1, A_2 exist; and Case (c)—neither exist. First we need the following very useful facts. For $B \in \mathscr{B}$, the definition of $h(B)$ implies that there exists a $D' \in \mathscr{D}$ with $h(D') = h(B)$ and $B \not\cap D'$. We call such a D' an *higher barrier* of B. Similarly for $D \in \mathscr{D}$, there exists a $B' \in \mathscr{B}$ which we call a *lower barrier* of D with $l(B') = l(D)$ and $B' \not\cap D$. Clearly any higher (lower) barrier of an arc A is in the same component as A (by condition (ii) of S).

Case (a)—A_1, A_2 both exist. We can assume A_1 is a $\mathscr{D}$ arc; if not, then replace it by one of its higher barriers D_1 with $h(D_1) = k_1^h$ and $l(D_1) = k_1^{h^*} \leqq k_1^h$ and set $k_1^h = k_1^{h^*}$.

Similarly, we can assume A_2 is a $\mathscr{B}$ arc. Now $O(A_1, A_2)$ or $S(A_1, A_2)$ depending on whether or not $A_1 \cap A_2$ by conditions (i) or (ii), but A_1 and A_2 are supposed to be in different components.

Case (b)—Exactly one of A_1, A_2 exist. Assume A_1 or A_2 is chosen so as to maximize the spread $k_1^h - k_1^l$ or $k_2^h - k_2^l$. Suppose A_1 is the one (A_2 is a symmetrically equivalent situation). Then there exist arcs A_2^l, A_2^h in $\mathscr{C}_2$ with one of A_2^l's endpoints at level k_2^l and one of A_2^h's endpoints at level k_2^h and a minimal sequence of arcs in $\mathscr{C}_2, A_2^1, A_2^2, A_2^3, \cdots, A_2^m$, where $A_2^l = A_2^1, A_2^h = A_2^m$, and every consecutive pair A_2^i, A_2^{i+1} is related by S or O (but no other pairs A_2^i, A_2^j are related). Further we can assume that no A_2^i, except possibly A_2^h, has an endpoint below level k_1^l (or else we could treat A_1, A_2^i, A_2^1 as a "shorter" version of a case (b) situation). Assuming A_2^l and A_2^h were chosen to minimize the length of the sequence of A_2^i's, we see that no A_2^i except A_2^1 $(=A_2^l)$ can have an endpoint between k_1^l and k_1^h and no A_2^i except A_2^m $(=A_2^h)$ can have an endpoint higher than k_1^h. It follows that for all $A_2^i, 2 \leqq i \leqq m-1, l(A_2^i) = k_1^l$ and $h(A_2^i) = k_1^h$. Also the other endpoint of A_2^l cannot be above k_1^h and the other endpoint of A_2^h cannot be in between k_1^l and k_1^h. Now the only way A_2^1 can be related to A_2^2 by S or O is to have a common end level (k_1^l or k_1^h) with A_2^2; similarly, $l(A_2^h) = k_1^l$ or k_1^h. Recall that we assume $k_1^h - k_1^l$ is as large as possible. If $A_1 = B_1$ is a B-arc, its upper barrier D_1' ($h(D_1') = k_1^h$ and $B_1 \not\cap D_1'$) must have $l(D_1') = k_1^l$ or else A_1 could be D_1' with a larger value of $k_1^h l(D_1')$ (the same sequence of A_2^i's or a subsequence of it works for D_1'). Similarly if $A_1 = D_1$ is a D-arc, we claim that its lower barrier B_1' ($l(B_1') = k_1^l$ and $B_1' \not\cap D_1$) must have $h(B') = k_1^h$. If $k_1^h < h(B_1') < k_2^h$, then A_1 could be B_1' with a larger value of $h(B_1') - k_1^l$. If either $k_2^h < h(B_j)$ and $l(A_2^h) = k_1^l$ or $k_2^h = h(B_1')$ and $l(A_2^h) < k_1^l$, then A_2^h or B_1' could replace A_1 with a larger spread. If $k_2^h \leqq h(B_1')$ and $l(A_2^h) = k_1^h$, then A_2^{m-1} (with $h(A_2^{m-1}) = k_1^h$) must overlap $A_2^h = (A_2^m)$ and $A_2^h = B_2^h$ is a $\mathscr{B}$-arc and $A_2^{m-1} = D_2^{m-1}$ must be a $\mathscr{D}$-arc—this is the only way two such arcs can be related; and now $D_1, B_1', B_2^h, D_2^{m-1}$, and B^*, the lower barrier of D_2^{m-1}, satisfy condition (vi) relating D_1 and D_2^{m-1} (or if $h(B^*) > k_1^h, D_1, B_1', D_2^{m-1}, B_1^*$ satisfy (iv)). Finally suppose $k_2^h = h(B_1')$ and $l(A_2^h) = k_1^l$. By the maximality of $k_1^h - k_1^l$ and in order to avoid previously considered situations we can assume that $A_2^h = B_2^h$ is a $\mathscr{B}$-arc, $A_2^{m-1} = D_2^{m-1}$ is a $\mathscr{D}$-arc, and D_2^{m-1} is related to B_2^h $(=A_2^m)$ by condition (ii) so that $B_2^h \cap D_2^{m-1}$ (the other conditions for relating A_2^{m-1} and A_2^m require both end sections of the two arcs to be the same levels or that there exist other arcs in $\mathscr{C}_2$ with endpoints above k_2^h or below k_1^l—a situation causing B_1' to replace A_1 with larger spread). Now condition (iv) applies to $D_1, B_1', D_2^{m-1}, B_2^h$. This proves our claim that $h(B_1') = k_1^h$. So A_1 and its (lower or higher) barrier A_1^0 have the same end sections. By arguments similar to those just employed, it is not hard to show that the A_2^i can be assumed to be related by conditions (i) or (ii). Now A_1, A_1^0 and the sequence $A_2^1, A_2^2, \cdots, A_2^m$ satisfy condition (vii).

Case (c)—Neither A_1 nor A_2 exist. Let A_1^l and A_1^h have endpoints at levels k_1^l, k_2^h, respectively; define A_2^l and A_2^h similarly. Now to avoid Case (b), the other end of A_1^l must not be in between k_2^l and k_2^h. Assume first there is no arc in $\mathscr{C}_1$ from below k_2^l to above k_2^h. Then to link A_1^l with A_1^h in $\mathscr{C}_1$ there must be D_1^* and B_1^* in $\mathscr{C}_1$ with $h(D_1^*) = l(B_1^*) = k_2^l$ and $D_1^* \cap B_1^*$. Clearly we can let $A_1^l = D_1^*$. Next we observe that the other endpoint of A_2^l, besides the one in k_2^l, must be in k_1^l or k_1^h (any other level makes A_2^l or A_1^l a Case (b) situation). We claim that $l(A_2^l) = k_1^l$. There cannot exist A_2^* and B_2^* in $\mathscr{C}_2$ with $h(D_2^*) = l(B_2^*) = k_1^h$ and $D_2^* \cap B_2^*$ by condition (v). If $h(A_2^l) = k_1^h$, then to avoid the preceding condition (v) or a Case (b) situation, the next arc A_2^0 in a sequence of related arcs from A_2^l to A_2^h has $h(A_2^0) = k_1^h$ and $l(A_2^0) = k_2^l$ or $\leqq k_1^l$, and we go back and forth between k_1^h and k_2^l until some arc A' goes from level k_1^h to or below level k_1^l. We can assume $A' = A_2^0$ and the preceding arc in the sequence was A_2^l, and such A_0^2 and A_2^l can only be related in this setting if $A_2^0 = D_2^0$ and $A_2^l = B_2^l$. But now we must have $D_2^0 \cap B_1^*$ (or they are related by condition (ii)) and so $h(B_1^*) = k_1^h$ (if $<k_1^h$, A_2^l is a Case (b) situation). Now D_2^0, B_2^l, D_1^*, B_1^* and D' the barrier of B_1^* ($l(D_1) = k_1^h$ or condition (iv) will apply) satisfy condition (vi).

This proves our claim that $l\ (A_2^l) = k_1^l$. Further we can assume $A_2^l = D_2^l$, or if a $\mathscr{B}$-arc we take its barrier which, like A_2^l, must have lower endpoint in k_1^l. Then D_2^l must be linked to A_2^h in $\mathscr{C}_2$ by a $\mathscr{B}$-arc B^0 with $l(B^0) = k_1^l$, $h(B^0) > k_1^h$, and $B^0 \not\cap D_2^l$ ($h(B^0) = k_2^l$ is the only other possibility not leading to Case (b)—but this is "going around in a circle" back to k_2^l, which will have to be followed by a A_2^l-type arc from k_2^l to k_1^l or k_1^h). Now we get the same problem as when $h(A_2^l) = k_1^h$; condition (iv) or (vi) will apply to B^0, D_2^l, D_1^*, B_1^* and B', the lower barrier of D_1^*. This completes Case (c) when there is no arc in $\mathscr{C}_1$ from below k_2^l to above k_2^h. By symmetry, we also rule out that there is no arc in $\mathscr{C}_2$ from above k_1^h to below k_1^l.

Now we can assume A_1^l goes from k_1^l to $\geqq k_2^h$ and A_2^h goes from k_2^h to $\leqq k_1^l$. To avoid Case (a) or (b) situations, we are forced to have $l(A_1^l) = l(A_2^h) = k_1^l$ and $h(A_1^l) = h(A_2^h) = k_2^h$. Moreover, the (lower or higher) barriers of A_1^l and A_2^h must also go between k_1^l and k_2^h. So we have B_i', $D_i' \in \mathscr{C}_i$, $B_i' \not\cap D_i'$, $i = 1, 2$, going from k_1^l to k_2^h. Now for B_1', D_1' to be connected in $\mathscr{C}_1$ with A_1^h with endpoint in k_1^h, we must have $A_1^h = B_1^h$ with $l(B_1^h) = k_1^h$, $h(B_1^h) = k_2^h$ and $B_1^h \not\cap D_1'$ or $A_1^h = D_1^h$ with $h(D_1^h) = k_1^h$, $l(D_1^h) = k_1^l$ and $D_1^h \not\cap D_1'$. A similar situation holds for A_2^l, and now either Case (a), condition (iii), or condition (iv) will hold for all or a subset of these six arcs. This completes Case (c).

Thus $\mathscr{C}_1$, $\mathscr{C}_2$ must be equal and the proposition is proved. Q.E.D.

PROPOSITION 3. *Any circular-arc model for G_j (defined in Stage Two, Subcase* IIa) *can be expanded to a circular-arc model for G (if one exists) with the only possible change in the order of endpoints in S_{2j-1}, S_{2j}, S_{2j+1} in G_j's model being for pairs of arcs which overlap the same subset of $\mathscr{P}_j^0$.*

Proof. The only missing arc endpoints in S_{2j-1}, S_{2j}, S_{2j+1} (for arcs in G but not G_j) are: (a) arcs A with Cc(A) in S_{2j+1} or Cl(A) in S_{2j-1} and A touches none of $\mathscr{P}_j'$ (these arcs clearly cannot affect the order of endpoints of G_j's arcs); (b) arcs A with Cc(A) or Cl(A) in S_{2j} and A touches all of $\mathscr{P}_j'$ (since $\mathscr{N}(A_j^*) \not\subset \mathscr{N}(A)$ but $\mathscr{P}_j' \subset \mathscr{N}(A)$, there must be another Case (b) arc A' with Cl(A') or Cc(A') in S_{2j} and $A \not\cap A'$; but all such case (b) arcs must have their endpoint in S_{2j} within the region of common overlap of arcs of $\mathscr{P}_j'$, where no endpoints of G_j arcs occur, and so these endpoints cannot affect the order of the endpoints of arcs in G_j); (c) arcs A with Cc(A) in S_{2j-1} or Cl(A) in S_{2j+1} and A touches all of $\mathscr{P}_j'$. The Case (c) arcs require a little discussion. Suppose there are two arcs A_1, A_2 in G_j with Cl(A_1) and Cl(A_2) in S_{2j-1} and an arc A in $G - G_j$ with Cc(A) in S_{2j-1} which overlaps A_1 but not A_2 (this implies $A_2 \in \mathscr{Q}_j$). If A_1, A_2 overlap the same subset of $\mathscr{P}_j'$, then the relevant order of Cl(A_1) and Cl(A_2) was arbitrary in the model of G_j and may be altered, as mentioned above, to allow only A_1 to overlap A. If $\mathscr{N}(A_2) \cap \mathscr{P}_j' \subset \mathscr{N}(A_1) \cap \mathscr{P}_j'$, then Cl($A_1$), Cl($A_2$) are already forced to be ordered so A can overlap A_1 but not A_2. ($\mathscr{N}(A_1) \cap \mathscr{P}_j' \subset \mathscr{N}(A_2) \cap \mathscr{P}_j'$ would be impossible in a circular-arc model for G). A similar analysis applies if Cc(A_1), Cc(A_2) $\in S_{2j+1}$. Q.E.D.

PROPOSITION 4. *If G is a circular-arc graph, then any pair of arcs A, A' in $\mathscr{B} \cup \mathscr{D}$ (as defined in Stage Three of the algorithm) with their other endpoints in different sections are comparable in the partial order $\overset{*}{\leqq}$, i.e. $A \overset{*}{\leqq} A'$ or $A' \overset{*}{\leqq} A$.*

Proof. First consider $B_1, B_2 \in \mathscr{B}$ (i.e., Cl(B_1), Cl(B_2) $\in S_j$) with B_2 extending (counterclockwise) to a farther endsection than B_1. If $\mathscr{N}(B_1) \subset \mathscr{N}(B_2)$, then $B_1 \leqq B_2$. Otherwise, there exists $D_1 \in \mathscr{N}(B_1) - \mathscr{N}(B_2)$ which B_1 can only overlap in S_j, in which case $B_2 \leqq D_1 \leqq B_1$. By a similar argument we have $D_1 \overset{*}{\leqq} D_2$ or $D_2 \overset{*}{\leqq} D_1$ for any two arcs in $\mathscr{D}$ with different other endsections. Next consider $B \in \mathscr{B}$, $D \in \mathscr{D}$ with different other endsections. If $B \not\cap D$, then $B \leqq D$. If $B \cap D$ but B and D cannot overlap at their other ends, then $D \leqq B$. Suppose finally $B \cap D$ and B, D must overlap at their other ends. If there exist $B' \in \mathscr{B} - \mathcal{N}(D)$ and $D' \in \mathscr{D} - \mathcal{N}(D)$ with $B' \cap D'$, then we have $B \leqq D' \leqq B' \leqq D$ ($B' \not\cap D$ and $D' \not\cap B$ implies B' and D' cannot touch at their other sides). If no such B', D' pair exists, then for all $D' \in \mathscr{D} - \mathcal{N}(B)$, $\mathcal{N}(D') \subset \mathcal{N}(D)$ and so $D \leqq B$. Q.E.D.

REFERENCES

[1] S. BENZER, *On the topology of the genetic fine structure*, Proc. Nat. Acad. Sci., 45 (1959), pp. 1607–1620.

[2] K. S. BOOTH, *PQ-tree algorithms*, Ph.D. thesis, Computer Science Department, University of California, Berkeley, 1975.

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

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

[5] F. GAVRIL, *Algorithms on circular-arc graphs*, Networks, 4 (1974), pp. 357–369.

[6] P. C. GILMORE AND A. J. HOFFMAN, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.

[7] L. HUBERT, *Some applications of graph theory and related non-metric techniques to problems of approximate seriation: the case of symmetry proximity measures*, British J. Math. Statist. Pscychology, 27 (1974), pp. 133–153.

[8] C. B. LEKKERKERKER AND J. C. BOLAND, *Representation of a finite graph by a set of intervals on the real line*, Fund. Math., 51 (1962), pp. 45–64.

[9] G. S. LUEKER, *Efficient algorithms for chordal graphs and interval graphs*, Ph.D. thesis, Program in Applied Mathematics and Electrical Engineering, Princeton University, 1975.

[10] F. S. ROBERTS, *Discrete Mathematical Models*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] F. W. STAHL, *Circular genetic maps*, J. Cell Physiology, 70 (Suppl. 1) (1967), pp. 1–12.

[12] K. E. STOUFFERS, *Scheduling of traffic lights—a new approach*, Transportation Res., 2 (1968), pp. 199–234.

[13] W. TROTTER AND J. MOORE, *Characterization problems for graphs, partially ordered sets, lattices, and families of sets*, Discrete Math., 16 (1976), pp. 361–381.

[14] A. TUCKER, *Matrix characterizations of circular-arc graphs*, Pacific J. Math., 39 (1971), pp. 535–545.

[15] ———, *Structure theorems for some circular-arc graphs*, Discrete Math., 7 (1974), pp. 167–195.

[16] ———, *Coloring a family of circular-arc graphs*, SIAM J. Appl. Math., 29 (1975), pp. 493–502.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

SOLUTIONS OF THE ITERATION EQUATION AND EXTENSIONS OF THE SCALAR ITERATION OPERATION*

STEPHEN L. BLOOM,† CALVIN C. ELGOT‡ AND JESSE B. WRIGHT‡

Abstract. We study the solutions to a (vector) equation somewhat analogous to the traditional equations of linear algebra. Whereas, in introductory linear algebra the domain of discourse is the field of real numbers (or an arbitrary field) our domain of discourse is the algebraic theory of (multi-rooted, leaf-labeled) trees (or, more generally, any iterative theory).

As in linear algebra, we obtain a necessary and sufficient condition for our equations to have unique solutions and we can describe "parametrically" the totality of solutions. However, whereas in linear algebra, there is no way of giving $1 \div 0$ meaning in such a way that all the "old laws" hold, we can give meaning to the "iteration operation" (the analogue of division into 1) in such a way that all the "old laws" still hold. Indeed, we can describe "parametrically" all such ways of extending the (partially defined) scalar iteration operation to all trees (more generally, morphisms).

Key words. iteration, semantics, flowchart schemes, iterative theory

1. Introduction.

1.1. An example. The notion of iterative theory (reviewed in this section) is discussed in [5], [3], [11], [4] and [7] the principal reference for this paper. Its relationship to the theory of computation is indicated in [5]. An important example of an iterative theory is provided by multi-rooted, (locally) ordered trees (not necessarily finite) with *termini* (a distinguished subset of the set of all leaves of a tree) which are "labeled" by positive integers. An n-rooted (vector) tree f (or "forest" of n singly rooted trees) of this kind whose termini labels come from the set $\{1, 2, 3, \cdots, p\}$—abbreviated $[p]$—is indicated as follows

$$f\colon n \to p.$$

When $p = 0$, there are no termini, i.e., all the leaves (if any) are unlabeled. If $g\colon p \to q$ is another such tree, by $f \cdot g\colon n \to q$, we mean the tree which is obtained by "attaching", to each terminus of f labeled j, for each $j \in [p]$, a "copy" of the jth singly rooted tree of g; $f \cdot g$ is the *composition* of f and g. (More strictly speaking, this operation is on isomorphism classes of trees rather than on individual trees.)

The tree $I_p \oplus 0_n\colon p \to p + n$, $n \geqq 0$, consists of a sequence of p vertices; the jth vertex is both the jth root and a terminus labeled j, for each $j \in [p]$. Notice that $f \cdot I_p = f$ and $I_p \cdot g = g$.

Another basic operation, besides composition, is "source pairing". If $f_i\colon n_i \to p$, $i \in [2]$, are forests, the forest $(f_1, f_2)\colon n_1 + n_2 \to p$ has f_1 as its first n_1 singly rooted trees and f_2 as its next n_2 singly rooted trees. This operation (again, strictly speaking on isomorphism classes), as well as composition, is associative. If $r > 2$, we write

$$(f_1, f_2, \cdots, f_r)$$

for the extension of the binary operation to an r-ary operation.

The very familiar operation of constructing a new singly rooted (or scalar) tree from n single rooted trees by adjoining a new vertex may be described as

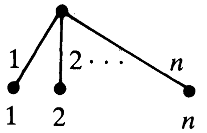
$$\gamma_n \cdot (f_1, f_2, \cdots, f_n)$$

* Received by the editors May 23, 1978.

† Department of Mathematics, Stevens Institute of Technology, Hoboken, New Jersey 07030.

‡ Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

where f_i is a scalar tree, for each $i \in [n]$ and $\gamma_n\colon 1 \to n$ is the tree depicted by



The trees γ_n, $n = 0, 1, 2, 3, \cdots$, are called *atomic.* A *primitive tree* is one such that every successor of the root is a terminus. A vector tree is *primitive* if each of its component scalar trees is primitive.

The collection of (isomorphism classes of) of trees $n \to p$ for variable n, p form a category Tr whose *objects* are $0, 1, 2, \cdots$ and whose *morphisms* $n \to p$ are trees $n \to p$. Each object n in Tr is a coproduct of the object 1 with itself n times and thus forms an *algebraic theory* in the sense of Lawvere (cf. [13]). This algebraic theory is *ideal* (cf. [5]) in that if $f\colon 1 \to n$ is a tree which is *nontrivial* (in the sense that its root is not a terminus) then so is $f \cdot g$ for every $g\colon n \to p$ in Tr.

The ideal algebraic theory Tr is *closed with respect to conditional iteration*, briefly, is *iterative*, in that the equation in Tr

$$\xi = g \cdot (I_p, \xi),$$

where $\xi\colon n \to p$ and where $g\colon n \to p+n$ is *ideal* (i.e. is a forest of n nontrivial singly rooted trees), has a unique solution for the "unknown" ξ. This "vector" equation is, in a sense we will not make clear here, linear and may be "rewritten" as n (linear) equations in n scalar "unknowns" $\xi_i\colon 1 \to p$, $i \in [n]$. This equation is the *iteration equation determined by g.*

In the case $p = 0$, the iteration equation for g takes the simpler form:

$$\xi = g \cdot \xi.$$

The collection of trees of "finite index" (i.e. trees having only a finite number of nonisomorphic subtrees, cf. [7]) forms an iterative subtheory tr of Tr.

One reason for the importance of the iteration equation is this: for each (possibly infinite) tree $f\colon 1 \to p$ in tr there is a primitive $g\colon n \to p+n$ in *tr* such that f is the first component of the solution to the iteration equation for g. Another reason is that in models of computation, the solution to the iteration equation expresses "looping".

1.2. Solutions. While the solution (in a given iterative theory) to the iteration equation for $g\colon n \to p+n$ is unique in the case that g is ideal, in general, there are many solutions. For example, in the case that $p = 0$ and $g = I_n$, each morphism $f\colon n \to n$ satisfies the iteration equation for g. We give, in 2 (cf. 2.15), a kind of parametric description of all solutions to the iteration equation for g, where $g\colon n \to p+n$ is an arbitrary morphism. We obtain as a corollary (cf. 2.19) a necessary and sufficient condition on g for its iteration equation to have a unique solution. This corollary is equivalent to Theorem B of J. Tiuryn [8].

When applied to Tr (cf. 2.21) the corollary states that g's iteration equation has a unique solution iff the tree

$$g^n = g \cdot (I_p \oplus 0_n, g) \cdot (I_0 \oplus 0_n, g) \cdot (I_p \oplus 0_n, g) \cdots, \quad (n \text{ occurrences of ``}g\text{''})$$

has the property that each trivial component of g^n carries a label from the set $[p]$ (out of the set $[p+n]$ of possible labels). In the case $p = 0$,

$$g^n = g \cdot g \cdot g \cdots \quad (n \text{ occurrences of ``}g\text{''})$$

$[p]$ is the empty set and the property of g becomes simply:

$$\text{each component of } g^n \text{ is nontrivial.}$$

1.3. Extension. In iterative theories we may define $g^\dagger$ (the *iterate of* g) to be the unique solution of g's iteration equation, if g's iteration equation has a unique solution; in the contrary case we may regard "$g^\dagger$" as meaningless. It is natural then to raise the question: it is possible to extend the meaning of $g^\dagger$ to *all* g in such a way that all the old "laws", i.e., "identities" still hold. The answer to the question is "yes". This matter is taken up in § 3 (cf. 3.7 and 3.8) for the case that g is scalar, i.e., g has source 1; the vector case is reserved for a sequel to this paper (in preparation).

The "full answer" to the question is rather neat and perhaps surprising. It is this: if one chooses $\perp : 1 \to 0$ in the iterative theory arbitrarily[1] and defines $I_1^\dagger = \perp$, then the requirement that the old laws still persist, uniquely determines $g^\dagger$ for all g. Thus, in Tr, we may choose $\perp : 1 \to 0$ to be any infinite tree (without termini) and we may define the iterate of the trivial tree $I_1 : 1 \to 1$ to be $\perp$ without violating any laws!

There is another sense in which the result is surprising, viz: while the results mentioned in § 1.2 are reminiscent of and roughly analogous to results for linear equations over a field, and while the question raised also has an analogue in the field domain (replace "$g^\dagger$" by "$1 \div g$"), the answer in the field domain is "no".

1.4. The sequel. The sequel [1] to this paper (to which we've already alluded) centers around a formula from [4] which is an iterative theory identity. This formula expresses the iterate of an $(n+1)$ dimensional vector morphism in terms of the iterates of n-dimensional and 1-dimensional morphisms. The formula may be used to define the vector iteration operation (on all morphisms) in terms of the scalar iteration operation. It turns out that *all* the old laws valid in iterative theories are still valid.

The morphism $g^\dagger$, for the arbitrary vector morphism g in Tr (and other "tree theories"), may be expressed as a metric limit, no matter how $\perp : 1 \to 0$ is selected. (The trees $n \to p$ in Tr form a complete metric space as was noted independently in [9].) This contrasts with the fact that it is not always possible to define a partial ordering (compatible with composition) in Tr such that $g^\dagger$ is the least (or greatest) solution of the iteration equation for g. Whether or not one can define such an ordering depends on the choice of $\perp$. This fact is of interest in connection with the many mathematical treatments of the semantics of flowchart schemes that make fundamental use of partial orderings, (e.g. [10], [12], [2]).

1.5. Elementary properties of algebraic theories. An *algebraic theory* T is a category whose objects are the nonnegative integers, having for each $n > 0$, n "distinguished morphisms" $\mathbf{i}: 1 \to n$, $i \in [n]$[2] (where $[n] = \{1, 2, \cdots, n\}$; $[0] = \varnothing$) such that: for each family of morphisms $f_i : 1 \to p$, $i \in [n]$, $n \geqq 0$, there is a unique morphism $f: n \to p$ such that for each $i \in [n]$, f_i is the composition

$$f_i : 1 \xrightarrow{i} n \xrightarrow{f} p.$$

The morphism f is called the *source tupling* of the morphisms f_i, $i \in [n]$, and is denoted $(f_1, f_2, \cdots, f_n)$. In the case $n = 0$, this condition requires the existence of a unique

[1] Every iterative theory except $\mathcal{N}$ contains at least one morphism $1 \to 0$. The theory $\mathcal{N}$ may be described as the subtheory of Tr which consists of all the trivial trees in Tr; it may also be described as the theory whose morphisms $n \to p$ are the functions $[n] \to [p]$.

[2] We admit this notation is ambiguous: **2** may have target 2 or any number larger than 2, but in context the target should be clear.

morphism 0_p: $0 \to p$. All morphisms $n \to p$, n, $p \geqq 0$, formed by source tupling the distinguished morphisms are called *base* morphisms. When the distinguished morphisms are distinct, (which is the case in every algebraic theory but two: the "trivial theories") the base morphisms are in 1–1 correspondence with the collection of functions $[n] \to [p]$. The function f: $[n] \to [p]$ corresponds to the source tupling ($\mathbf{f}(\mathbf{1})$, $\mathbf{f}(\mathbf{2}), \cdots, \mathbf{f}(\mathbf{n})$), where $\mathbf{f}(\mathbf{i})$: $1 \to p$ is distinguished.

If "f" denotes a function $[n] \to [p]$, then "f: $n \to p$" will also denote the morphism in T corresponding to f. The base morphism corresponding to the identity function on $[n]$ is denoted I_n.

The *composition* of f: $n \to p$ and g: $p \to q$ is denoted either $f \cdot g$ or $n \xrightarrow{f} p \xrightarrow{g} q$.

It is convenient to define an operation, derived from source tupling, which pairs two morphisms with arbitrary sources. If f_i: $n_i \to p$, $i = 1, 2$, then the *source pairing* (f_1, f_2): $n_1 + n_2 \to p$ is the unique morphism satisfying

$$\mathbf{i} \cdot (f_1, f_2) = \begin{cases} \mathbf{i} \cdot f_1, & \text{if } i \in [n_1], \\ \mathbf{j} \cdot f_2, & \text{if } i = n_1 + j,\ j \in [n_2]. \end{cases}$$

Let κ: $[p_1] \to [p_1 + p_2]$, λ: $[p_2] \to [p_1 + p_2]$ be the inclusion and translated inclusion functions (i.e., $i\kappa = i$, $i\lambda = p + i$). If f_i: $n_i \to p_i$, $1 = 1, 2$, then we define the "circle plus" of f_1 and f_2 as follows: $f_1 \oplus f_2$: $n_1 + n_2 \to p_1 + p_2$ is the morphism $(f_1 \cdot \kappa, f_2 \cdot \lambda)$.

Whenever the expressions below are meaningful, i.e., for the appropriate sources and targets, the following assertions hold in any algebraic theory (see [5]).

(1.5.1) $$(f_1 \oplus f_2) \cdot (g_1 \oplus g_2) = f_1 \cdot g_1 \quad \oplus \quad f_2 \cdot g_2,$$

(1.5.2) $$(f_1 \oplus f_2) \cdot (g_1, g_2) = (f_1 \cdot g_1, f_2 \cdot g_2),$$

(1.5.3) $$(0_p, g) = g = (g, 0_p),$$

(1.5.4) $$(f_1, g_2) \cdot g = (f_1 \cdot g, f_2 \cdot g).$$

The algebraic theory T is *ideal* if for each nondistinguished morphism g: $1 \to n$, $g \cdot f$ is nondistinguished, for any f: $n \to p$. A nondistinguished morphism g: $1 \to n$ in T is called ideal.

An *iterative theory* is a nontrivial ideal theory such that for each ideal morphism g: $1 \to p + 1$ there is a unique morphism $g^\dagger$: $1 \to p$ such that

(*) $$g^\dagger = g \cdot (I_p, g^\dagger).$$

In an iterative theory, if g: $n \to p + n$, $n > 1$ is ideal, i.e., for each i $\in [n]$, $\mathbf{i} \cdot g$ is ideal, it can be shown there is a unique morphism $g^\dagger$: $n \to p$ such that (*) holds [4].

If g: $n \to p + n$ is any morphism in an algebraic theory, the "powers" of g are defined as follows:

$$g^0 = 0_p \oplus I_n$$

and

$$g^{r+1} = g^r \cdot (I_p \oplus 0_n, g): n \to p + n.$$

The following facts about g^r are quite useful.

(1.5.5) $$(I_p \oplus 0_n, g)^r = (I_p \oplus 0_n, g^r), \quad \text{all } r \geqq 0;$$

(1.5.6) $$\text{if } \xi = g \cdot (I_p, \xi), \quad \text{then } \xi = g^r \cdot (I_p, \xi), \quad \text{all } r \geqq 0.$$

1.6. Γtr. In §§ 3 and 4, Γ will denote a ranked set; i.e., Γ is the disjoint union $\bigcup(\Gamma_k\colon k \geqq 0)$.

In [7] it was shown that Γtr, the collection of Γ-trees of "finite index", formed an iterative theory which is freely generated by Γ. This means that for any iterative theory J and any function F that maps $\gamma \in \Gamma_k$ into an *ideal* morphism $\bar{\gamma}\colon 1 \to k$ in J, there is a unique theory morphism $\mathbf{F}\colon \Gamma\mathrm{tr} \to J$ extending F. This result is generalized by the Universality Theorem in § 3.

2. All solutions of the iteration equation. The determination of all solutions of the iteration equation depends upon classifying the component positions of a morphism $g\colon n \to p+n$ into three disjoint categories. A position $i \in [n]$ of g is either "singular", "power successful" or "power ideal" (see § 2.3). Singular positions are responsible for the existence of more than one solution of the iteration equation. On the other hand, if all the component positions are either power successful or power ideal, there is a unique solution of the iteration equation for g.

2.1. Let $g\colon n \to p+n$ be an arbitrary morphism in an iterative theory I. The *iteration equation for* g is the equation in the "variable" $\xi\colon n \to p$

$$\xi = g \cdot (I_p, \xi). \tag{2.1.1}$$

By definition of "iterative theory" (more fully, "ideal theory closed under conditional iteration") whenever g is an ideal morphism, the equation (2.1.1) has a unique solution, i.e., there is a unique morphism $\xi\colon n \to p$ which satisfies (2.1.1). This solution is denoted $g^\dagger$. If g is not ideal, the iteration equation for g may have many solutions. In this section, solutions to (2.1.1) are determined.

In the case $n > 1$ it is useful to rewrite (2.1.1) as a system of simultaneous equations. For $i \in [n]$, let $\xi_i = \mathbf{i} \cdot \xi = 1 \xrightarrow{\mathbf{i}} n \xrightarrow{\xi} p$ and let $g_i = \mathbf{i} \cdot g\colon 1 \xrightarrow{\mathbf{i}} n \xrightarrow{g} p+n$. Then (2.1.1) is equivalent to the system

$$\begin{aligned} \xi_1 &= g_1 \cdot (I_p, \xi_1, \xi_2, \cdots, \xi_n),\\ \xi_2 &= g_2 \cdot (I_p, \xi_1, \xi_2, \cdots, \xi_n),\\ &\vdots\\ \xi_n &= g_n \cdot (I_p, \xi_1, \xi_2, \cdots, \xi_n). \end{aligned} \tag{2.1.2}$$

When g is not ideal, there is at least one $i \in [n]$ for which g_i is a base morphism $1 \to p+n$. If g_i is $0_p \oplus \mathbf{i}'$ for some $i' \in [n]$, then the ith equation (2.1.2) is a "pure variable equation":

$$\xi_i = \xi_{i'}. \tag{2.1.3}$$

Similarly, if g_i is $\mathbf{j} \oplus 0_n$, for some $j \in [p]$, then the ith equation in (2.1.2) determines the value of ξ_i:

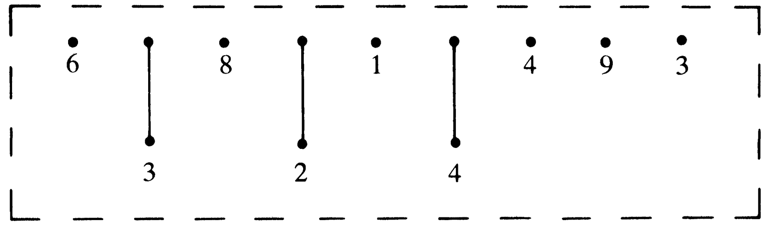
$$\xi_i = \mathbf{j}\colon 1 \to p. \tag{2.1.4}$$

To illustrate these possibilities, we will discuss an example in some detail.

2.2. Example. Throughout § 2 we will discuss the solution of the iteration equation for a morphism G in the iterative theory of Γ-trees (see [7]). We recall that a tree $f\colon n \to p$ is an n-tuple $(f_1, \cdots, f_n)$ of rooted (locally-ordered, locally-finite) trees, some of whose leaves are labeled with elements of $[p]$. To compose $f\colon n \to p$ with $g\colon p \to q$ one attaches to each leaf of f labeled $i \in [p]$ a copy of the ith component of g. In our example, we assume all vertices of G of outdegree one have the same (unindicated) label in Γ_1.

Let $G\colon 9 \to 10$ be the tree

(2.2.1) $G =$



Then the system of equations corresponding to the iteration equation for G (the solution is depicted by (2.6.2)) is

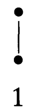
(2.2.2)

$$\begin{aligned}
\xi_1 &= \xi_5 && (1)\\
\xi_2 &= G_2' \cdot \xi_2 && (2)\\
\xi_3 &= \xi_7 && (3)\\
\xi_4 &= G_4' \cdot \xi_1 && (4)\\
\xi_5 &= \mathbf{1}\colon 1 \to 1 && (5)\\
\xi_6 &= G_6' \cdot \xi_3 && (6)\\
\xi_7 &= \xi_3 && (7)\\
\xi_8 &= \xi_8 && (8)\\
\xi_9 &= \xi_2. && (9)
\end{aligned}$$

Here, for $i = 2, 4, 6$

$$G_i'\colon 1 \xrightarrow{G_i} 10 \xrightarrow{\text{base morphism}} 1,$$

i.e. $G_2' (= G_4' = G_6')$ is the tree



so that $G_i = G_i' \cdot \mathbf{j}$, for some unique $j \in [10]$.

Although these equations are sufficiently clear to allow one to determine all solutions of the iteration for G, before doing so we note that the equations fall into three distinct groups. Equations (1) and (5) of (2.2.2) may be grouped together, since they determine the value of ξ_1 and ξ_5 as the base morphism $\mathbf{1}\colon 1 \to 1$. The "pure variable" equations (3), (7), and (8) may be grouped together, since they determine only that $\xi_3 = \xi_7$; ξ_3 $(=\xi_7)$ and ξ_8 may be any trees $1 \to 1$. The last group is the remaining equations (2), (4), (6), and (9). In § 2.4 we will rearrange the equations so that these groups occur together. The classification of these three groups of equations is discussed in general in the next section.

2.3. The initial and final classification. Let $g\colon n \to p + n$ be a morphism in an iterative theory. The *initial classification* of the component positions $i \in [n]$ of g is the following.

2.3.1. The position i is a *successful* position (for g) if for some (unique) $j \in [p]$, $\mathbf{i} \cdot g = \mathbf{j}\colon 1 \to p + n$. We define the function $g^\varepsilon\colon S_g \to [p]$, where S_g is the set of successful positions of g, by $ig^{\varepsilon} = j$. Thus if i is a successful position for g, the ith equation in (2.1.2) has the form (2.1.4). In the example § 2.2, only 5 is a successful position of G so that $S_g = \{5\}$ and from the fifth equation of the example, $5G^{\varepsilon} = 1$.

2.3.2. The position i is *ideal* if the morphism $\mathbf{i} \cdot g$ is ideal. Let I_g be the set of ideal positions of g. In the example of § 2.2, positions 2, 4, and 6 are ideal for G so that $I_G = \{2, 4, 6\}$.

2.3.3 The position i is *potentially singular* if, for some (unique) $i' \in [n]$, $i \cdot g = 0_p \oplus \mathbf{i}'$. We define the function $g^{\nu}: P_g \to [n]$, where P_g is the set of potentially singular positions of g, by $ig^{\nu} = i'$. Thus if i is potentially singular, the ith equation in (2.1.2) has the form (2.1.3). In example of § 2.2, the positions 1, 3, 7, 8, and 9 are potentially singular for G so that $P_G = \{1, 3, 7, 8, 9\}$ and from the first, third, seventh, eighth, and ninth equations of the example: $1G^{\nu} = 5$, $3G^{\nu} = 7$, $7G^{\nu}.= 3$, $8G^{\nu} = 8$, $9G^{\nu} = 2$,

For the *final classification* of the component positions of g, we note that exactly one of three possibilities can occur. Either for every r, i is a potentially singular position for g^r or not. If not, either $\mathbf{i} \cdot g^r$ is $\mathbf{j} \oplus 0_n$ for some $j \in [p]$, or $\mathbf{i} \cdot g^r$ is ideal. Thus the possibilities are:

(2.3.4) For every $r \geqq 1$ there is an $i_r \in [n]$ such that $\mathbf{i} \cdot g^r = 0_p \oplus \mathbf{i}_r$.

(Recall the definition of g^r, given in 1.5.)

(2.3.5) There is some $r \geqq 1$ and some $j \in [p]$ such that $\mathbf{i} \cdot g^r = \mathbf{j} \oplus 0_n$.

(2.3.6) There is some $r \geqq 1$ such that $\mathbf{i} \cdot g^r$ is ideal.

2.3.7. We say i is a *singular* position of g if (2.3.4) occurs. In the example of § 2.2, positions 3, 7 and 8 are singular for G. (The singular positions are responsible for the existence of more than one solution of the iteration equation, as will be seen below.)

2.3.8. We say i is a *power successful* position of g if (2.3.5) holds. Note that if $\mathbf{i} \cdot g^r = \mathbf{j} \oplus 0_n$, for some $j \in [p]$, then $\mathbf{i} \cdot g^s = \mathbf{j} \oplus 0_n$ for all $s > r$. In the example of § 2.2, the power successful positions are 1 and 5. (Since 5 is a successful position for G, it is clearly power successful: (2.3.5) holds with $r = 1$. Position 1 is power successful, since $\mathbf{1} \cdot G^2 = G_5 = \mathbf{1}: 1 \to 1$.)

2.3.9. Lastly, we say i is a *power ideal* position if (2.3.6) holds. Note that if $\mathbf{i} \cdot g^r$ is ideal, so is $\mathbf{i} \cdot g^s$, all $s > r$. In the example of § 2.2, positions 2, 4, 6 and 9 are power ideal for G. (Indeed 2, 4 and 6 are ideal, and $\mathbf{9} \cdot G^2 = G_2$, an ideal tree.)

It is convenient to define a function $g^{\sigma}: [n] \to [n]^+ \cup [n]^{\infty}$ where $[n]^+([n]^{\infty})$ is the set of nonempty finite (infinite) sequences of elements of $[n]$.

2.3.10. *Definition of* $g^{\sigma}: [n] \to [n]^+ \cup [n]^{\infty}$. Suppose i is a singular position of g. Then ig^{σ} is the infinite sequence in $[n]^{\infty}$

$$ig^{\sigma} = u_1 u_2 u_3 \cdots$$

where $u_1 = i$ and for each $r \geqq 1$, $u_r g^{\nu} = u_{r+1}$ or equivalently,

$$\mathbf{i} \cdot g^r = \quad \mathbf{u}_r \cdot g \quad = 0_p \oplus \mathbf{u}_{r+1}. \tag{2.3.11}$$

A formal proof of (2.3.11) is by induction on r. The truth of the equation is, on reflection, obvious.

Suppose i is a power successful position of g. Then ig^{σ} is the finite sequence $u_1 u_2 \cdots u_t$ where $t \geqq 1$, $u_1 = i$, $u_t \in S_g$ and where, for $r < t$, (2.3.11) holds.

Lastly, suppose i is a power ideal position of g. We define ig^σ to be the finite sequence $u_1u_2 \cdots u_t$, where $u_1 = i$, $u_t \in I_g$ and for $r < t$, (2.3.11) holds. Thus $\mathbf{i} \cdot g^t = \mathbf{u}_t \cdot g$ is ideal.

We note that for any potentially singular position i of g if $ig^\nu = i$, then i is a singular position and $ig^\sigma = iii \cdots$.

2.3.12. We let M_g, K_g, and L_g denote the set of power successful, singular and power ideal positions of g, respectively. We will, however, usually drop the subscripts.

For the sake of illustration, we will compute several values of the function $G^\sigma: [9] \to [9]^+ \cup [9]^\infty$, where G is the tree of example in § 2.2. The position 3 is singular for G, and

$$3G^\sigma = 3\ 7\ 3\ 7\ 3\ 7 \cdots.$$

The position 1 is power successful for G, and

$$1 \cdot G^\sigma = 1\ \ 5.$$

The positions 9 and 2 are power ideal for G, and

$$9 \cdot G^\sigma = 9\ \ 2;$$

$$2 \cdot G^\sigma = 2.$$

2.3.13. Note that if $ig^\sigma = u_1u_2u_3 \cdots = i\, u_2u_3 \cdots$ and $u_1 = i$ is in M (resp., K, L), so is u_r, all $r \geqq 1$; similarly, if u_r, $r > 1$ is in M (resp., K, L), so is u_s, for all s, $1 \leqq s < r$. Note also that $u_2g^\sigma = u_2u_3u_4 \cdots$, so that $ig^\sigma = iv$, where $v = u_2g^\sigma$.

If $i \in M$ or L and $ig^\sigma = u_1u_2 \cdots u_{t+1}$, $t \geqq 0$, then the elements $u_1, \cdots, u_t$ are *distinct* potentially singular positions of g. Thus $t < n$ and (*using the finiteness of* n) it follows that

(2.3.14) i is a power successful position of g iff i is a successful position of g^n; in this case $\mathbf{i} \cdot g^n = \mathbf{j}$ for some $j \in [p]$;

(2.3.15) i is a power ideal position of g iff i is an ideal position of g^n; in this case $\mathbf{i} \cdot g^n$ is ideal;

(2.3.16) i is a singular position of g iff i is a potentially singular position of g^n; in this case $\mathbf{i} \cdot g^n = 0_p \oplus \mathbf{i}'$ for some $i' \in [n]$.

In particular, no position of g is singular iff for each $i \in [n]$, $\mathbf{i} \cdot g^n$ is ideal or successful.

2.3.17. Because the description of a morphism g in an iterative theory is often given in such a way that one can immediately "read off" the set P_g (as, for example, in (2.2.1) or (2.2.2)), we note that in (2.3.14), (2.3.15) and (2.3.16), the superscript "n" may be replaced by $\|P_g\| + 1$, where $\|P_g\|$ is the cardinality of P_g.

In the example, $\|\mathrm{P_G}\| = 5$. Thus, in this case, we have $n = 9$ and $\|P_G\| + 1 = 6$.

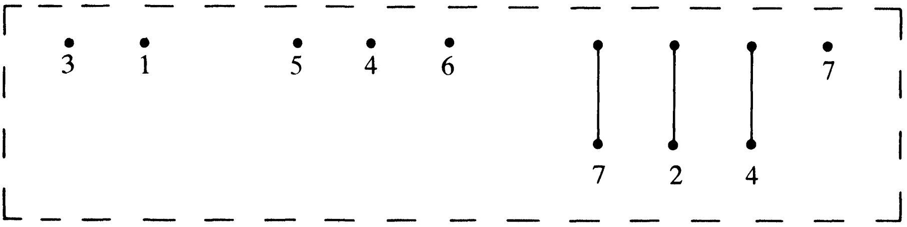
2.4. We return to the example of § 2.2. As mentioned, it would be convenient to rearrange the system of equations (2.2.2) so that the power successful, singular and power ideal positions occur together. This amounts to "rearranging" G. Recall from 2.3.7–2.3.9 that the set M of power successful positions of G is $\{1, 5\}$; K, the set of singular positions, is $\{3, 7, 8\}$, and the set L of power ideal positions of G is $\{2, 4, 6, 9\}$. Let $\pi: [9] \to [9]$ be a permutation that maps M bijectively onto $\{1, 2\}$, K bijectively onto $\{3, 4, 5\}$ and L bijectively onto $\{6, 7, 8, 9\}$. For definiteness, suppose that π is the unique such permutation which is order preserving so that π is given by reading top-down the following table

i	1	2	3	4	5	6	7	8	9
$i\pi$	1	6	3	7	2	8	4	5	9

and π^{-1} is given by reading the table bottom-up. Then the tree $\bar{G} = \pi^{-1} \cdot G \cdot (I_1 \oplus \pi)$ is

(2.4.1)

$\bar{G} =$



and the system of equations corresponding to the iteration equation for $\bar{G}$ (the solution is depicted by (2.6.1)) is

$$\xi_1 = \xi_2, \quad \xi_2 = \mathbf{1}\colon 1 \to 1,$$

$$(2.4.2) \quad \xi_3 = \xi_4, \quad \xi_4 = \xi_3, \quad \xi_5 = \xi_5,$$

$$\xi_6 = \bar{G}_6' \cdot \xi_6, \quad \xi_7 = \bar{G}_7' \cdot \xi_1, \quad \xi_8 = \bar{G}_8' \cdot \xi_3, \quad \xi_9 = \xi_6.$$

See (2.2.2) for the meaning of the prime.

The function $\bar{G}^\nu$ is given by Table 1 and $\bar{G}^\sigma$ is given by Table 2.

TABLE 1

i	1	3	4	5	9
$i\bar{G}^\nu$	2	4	3	5	6

TABLE 2

	$\bar{G}^\sigma$
1	1 2
2	2
3	3 4 3 4 ···
4	4 3 4 3 ···
5	5 5 5 ···
6	6
7	7
8	8
9	9 6

The $\bar{G}^\sigma$-table verifies that the power successful positions for $\bar{G}$ are 1 and 2; the singular positions are 3, 4 and 5 and the remaining positions are power ideal. It is much easier to compute the solutions $\xi = (\xi_1, \xi_2, \cdots, \xi_9)\colon 9 \to 1$ of the iteration equation for $\bar{G}$ from (2.4.2) than to find the solutions to (2.2.2). The relation between G, $\bar{G}$ and the solutions of their iteration equations is discussed in general below.

2.5. Conjugation. Let $g, \bar{g}\colon n \to p + n$ be morphisms in the iterative theory I, and let π $[n] \to [n]$ be a permutation.

2.5.1. DEFINITION. (a) $\bar{g}$ is the *π-conjugate* of g if $\bar{g} = \pi^{-1} \cdot g \cdot (I_p \oplus \pi)$.

(b) $\bar{g}$ is *conjugate* to g if for some permutation π of $[n]$, $\bar{g}$ is the π-conjugate of g.

Note that in the case $p = 0$, if $\bar{g}$ is the π-conjugate of g, $\bar{g} = \pi^{-1} \cdot g \cdot \pi$; (thus the use of group theoretic terminology).

2.5.2. *Remark.* We observe that the relation "$\bar{g}$ is conjugate to g" is an equivalence relation. Indeed, if $\bar{g}$ is the π-conjugate of g, g is the π^{-1}-conjugate of $\bar{g}$; if g_2 is the π_1-conjugate of g_1, and g_3 is the π_2-conjugate of g_2, then g_3 is the $\pi_1\pi_2$-conjugate of g_1.

2.5.3. PROPOSITION. *Let $\bar{g}$ be the π-conjugate of g: $n \to p+n$. Then:* (a) *for each $r \geqq 0$, $\bar{g}^r = \pi^{-1} \cdot g^r \cdot (I_p \oplus \pi)$;*

(b) for each $i \in [n]$, i is a power successful (respectively: singular, power ideal) position for g iff $i\pi$ is a power successful (respectively: singular, power ideal) position of $\bar{g}$;

(c) *if ξ: $n \to p$ is a solution of the iteration equation for $\bar{g}$, then $\pi \cdot \xi$ is a solution of the iteration equation for g;*

(d) *the map $\xi \mapsto \pi \cdot \xi$ is a bijection between the set of solutions ξ: $n \to p$ of the iteration solution for $\bar{g}$ and the set of solutions of the iteration equation for $\bar{g}$.*

Proof. (a) The proof is by induction on r. The case $r = 0$ is trivial. Now

$$\begin{aligned}\bar{g}^{1+r} &= \bar{g}^r \cdot (I_p \oplus 0_n, \bar{g})\\ &= \pi^{-1} \cdot g^r \cdot (I_p \oplus \pi) \cdot (I_p \oplus 0_n, \bar{g})\end{aligned}$$

by the induction assumption

$$\begin{aligned}&= \pi^{-1} \cdot g^r \cdot (I_p \oplus \pi) \cdot (I_p \oplus 0_n, \pi^{-1} \cdot g \cdot (I_p \oplus \pi))\\ &= \pi^{-1} \cdot g^r \cdot (I_p \oplus 0_n, g \cdot (I_p \oplus \pi)) \quad \text{by (1.5.2)}\\ &= \pi^{-1} \cdot g^r \cdot (I_p \oplus 0_n, g)(I_p \oplus \pi) \quad \text{by (1.5.4) and (1.5.1)}\\ &= \pi^{-1} \cdot g^{r+1} \cdot (I_p \oplus \pi),\end{aligned}$$

which completes the induction.

(b) By part (a), for every $r \geqq 1$

$$\pi \cdot \bar{g}^r = g^r \cdot (I_p \oplus \pi).$$

Thus if i is a power successful position for g, for some $r \geqq 1$, $j \in [p]$, $\mathbf{i} \cdot g^r = \mathbf{j} \oplus 0_n$, and $\mathbf{i} \cdot g^r \cdot (I_p \oplus \pi) = \mathbf{j} \oplus 0_n = \mathbf{i} \cdot \pi \cdot \bar{g}^r = \mathbf{i\pi} \cdot \bar{g}^r$. Hence $i\pi$ is a power successful position for $\bar{g}$. By 2.5.2 and what was just shown, if $i\pi$ is a power successful position of $\bar{g}$, then $(i\pi)\pi^{-1} = i$ is a power successful position of g,

The remaining statements of (b) follow from part (a) in the same way.

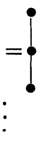
(c) Suppose $\xi = \bar{g} \cdot (I_p, \xi)$. Then $\xi = \pi^{-1} \cdot g \cdot (I_p \oplus \pi) \cdot (I_p, \xi) = \pi^{-1} \cdot g \cdot (I_p, \pi \cdot \xi)$, by (1.5.2). Thus $\pi \cdot \xi = g \cdot (I_p, \pi \cdot \xi)$, so that $\pi \cdot \xi$ is a solution of the iteration equation for g.

(d) The map $\xi \mapsto \pi \cdot \xi$ is a surjection, by 2.5.2 and part (c). If $\pi \cdot \xi = \pi \cdot \xi'$, clearly $\xi = \xi'$, so the map is also injective. This completes the proof.

2.6. Example continued. From the system of equations (2.4.2) it follows that every solution to the iteration equation of the tree $\bar{G}$, given in (2.4.1) is a nine-tuple of trees

$$\xi = (\xi_1, \xi_2, \xi_3, \xi_4, \xi_5, \xi_6, \xi_7, \xi_8, \xi_9)$$

such that $\xi_1 = \xi_2 = \mathbf{1}$; $\xi_3 = \xi_4$, ξ_5 is arbitrary, ξ_6 is the infinite tree

$\xi_6 =$ 

$$\xi_7 = \bar{G}_7' \cdot \mathbf{1}; \qquad \xi_8 = \bar{G}_8' \cdot \xi_3 \quad \text{and} \quad \xi_9 = \xi_6.$$

Conversely, every nine-tuple of trees satisfying the above conditions will be a solution of the iteration equation for $\bar{G}$.

The "general" solution of the iteration equation for $\bar{G}$ is depicted by Fig. 2.6.1; cf. (2.4.1).

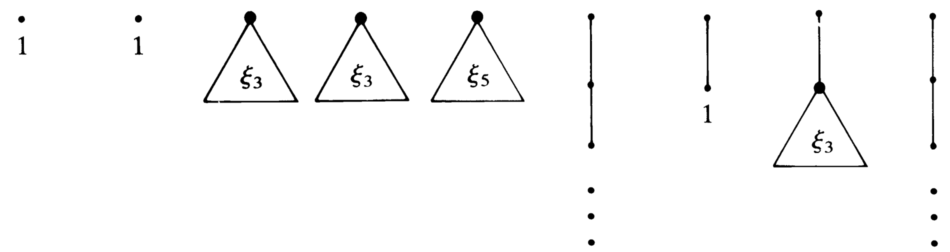


FIG. 2.6.1. *Solution to the iteration equation $\bar{G}$ (cf. (2.4.1)).*

Thus, by 2.5.3(d) all of the solutions of the iteration equation for G are the trees $\pi \cdot \xi$, where π is the permutation of [9] given in § 2.4 and ξ is a solution of the iteration equation for $\bar{G}$ (see Fig. 2.6.2).

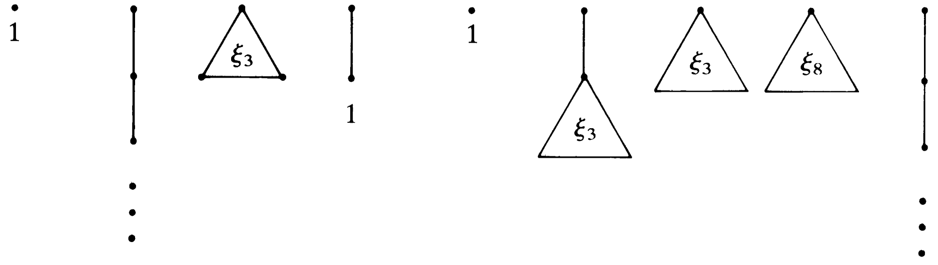


FIG. 2.6.2. *Solution to the iteration equation for G.*

The generalization of this method for solving the iteration equation is contained in Theorem of §2.8.

2.7. Let $g: n \to p+n$ be a morphism in an iterative theory. Let $M, K, L \subseteq [n]$ be respectively the set of power successful, singular and power ideal positions of g. Let the cardinalities of these sets be m, k and l respectively, so that $m+k+l=n$. Let π be a permutation of $[n]$ which maps M bijectively onto $[m]$, K onto $m+[k]$, and L onto $m+k+[l]$. Let $\bar{g}$ be the π-conjugate of g. Then, by 2.5.3(b),

(a) the set of power successful positions of $\bar{g}$ is [m];

(b) the set of singular positions of $\bar{g}$ is $m+[k]$;

(c) the set of power ideal positions of $\bar{g}$ is $m+k+[l]$.

We call a morphism $\bar{g}$ with the properties (a), (b), (c) a *mkl-morphism* ("*mkl*" is pronounced to rhyme with "nickle").

2.8. THEOREM. *Let $\bar{g}$: $n \to p+n$ be a mkl-morphism in an iterative theory. Then*

(a) *$\bar{g}$ is uniquely expressible in the form*

(2.8.1) $$\bar{g} = (a \oplus b \oplus 0_l, h),$$

where

(2.8.2) *$a: m \to p+m$ is base and a^m "factors through p"; i.e., for some (unique) base morphism $\bar{a}: m \to p$, $a^m: m \xrightarrow{a} p \xrightarrow{I_p \oplus 0_m} p+m$ (thus every position of a is* *power successful); note that* $m=0$ *if* $p=0$;

(2.8.3) $b\colon k\to k$ *is base*;

(2.8.4) $h\colon l\to p+m+k+l$ *is power ideal*;

(2.8.5) *moreover, for any morphisms* $\xi_1\colon m\to p$, $\xi_2\colon k\to p$, *the morphism* $f=h\cdot[(I_p,\xi_1,\xi_2)\oplus I_l]\colon l\to p+l$ *is power ideal.*

(b) *The solutions of the iteration equation for* $\bar{g}$ *are expressible in the form* (ξ_1,ξ_2,ξ_3) *where* $\xi_1\colon m\to p$, $\xi_2\colon k\to p$, $\xi_3\colon l\to p$ *satisfy*

(2.8.6) $\xi_1=a\cdot(I_p,\xi_1)$; *i.e.,* ξ_1 *is a solution of the iteration equation for* a;

(2.8.7) $\xi_2=b\cdot\xi_2$; *when* $p=0$, ξ_2 *is a solution of the iteration equation for* b;

(2.8.8) $\xi_3=f\cdot(I_p,\xi_3)$; *with* f *as in* (2.8.5), ξ_3 *is a solution of the iteration equation for* f.

(c) *The solution*,*of* (2.8.6) *is unique* (*and is denoted* $a^\dagger$; *in fact,* $a^\dagger=\bar{a}$). *For each choice of* ξ_1 *and* ξ_2, *the solution of* (2.8.8) *is unique* (*and is denoted* $f^\dagger$). *Further,* $f^\dagger=h^\dagger\cdot(I_p,\xi_1,\xi_2)$.

(d) *The solutions of the iteration equation for* $\bar{g}$ *are all expressible in the form*

$$(a^\dagger,\xi_2,\quad h^\dagger\cdot(I_p,a^\dagger,\xi_2))$$

where ξ_2 *satisfies* (2.8.7).

The proof of the Theorem 2.8 occupies §§2.9–2.14.

2.9. On (2.8.2). Let $i\in[m]$, so that i is a power successful position of $\bar{g}$. If i is not a successful position of $\bar{g}$, $\mathbf{i}\cdot\bar{g}=0_p\oplus\mathbf{i}'\colon 1\to p+n$, for some $i'\in[m+k+l]=[n]$. By 2.3.12, i' must belong to $[m]$. The morphism $a\colon m\to p+m$ is thus the base morphism corresponding to the function $a\colon[m]\to[p+m]$ determined by the requirement that

(2.9.1) $ia=p+i'$ if i is not a successful position of $\bar{g}$, $\mathbf{i}\cdot\bar{g}=0_p\oplus\mathbf{i}'$;

otherwise,

(2.9.2) $$ia=j\quad \text{if}\,\mathbf{i}\cdot\bar{g}=\mathbf{j}\oplus 0_{\mathrm{n}},\quad j\in[p].$$

Note that for $i\in[m]$,

(2.9.3) $i\bar{g}^\sigma=ia^\sigma=u_1u_2\cdots u_t$, where $u_1=i$, and u_t is a successful position of $\bar{g}$, and for $1\leqq r\leqq t$, $\mathbf{i}\cdot g^r=\mathbf{i}\cdot a^r=0_p\oplus\mathbf{u}_{r+1}$.

Lastly, since all of the entries $u_1,\cdots,u_t$ are necessarily distinct elements of $[m]$, we infer $t\leqq m$; thus

(2.9.4) $$\mathbf{i}\cdot a^t=\boldsymbol{i}\cdot a^m=\mathbf{j}\oplus 0_m,\quad \text{for some}\ j\in[p].$$

If we define $\bar{a}\colon[m]\to[p]$ by

$$i\bar{a}=j\qquad \text{if (2.9.4) holds,}$$

we have $a^m=\bar{a}\cdot(I_p\oplus 0_m)$, so that a^m factors through p, as claimed.

2.10. On (2.8.3). Let $i\in m+[k]$, so that i is a singular position of $\bar{g}$. Thus if

(2.10.1) $$i\cdot\bar{g}^\sigma=u_1u_2\cdots$$

each item u_r is a singular position of $\bar{g}$, and hence in particular, $u_2\in m+[k]$. We define the function $b\colon [k] \to [k]$ by the requirement that for $i \in m+[k]$

$$(i-m)b = i' - m$$

where $i' = u_2$ in (2.10.1). Then the morphism α, where

$$\alpha = a \oplus b \oplus 0_l\colon m+k \to p+m+k+l = p+n$$

is the morphism such that $\mathbf{i} \cdot \alpha = \mathbf{i} \cdot \bar{g}$, for all $i \in [m+k]$. Thus the first $m+k$ components of $\bar{g}$ are writable as $(a \cdot (I_{p+m} \oplus 0_{k+l}),\ b \cdot (0_{p+m} \oplus I_k \oplus 0_l))$ which equals $a \oplus b \oplus 0_l$.

2.11. On (2.8.4) and (2.8.5). The morphism $h\colon l \to p+m+k+l$ is defined by the requirement that its l-components are the last l-components of $\bar{g}$.

Thus, by 2.3.13, if $i \in [l]$, and if $(m+k+i)\bar{g}^\sigma = u_1 u_2 \cdots u_t$, then

$$ih^\sigma = v_1 v_2 \cdots v_t$$

where $m+k+v_r = u_r$, for $r \in [t]$. Also $\mathbf{u}_t \cdot \bar{g} = \mathbf{m+k+i} \cdot \bar{g}^t$ is ideal; hence $\mathbf{v}_t \cdot h = \mathbf{i} \cdot h^t$ is ideal. Since $t \leqq l$, $\mathbf{i} \cdot h^l$ is ideal, for all $i \in [l]$. Thus h is power ideal, proving (2.8.4).

The assertion (2.8.5) follows immediately from the following more general fact (and from (3.1.2) of [5]).

2.11.1. PROPOSITION. *Let $h\colon l \to q+l$ and $\beta\colon q \to s$ be morphisms in an algebraic theory. If $f = h \cdot (\beta \oplus I_l)$, then for all $r \geqq 1$, $f^r = h^r \cdot (\beta \oplus I_l)$.*

Proof. The proof is by induction on r. When $r=1$, there is nothing to prove. Assume $f^r = h^r \cdot (\beta \oplus I_l)$. Then

$$\begin{aligned} f^{r+1} &= f^r \cdot (I_s \oplus 0_l, f) = f^r \cdot (I_s \oplus 0_l, h \cdot (\beta \oplus I_l)) \\ &= h^r \cdot (\beta \oplus I_t) \cdot (I_s \oplus 0_l, h \cdot (\beta \oplus I_t)). \\ &= h^r \cdot (\beta \oplus 0_l, h \cdot (\beta \oplus I_l)) \qquad \text{by (1.5.2), since } \beta \cdot (I_s \oplus 0_l) = \beta \oplus 0_l; \\ &= h^r \cdot (I_1 \oplus 0_l, h) \cdot (\beta \oplus I_l) \qquad \text{by (1.5.4)} \\ &= h^{r+1} \cdot (\beta \oplus I_l). \end{aligned}$$

The proof of (a) of the Theorem 2.8 is now complete.

2.12. Proof of (b). The iteration equation for $\bar{g}\colon n \to p+n$ is

$$\xi = \bar{g} \cdot (I_p, \xi), \quad \text{where } \xi\colon n \to p. \tag{2.12.1}$$

Using (2.8.1) and writing $\xi = (\xi_1, \xi_2, \xi_3)$ (as in the Theorem 2.8(b)) the equation (2.12.1) becomes

$$\begin{aligned} (\xi_1, \xi_2, \xi_3) &= (a \oplus b \oplus 0_l, h) \cdot (I_p, \xi_1, \xi_2, \xi_3) \\ &= (a \cdot (I_p, \xi), \quad b \cdot \xi_2, \quad h \cdot (I_p, \xi_1, \xi_2, \xi_3)) \end{aligned} \tag{2.12.2}$$

by (1.5.4), (1.5.2) and (1.5.3).

Thus the equation (2.12.2) is equivalent to the three equations (2.8.6), (2.8.7) and

$$\xi_3 = h \cdot (I_p, \xi_1, \xi_2, \xi_3). \tag{2.12.3}$$

This latter equation is the same as (2.8.8), since

$$h \cdot (I_p, \xi_1, \xi_2, \xi_3) = h \cdot [(I_p, \xi_1, \xi_2) \oplus I_l] \cdot (I_p, \xi_3) \tag{2.12.4}$$

by (1.5.2).

2.13. Proof of (c). Every solution of (2.8.6) is also a solution to

$$\xi_1 = a^m \cdot (I_p, \xi_1). \tag{2.13.1}$$

By (2.8.2), there is a base morphism $\bar{a}\colon m \to p$ such that $a^m = \bar{a} \cdot (I_p \oplus 0_m)$. Thus, if ξ_1 is a solution of (2.13.1),

$$\xi_1 = \bar{a} \cdot (I_p \oplus 0_m) \cdot (I_p, \xi_1) = \bar{a}.$$

Thus (2.13.1) has at most one solution, and (2.8.6) has at most one solution, viz., $\bar{a}$. But, by the definition of $\bar{a}$ in § 2.9,

$$a \cdot (I_p, \bar{a}) = \bar{a},$$

which proves (2.8.6) has at least one solution.

Since f is power ideal, by (2.8.5), there is a unique solution $f^\dagger$ to (2.8.8). The fact that $f^\dagger = h^\dagger \cdot (I_p, \xi_1, \xi_2)$ follows immediately from the following easily proved fact.

2.13.2. PROPOSITION. *Let $h\colon l \to q + l$ be a power ideal morphism in an iterative theory I, and let $\beta\colon q \to s$ be an arbitrary morphism in I. Then $[h \cdot (\beta \oplus I_l)]^\dagger = h^\dagger \cdot \beta$.*

Proof of 2.13.2. By 2.11.1, $h \cdot (\beta \oplus I_l)\colon l \to s + l$ is power ideal since h is, and thus there is a unique solution $[h \cdot (\beta + I_l)]^\dagger$ of its iteration equation. But

$$\begin{aligned} h \cdot (\beta \oplus I_l) \cdot (I_s, \quad h^\dagger \cdot \beta) &= h \cdot (\beta, h^\dagger \cdot \beta) && \text{by (1.5.2)} \\ &= h(I_q, h^\dagger) \cdot \beta && \text{by (1.5.4)} \\ &= h^\dagger \cdot \beta. \end{aligned}$$

Thus $h^\dagger \cdot \beta$ is one solution of the iteration equation for $h \cdot (\beta \oplus I_l)$.

2.14. On (d). The proof of (d) is immediate from parts (b) and (c).

This completes the proof of the Theorem 2.8.

A number of facts follow easily from the theorem and § 2.5.

2.15. COROLLARY. *Let $g\colon n \to p + n$ be an arbitrary morphism in an iterative theory. Let $\pi\colon [n] \to [n]$ be a permutation such that the π conjugate $\bar{g}$ of g is a mkl-morphism $(a \oplus n \oplus 0_l, h)$ as in § 2.8. Then every solution of the iteration equation for g is expressible as $\pi \cdot (a^\dagger, \xi_2, h^\dagger \cdot (I_p, a^\dagger, \xi_2))$ where $\xi_2\colon K \to p$ satisfies $\xi_2 = b \cdot \xi_2$.*

2.16. COROLLARY. *Let $\bar{g} = (a \oplus b \oplus 0_l, h)$ be a mkl-morphism in an iterative theory. The set of solutions of the iteration equation for $\bar{g}$ is in bijective correspondence with the set of solutions $\xi_2\colon k \to p$ of the equation*

$$\xi_2 = b \cdot \xi_2. \tag{2.16.1}$$

The proof of the Corollary 2.16 is immediate from the Theorem 2.8(d). Note that in the case $k = 0$ (i.e., when there are no singular positions for $\bar{g}$) there is a unique morphism $0 \to p$; thus the iteration equation for $\bar{g}$ (and hence for all morphisms conjugate to $\bar{g}$) will have a unique solution, i.e., will have exactly one solution. A morphism $g\colon n \to p + n$ having no singular positions is called *nonsingular.* In this terminology, we have proved the next corollary.

2.17. COROLLARY. *If $g\colon n \to p + n$ is a nonsingular morphism in an iterative theory, the iteration equation for g has a unique solution (denoted $g^\dagger$).*

The converse of this corollary is almost always true. The statement of the corollary really tells the whole story. There are, however, some "exceptional" combinations of iterative theory I and target p of ξ_2 such that the equation (2.16.1) has a unique solution for any value of k. First we tabulate the cases which yield a multiplicity of solutions—or none—independent of k so long as k is positive. To this end let $\mathcal{N}$ be the only iterative theory which has no morphisms $1 \to 0$, i.e., the iterative theory all of whose morphisms are base morphisms.

2.18. PROPOSITION. *The equation $\xi = b \cdot \xi$ (where b: $k \to k$ is a base morphism in the iterative theory I, where ξ: $k \to p$ and where $k > 0$) has two or more solutions in I provided that*

(a) $\perp: 1 \to 0$, $\perp': 1 \to 0$ *are in I and* $\perp \neq \perp'$, *or*

(b) $I \neq \mathcal{N}$ *and* $p \geqq 1$, *or*

(c) $I = \mathcal{N}$ *and* $p \geqq 2$.

The equation has no solution when

(d) $I = \mathcal{N}$ *and* $p = 0$.

Proof. Note that the equation $\xi = b \cdot \xi$ requires only that certain components of ξ be equal to certain other components of ξ. Hence, if each of the k components of ξ are equal, the equation in question is satisfied. The solutions we will identify will have all their components equal so we need specify only what the components are.

Case (*a*). $\perp \cdot 0_p$, $\perp' \cdot 0_p$,

Case (*b*). $\perp \cdot 0_p$, $I_1 \oplus 0_{p-1}$,

Case (*c*). $I_1 \oplus 0_{p-1}$, $0_1 \oplus I_1 \oplus 0_{p-2}$.

The final case is obvious since there are no morphisms with target 0.

The following is essentially Theorem B of [8].

2.19. COROLLARY. *Let g: $n \to p + n$ be a morphism in an iterative theory I. A necessary and sufficient condition that the iteration equation for g have a unique (i.e., one and only one) solution is that*

(2.19.1) *g be nonsingular, i.e.,* $K_g = \varnothing$,

or

(2.19.2) *I have exactly one morphism* $1 \to 0$ *and* $p = 0$,

or

(2.19.3) $I = \mathcal{N}$ *and* $p = 1$.

Proof. See §§ 2.18, 2.16, and 2.5.

2.20. COROLLARY. *Let I be an iterative theory having at least two distinct morphisms* $1 \to 0$. *Then the iteration equation for a morphism g: $n \to n$ has a unique solution iff g is power ideal.*

Proof. If g is power ideal, g is nonsingular, so that by the corollary 2.17 the iteration equation for g has a unique solution. Conversely, by the corollary 2.19, if the iteration equation for g has a unique solution, g is nonsingular. But a nonsingular morphism g: $n \to n$ is clearly power ideal. This completes the proof.

The following is a special case of the Corollary 2.19.

2.21. In Tr, a necessary and sufficient condition that the iteration equation for g have a unique solution is that g be nonsingular.

3. The category Pit of $I_1^\dagger$-iterative theories. The goal of this section is to show that given any morphism $\perp: 1 \to 0$ in an iterative theory, by requiring only that $(0_p \oplus I_1)^\dagger = 0_p \oplus \perp$ (for each $p \geqq 0$) the scalar iteration operation may be consistently extended to all scalar morphisms. This fact follows from 3.7, Universality Theorem.

3.1. PROPOSITION. *Let F: $J_1 \to J_2$ be an ideal theory-morphism between ideal theories J_1, J_2 (i.e., if f is an ideal morphism in J_1 then fF is an ideal morphism in J_2). If g: $n \to p + n$ is a nonsingular morphism in J_1 then gF is a nonsingular morphism in J_2.*

Proof. The proof is immediate from the fact (cf. 2.3.13) that g is nonsingular iff for each $i \in [n]$, $i \cdot g^n$ is ideal or successful. □

In particular, if each J_i is iterative and g is nonsingular then the iteration equations for g and gF have unique solutions, $g^\dagger$ and $(gF)^\dagger$, respectively. Moreover, $g^\dagger F = (gF)^\dagger$.

This nice behavior of F contrasts sharply with the case that the theory-morphism F is not ideal. This case is taken up in the Lemma 3.2. For that discussion we recall that each morphism $f: 1 \to p$ (in any algebraic theory) is a solution to the iteration equation for the base morphism $0_p \oplus I_1: 1 \to p+1$ in that theory. This base morphism corresponds to the function $[1] \to [p+1]$ whose value is $p+1$.

3.2. LEMMA. *If $F: J_1 \to J_2$ is a theory-morphism between ideal theories which is* not *ideal, then there exists a biscalar ideal morphism h in J_1 such that $hF = I_1$.* ("Biscalar" means "$1 \to 1$".)

Proof. By hypothesis there exists a scalar ideal morphism $f: 1 \to p$ in J_1 such that fF is base (so that $p > 0$). Let $h = f \cdot \mu$, where $\mu: p \to 1$ is base. Then h is ideal and $\text{hF} = (f \cdot \mu)F = fF \cdot \mu F = fF \cdot \mu = I_1$.

PROPOSITION. *Let $F: J_1 \to J_2$ be a theory-morphism between iterative theories J_1, J_2 which is not ideal. The requirement* R1 *on extending the iteration operation in J_2 implies the requirement* R2.'

R1. *For all scalar ideal f in J_1, if $fF = 0_p \oplus I_1$, then $(fF)^\dagger$ is defined and $(fF)^\dagger = f^\dagger F$.*

R2. *For all p, $(0_p \oplus I_1)^\dagger$ is defined in J_2 and $(0_p \oplus I_1)^\dagger = 0_p \oplus I_1^\dagger$.*

Proof. According to the lemma, there exists an ideal $h: 1 \to 1$ in J_1 such that $hF = I_1$. According to R1, we must define $\text{I}_1^\dagger = h^\dagger F$ (where, of course, $h^\dagger$ is the unique solution to the iteration equation for h). For each $p > 0$, let $g_p = h \cdot (0_p \oplus I_1) = 0_p \oplus h$. Then, $g_p: 1 \to p+1$ is ideal and $g_p F = 0_p \oplus I_1$. According to R1, we must define $(0_p \oplus I_1)^\dagger = g_r^\dagger F$. Now, as may readily be shown, $g_p^\dagger = 0_p \oplus h^\dagger$. Hence

$$(0_p \oplus I_1)^\dagger = g_p^\dagger F = (0_p \oplus h^\dagger)F = 0_p \oplus h^\dagger F = 0_p \oplus I_1^\dagger.$$

Note that when $p = 0$, R2 asserts only that $I_1^\dagger$ is defined.

3.3. DEFINITION. By an *$I_1^\dagger$-iterative theory*, we mean a *pointed iterative theory* (i.e., a pair $(J, \perp)$ where $\perp: 1 \to 0$ is a morphism in the iterative theory J) *in which we complete the scalar iteration operation by defining for all p, $(0_p \oplus I_1)^\dagger = 0_p \oplus \perp$.* (We note that in an iterative theory the only scalar morphism $1 \to p+1$ whose iteration equation does not have a unique solution is $0_p \oplus I_1$.) *Thus any pointed iterative theory may be regarded as an $I_1^\dagger$-iterative theory.* If it is necessary or desirable to be very explicit, the $I_1^\dagger$-iterative theory $(J, \perp)$ may be described by: $(J, I_1^\dagger = \perp)$.

Before introducing the category of $I_1^\dagger$-iterative theories we note the following.

3.4. PROPOSITION. *Let $F: (J_1, \perp) \to (J_2, \perp)$ be a theory-morphism $F: J_1 \to J_2$ between iterative theories such that $\perp F = \perp$. Then the conditions* (3.4.1) *and* (3.4.2) *on F are equivalent. In particular, if F preserves idealism, then* (3.4.1) *is satisfied and so* (3.4.2) *is too.*

(3.4.1) *For all scalar ideal morphisms $f: 1 \to p+1$, $fF = 0_p \oplus I_1 \Rightarrow f^\dagger F = 0_p \oplus \perp$.*

(3.4.2) *Regarding $(J_i, \perp)$ as an $I_1^\dagger$-iterative theory, $i \in [2]$, F preserves scalar iteration.*

Proof. The proof is obvious.

Notice that (3.4.1) does not require defining $(0_p \oplus I_1)^\dagger$ while (3.4.2) does.

3.5. Definition. The category Pit[3] of $I_1^\dagger$ -iterative theories has $I_1^\dagger$ -iterative theories as objects. A morphism F: $(J_1, \perp) \to (J_2, \perp)$ in this category is a theory morphism such that $\perp F = \perp$ and such that (3.4.1) holds (or, equivalently, (3.4.2) holds).

Remark. The proof that morphisms between $I_1^\dagger$-iterative theories are closed under composition is a bit more obvious using (3.4.2) rather than (3.4.1).

3.6. We employ the following conventions: Γ_Δ is obtained from Γ (see §1.6) by adjoining a "new" element Δ to Γ_1; Δ: $1 \to 1$ is an atomic tree in $\Gamma_\Delta$tr.

The following is fundamental. Recall from [7] that Γtr is the iterative theory freely generated by Γ.

Δ-Lemma. *For any tree* $\perp$: $1 \to 0$ *in* Γtr *there is a unique theory morphism* Φ: $\Gamma_\Delta\text{tr} \to \Gamma\text{tr}$ *satisfying*

(a) $\Delta\Phi = I_1$,

(b) $\Delta^\dagger\Phi = \perp$,

(c) $\gamma\Phi = \gamma$, *all* γ *in* Γ,

(d) *if* $g\Phi$ *is base, say* $g\Phi = \mathbf{j}$: $1 \to p+1$, *then* $g = \Delta^r \cdot \mathbf{j}$ *for some* $r \geqq 0$.

The proof of the Δ-lemma is given in §4.

Remark. If we distinguish $\Delta^\dagger$: $1 \to 0$ in Γ_Δ tr and distinguish $\perp$ in Γ tr and treat the two theories as $I_1^\dagger$-iterative theories, then, as may readily be seen from (d), Φ: $(\Gamma_\Delta \text{ tr}, I_1^\dagger = \Delta^\dagger) \to (\Gamma \text{ tr}, I_1^\dagger = \perp)$ becomes a morphism in the category Pit.

3.7. Universality Theorem. *Let* $\Box \in \Gamma_0$ *and let* J *be an iterative theory. Given an arbitrary function* $\mathbf{F}$: $\Gamma \to J$ *taking* $\gamma \in \Gamma_n$ *into* $\gamma\mathbf{F}$: $1 \to n$ *such that* $\perp = \Box F$, *there is a unique morphism*

$$(\Gamma \text{ tr}, I_1^\dagger = \Box) \xrightarrow{\mathbf{F}'} (J, I_1^\dagger = \perp) \quad \textit{in } \text{Pit}$$

which makes the following diagram commute:

$$\begin{tikzcd}
\Gamma \arrow[rr, "\gamma \in \Gamma_n \mapsto \text{atomic tree } \gamma:\, 1 \to n"] \arrow[drr, "\mathbf{F}"'] & & (\Gamma\text{tr}, I_1^{+} = \Box) \arrow[d, "\mathbf{F}'"] \\
& & (J, I_1^{+} = 1).
\end{tikzcd}$$

Proof. Let $\Gamma' = \{\gamma \text{ in } \Gamma | \gamma\mathbf{F} \text{ is ideal}\}$ so that $\Box \in \Gamma_0'$. Let $\mathbf{U}$: $\Gamma \to \Gamma_\Delta'$ tr be defined as follows:

$$\gamma\mathbf{U} = \Delta \cdot \mathbf{i}, \quad \text{if } \gamma\mathbf{F} = \mathbf{i}\text{: } 1 \to n \text{ is base;}$$

$$\Box\mathbf{U} = \Delta^\dagger;$$

$$\gamma\mathbf{U} = \gamma \quad \text{otherwise.}$$

Notice that, for all γ, $\gamma\mathbf{U}$ is an ideal morphism. Then there is a unique theory morphism

[3] "Pit" is suggested by "pointed iterative theory". As noted in definition 3.3 any pointed iterative theory "may be regarded" as an $I_1^\dagger$-iterative theory. "Pit", as used here, may be regarded as an abbreviation for "Pit($I_1^\dagger$)".

(since Γ tr is, as an iterative theory, freely generated by Γ; cf. [7])

$$\mathbf{U}'\colon \Gamma\,\mathrm{tr} \to \Gamma'_\Delta\,\mathrm{tr}$$

which extends $\mathbf{U}$—or more exactly—makes the following diagram commute:

$$\begin{tikzcd}
\Gamma \arrow[rr, "\gamma \in \Gamma_n \mapsto \text{atomic tree } \gamma\colon 1 \to n"] \arrow[rrd, "\mathbf{U}"'] & & \Gamma\mathrm{tr} \arrow[d, "\mathbf{U}'"] \\
& & \Gamma'_\Delta\mathrm{tr}.
\end{tikzcd}$$

Notice that, by proposition 3.4,

$$\mathbf{U}'\colon (\Gamma\,\mathrm{tr},, I_1^\dagger = \Box) \to (\Gamma'_\Delta\,\mathrm{tr}, I_1^\dagger = \Delta^\dagger) \tag{3.7.1}$$

is a Pit morphism since $\Box\mathbf{U}' = \Delta^\dagger$ and U' preserves idealism.

Similarly, if we define $\mathbf{E}\colon \Gamma' \to J$ by:

$$\gamma\mathbf{E} = \gamma\mathbf{F} \text{ for } \gamma \text{ in } \Gamma'\text{; in particular, } \Box\mathbf{E} = \perp;$$

we obtain a theory morphism $\mathbf{E}'\colon \Gamma'\,\mathrm{tr} \to J$ which "extends" $\mathbf{E}$. Again, we notice

$$\mathbf{E}'\colon (\Gamma'\,\mathrm{tr}, I_1^\dagger = \Box) \to (J, I_1^\dagger = \perp) \tag{3.7.2}$$

is a Pit morphism.

Applying the Δ-lemma in the case $\perp = \Box$, with "Γ'" in place of "Γ", we readily conclude (as already noted in the Remark of 3.6

$$\Phi\colon (\Gamma'_\Delta\,\mathrm{tr}, I_1^\dagger = \Delta^\dagger) \to (\Gamma'\,\mathrm{tr}, I_1^\dagger = \Box) \tag{3.7.3}$$

is a Pit morphism.

We now define

$$\mathbf{F}'\colon \Gamma\,\mathrm{tr} \overset{\mathbf{U}'}{\text{---}} \Gamma'_\Delta\,\mathrm{t} \xrightarrow{\Phi} \Gamma'\,\mathrm{tr} \xrightarrow{\mathbf{E}'} J. \tag{3.7.4}$$

One readily checks that $\mathbf{F}'$ "extends" $\mathbf{F}$ and notes that $\mathbf{F}'$ is a Pit morphism since $\mathbf{U}'$, Φ, $\mathbf{E}'$ all are, i.e.,

$$\mathbf{F}'\colon (\Gamma\,\mathrm{tr}, I_1^\dagger = \Box) \to (J, I_1^\dagger = \perp)$$

is a Pit morphism.

The uniqueness of $\mathbf{F}'$ follows from the fact that Γ tr is the smallest subtheory of Γ Tr which contains the atomic trees Γ and is closed under scalar iteration of ideal (scalar) morphisms.

3.8. Significance of the Universality Theorem. The Universality Theorem implies that all "laws" of iterative theories remain true when the meaning of $g^\dagger$ is extended to all scalar g. Indeed, if "E_1" and "E_2" are iterative theory expressions involving only the scalar† and if the assertion

(3.8.1) for all scalar ideal $f, g, \cdots (E_1 = E_2)$

is valid in all iterative theories then the assertion

(3.8.2) for all scalar $f, g, \cdots (E_1 = E_2)$

is valid in all $I_1^\dagger$-iterative theories.

(By a (scalar) *iterative theory expression*, we mean an expression constructed from letters $f, g, \cdots$ to denote arbitrary scalar ideal morphisms, symbols to denote base scalar morphisms, symbols for the algebraic theory operations, and a symbol $(^\dagger)$ for scalar iteration of ideal morphisms. For convenience one may include "0_p", "I_p", etc.).

Thus, while the extended scalar iteration operation was uniquely determined by one iterative theory identity (viz., $[0_p \oplus g]^\dagger = 0_p \oplus g^\dagger$) once $I_1^\dagger$ was assigned a meaning, it turns out that the extended operation satisfies *all* scalar iterative theory identities (see [1]).

4. Proof of the "Δ-lèmma". We will use the somewhat more suggestive notation Δ^∞ for the tree $\Delta^\dagger: 1 \to 0$ in Γ_Δ tr, and we will say a tree g in Γ_Δ tr "has no Δ^∞-subtrees" if there is no vertex v of g such that the tree of descendants of v (see [7, p. 9]) is isomorphic to Δ^∞.

Before defining the function Φ we define a function $T \to \Gamma$ tr whose domain T is the set (actually, subtheory) of trees in Γ_Δ tr having no Δ^∞-subtrees.

4.1. Definition. If $g: n \to p$ is a tree in T, then $g_0: n \to p$ is the tree in Γ tr obtained from g by replacing every occurrence of $\Delta: 1 \to 1$ in g by $I_1: 1 \to 1$.

For example, $(\Delta^2 \cdot b)_0 = b$, for any base $b: 1 \to p$.

We list the following obvious properties of the function $g \mapsto g_0$.

4.2. Proposition. (a) *Suppose both $g: n \to p$ and $h: p \to q$ are in T. Then so is $g \cdot h$, and $(g \cdot h)_0 = g_0 \cdot h_0$.*

(b) *Suppose both $g: n \to p$ and $h: m \to p$ are in T. Then so is $(g, h): n + m \to p$ and $(g, h)_0 = (g_0, h_0)$; similarly if $g_i: n_i \to p_i$, $i = 1, 2$, are in T, then so is $g_1 \oplus g_2$ and $(g_1 \oplus g_2)_0 = (g_1)_0 \oplus (g_2)_0$.*

(c) *For any base morphism $b: n \to p$, $b_0 = b$; in particular, $(0_p)_0 = 0_p$, where 0_p is the unique tree $0 \to p$.*

Remark. As a consequence of this proposition, T is a subtheory of Γ_Δ tr and the function

$$T \xrightarrow{g \mapsto g_0} \Gamma \text{ tr}$$

is a theory morphism.

Proofs of all of these statements are straightforward if one first represents a Γ_Δ-tree by a "surmatrix" (as in [7]) and uses the fact that the substitution of I_1 for Δ is a monoid homomorphism on the sets of words which occur as entries in the surmatrices.

Let $g: n \to p$ be any Γ_Δ-tree. We may specify those vertices v such that the tree of descendants of v is isomorphic to Δ^∞ in the following way.

4.3. Proposition. *Let $g: n \to p$ be any Γ_Δ-tree. There is a tree $\hat{g}: n \to p + 1$ such that g is the composition*

$$g: n \xrightarrow{\hat{g}} p + 1 \xrightarrow{I_p \oplus \Delta^\infty} p$$

and such that $\hat{g}$ has no Δ^∞-subtrees. Further, if g is ideal, $\hat{g}$ may be chosen to be ideal.

4.4. *Remark.* Note that there may be several ways of choosing the tree $\hat{g}$. However if

$$\hat{g}_1 \cdot (I_p \oplus \Delta^\infty) = \hat{g}_2 \cdot (I_p \oplus \Delta^\infty)$$

and if $\hat{g}_1$ and $\hat{g}_2$ have no Δ^∞-subtrees, then $(\hat{g}_1)_0 = (\hat{g}_2)_0$.

We now are in a position to define the map Φ.

4.5. *Definition of* Φ: Γ_Δ tr $\to \Gamma$ tr. Let $g: n \to p$ be an arbitrary tree in Γ_Δ tr. Write g as $\hat{g} \cdot (I_p \oplus \Delta^\infty)$, using Proposition 4.3. We define

$$g\Phi = (\hat{g})_0 \cdot (I_p \oplus \perp).$$

Note that $g\Phi$ is well-defined by remark 4.4. Note also that if g has no Δ^∞-subtrees, $g\Phi = g_0$, so that in particular

$$\Delta\Phi = I_1;$$

$$\gamma\Phi = \gamma, \qquad \gamma \in \Gamma_n, \quad n \geqq 0;$$

also $\Delta^\infty\Phi = \perp$. Thus to prove the Δ-lemma we must show both that Φ is a theory morphism and that (d) (**3.6**) holds.

Before beginning this task, we observe that $g\Phi$ is described in geometric terms as the tree obtained from g by first replacing all Δ^∞-subtrees by $\perp$ and then replacing all remaining Δ's by I_1.

4.6. PROPOSITION. Φ *preserves composition.*

Proof. Suppose $g: n \to p$ and $h: p \to q$ are trees in Γ_Δ tr. Writing g and h as in 4.3 as

(4.6.1) $$g: n \overset{\hat{g}}{\text{——}} p+1 \xrightarrow{I_p \oplus \Delta^\infty} p, \qquad h: p \overset{\hat{h}}{\text{——}} q+1 \xrightarrow{I_q \oplus \Delta^\infty} q$$

we have

$$\begin{aligned} g \cdot h &= \hat{g} \cdot (I_p \oplus \Delta^\infty) \cdot \hat{h} \cdot (I_q \oplus \Delta^\infty) && \text{by (4.7.1)} \\ &= \hat{g} \cdot (\hat{h} \cdot (I_q \oplus \Delta^\infty) \oplus \Delta^\infty) && \text{by (1.5.1) when } g_2\text{: } 0 \to 0 \\ &= \hat{g} \cdot (\hat{h} \oplus I_1) \cdot (I_q \oplus (I_1, I_1)) \cdot (I_q \oplus \Delta^\infty) && \text{by two applications of (1.5.1).} \end{aligned}$$

Using the fact that $\hat{g} \cdot (\hat{h} \oplus I_1)(I_q \oplus (I_1, I_1))$ has no Δ^∞-subtrees, by 4.2.,

$$\begin{aligned} (g \cdot h)\Phi &= \hat{g}_0 \cdot (\hat{h}_0 \oplus I_1) \cdot (I_q \oplus (I_1, I_1)) \cdot (I_q \oplus \perp) \\ &= \hat{g}_0 \cdot (I_p \oplus \perp) \cdot \hat{h}_0 \cdot (I_q \oplus \perp) \\ &= g\Phi \cdot h\Phi. \end{aligned}$$

□

4.7 PROPOSITION. Φ *preserves source tupling.*

Proof. Let $g: n \to p$ and $h: m \to p$ be trees in Γ_Δ tr. Using 4.3, we write

$$g: n \overset{\hat{g}}{\text{—}} p+1 \xrightarrow{I_p \oplus \Delta^\infty} p \quad \text{and} \quad h: m \overset{\hat{h}}{\text{—}} p+1 \xrightarrow{I_p \oplus \Delta^\infty} p.$$

Hence

$$(g, h) = (\hat{g} \cdot (I_p \oplus \Delta^\infty), \hat{h} \cdot (I_p \oplus \Delta^\infty)) = (\hat{g}, \hat{h}) \cdot (I_p \oplus \Delta^\infty)$$

and $(\hat{g}, \hat{h})$ has no Δ^{∞}-subtrees. Hence, by 4.2,

$$\begin{aligned}(g, h)\Phi &= (\hat{g}_0, \hat{h}_0) \cdot (I_p \oplus \perp)\\ &= (\hat{g}_0 \cdot (I_p \oplus \perp), \hat{h}_0 \cdot (I_p \oplus \perp))\\ &= (g\Phi, h\Phi).\end{aligned}$$

4.8. PROPOSITION. *If* $b\colon n \to p$ *is base*, $b\Phi = b$.
Proof. Indeed, since b has no Δ^{∞}-subtrees, $b\Phi = b_0 = b$, by 4.2.

4.9. COROLLARY. $\Phi\colon \Gamma_{\Delta}\,\mathrm{tr} \to \Gamma\,\mathrm{tr}$ *is a theory morphism.*
Proof. Use propositions 4.6, 4.7 and 4.8.

To prove the Δ-lemma, it only remains to show that (d) holds. Suppose: $g\colon 1 \to p+1$ is ideal while $g\Phi$ is base and $g = \gamma \cdot g'$ where γ is atomic. Since $(\gamma \cdot g')\Phi = \gamma\Phi \cdot g'\Phi$ is base we must have $\gamma = \Delta$. Similarly, $g' = \Delta \cdot g''$, $g'' = \Delta \cdot g'''$, etc. This process must terminate, for otherwise $g\Phi = \perp \cdot 0_{p+1}$ which is ideal.

REFERENCES

[1] S. L. BLOOM, C. C. ELGOT AND J. B. WRIGHT, *Vector iteration in pointed iterative theories*, IBM Res. Rep. RC7322, IBM T. J. Watson Research Center, Yorktown Heights, NY, Sept. 1978.

[2] J. B. WRIGHT, J. W. THATCHER, E. G. WAGNER AND J. A. GOGUEN, *Rational algebraic theories and fixed point solutions*, IEEE Symp. Foundations of Comp. Sci., Hous.on, TX, October 1976.

[3] S. L. BLOOM AND C. C. ELGOT, The existence and construction of free iterative theories, J. Comput. System Sci., 12 (1976), no. 3.

[4] S. L. BLOOM, S. GINALI AND J. RUTLEDGE, *Scalar and vector iteration*, Ibid., 14 (1977), pp. 251–256.

[5] C. C. ELGOT, *Monadic computation and iterative algebraic theories*, Logic Colloquium '73, 80, Studies in Logic, North-Holland, Amsterdam, 1975.

[6] ———, *Structured programming with and without GO-TO statements*, IEEE Trans. Software Engrg., SE-2 (1976), pp. 41–54; Erratum and Corrigendum, Ibid., SE-2 (1976), p. 232.

[7] C. C. ELGOT, S. L. BLOOM AND R. TINDELL, *The algebraic structure of rooted trees*, IBM Res. Rep. RC-6230, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1976; extended abstract in Proc. Johns Hopkins 1977 Conf. Inf. Sci. and Systems; J. Comput. System. Sci., 16 (1978) No. 3, pp. 362–399.

[8] J. TIURYN, *On the domain of iteration in iterative algebraic theories*, Proc. Math. Found. in Comput. Sci. 1976, Gdansk, Lecture Notes in Comput. Sci. No. 45, Springer-Verlag, New York, 1976.

[9] J. MYCIELSKI AND W. TAYLOR, *A compactification of the algebra of terms*, Algebra Universalis, 6 (1976), pp. 159–163.

[10] D. SCOTT, *The lattice of flow diagrams*, Semantics of Algorithmic Languages, Lecture Notes in Math. 182, Springer-Verlag, New York, 1971, pp. 311–366.

[11] S. GINALI, *Iterative algebraic theories, infinite trees, and program schemata*, Dissertation, University of Chicago, June 1976.

[12] M. WAND, *A concrete approach to abstract recursive definitions*, Automata Languages and Programming, M. Nivat, ed., 1972, North-Holland pp. 331–341.

[13] W. LAWVERE, *Functional semantics of algebraic theories*, Proc. Nat. Acad. Sci. U.S.A., 50 (1963), pp. 869–872.

[14] S. EILENBERG AND J. B. WRIGHT, *Automata in general algebras*, Information and Control, 11 (1967), pp. 52–70.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0003 $01.00/0

REFINING NONDETERMINISM IN RELATIVIZED POLYNOMIAL-TIME BOUNDED COMPUTATIONS*

CHANDRA M. R. KINTALA† AND PATRICK C. FISCHER‡

Abstract. Let $\mathscr{P}_{g(n)}$ denote the class of languages acceptable by polynomial-time bounded Turing machines making at most $g(n)$ nondeterministic moves on inputs of length n. For any constructible $g(n)$, $\mathscr{P} \subseteq \mathscr{P}_{g(n)} \subseteq \mathscr{NP}$. The classes $\mathscr{P}_{g(n)}$, for various $g(n)$ of the form $(\log n)^k$, $k \geqq 1$, are relativized and the relationships among those relativized classes are studied. In particular, oracle sets are constructed which (1) make all the relativized classes equal (this follows from Baker, Gill and Solovay (1975)); (2) make all the classes associated with powers of log n different; (3) for any k, make all the classes below $(\log n)^k$ different while the kth power class is equal to relativized $\mathscr{NP}$. Results regarding closure of the $(\log n)^k$ classes under complementation are also given.

1. Introduction. The fundamental open question posed by Cook [2] and Karp [4] asks whether $\mathscr{P}$ equals $\mathscr{NP}$. Here $\mathscr{P}$ is the class of languages recognized in polynomial time by deterministic Turing machines, and $\mathscr{NP}$ is the class of languages accepted in polynomial time by nondeterministic Turing machines. A language L is said to be $\mathscr{NP}$-complete if L is in $\mathscr{NP}$ and every language in $\mathscr{NP}$ is polynomial-time reducible to L. A wide variety of problems (currently estimated at 2,000) have been shown to be $\mathscr{NP}$-complete. Inherent in such studies is the general conviction that any $\mathscr{NP}$-complete language has the difficulty of the whole class $\mathscr{NP}$ embedded in it and hence it acts as a "representative" of the class $\mathscr{NP}$.

For many $\mathscr{NP}$-complete languages, if n is the length of the input, a nondeterministic algorithm exists requiring a total number of moves which is a polynomial in n, but the number of nondeterministic moves is at most linear in n. (By a nondeterministic move, we mean a "strict" nondeterministic move where there are at least two choices for the next step of the machine.) As a matter of fact, one can easily show that the $v/3$-clique problem (given a graph G with v vertices and e edges, does G have a clique of size $v/3$?) is $\mathscr{NP}$-complete and that there is a nondeterministic polynomial algorithm for this problem making at most $O(\sqrt{n})$ nondeterministic moves on inputs of length n, given any "standard" representation of the graphs as strings. On the other hand, the nondeterministic polynomial-time algorithm for accepting primes described by Pratt [7] needs $\Omega(n^2)$ nondeterministic moves even though the recognition of primes has not been shown to be $\mathscr{NP}$-complete.

These observations on the nondeterministic-step requirements of the candidates for the languages in $\mathscr{NP}$-$\mathscr{P}$ suggest that we study the class $\mathscr{NP}$ from the viewpoint of restricted nondeterminism.

DEFINITION 1.1. For any function $g(n) \geqq 0$, let

$\mathscr{P}_{g(n)} = \{L | L \subseteq \{0, 1\}^*$ and there is a constant c such that L is accepted by a polynomial-time bounded Turing machine making at most $g(n)$ c-ary nondeterministic moves$\}$.

Clearly, $\mathscr{NP} = \bigcup_{k=0}^{\infty} \mathscr{P}_{n^k}$. Also,

$$\mathscr{P}_0 = \mathscr{P}_{\log n} \subseteq \mathscr{P}_{(\log n)^2} \subseteq \cdots \subseteq \mathscr{P}_{(\log n)^k} \subseteq \cdots \subseteq \mathscr{P}_n \subseteq \mathscr{NP}.$$

(The first equality comes from the fact that log n nondeterministic moves of a Turing machine can be simulated deterministically by following at most $c \cdot \log n$ branches, i.e. a

* Received by the editors June 28, 1978, and in revised form January 30, 1979.

† Computer Science Department, University of Southern California, Los Angeles, California 90007.

‡ Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802.

polynomial number of branches.)[1] Refining the original question $\mathcal{P} = ?\ \mathcal{NP}$, we can ask whether $\mathcal{P}_{(\log n)^k} = ?\ \mathcal{P}_{(\log n)^{k+1}}$ for any $k \geqq 1$. We have not been able to answer the latter question for any k and suspect that these questions will be difficult to solve. To support this we can relativize the question analogously to the work of Baker, Gill, and Solovay [1], and show that for every $k \geqq 1$, the corresponding relativized questions have affirmative answers for some oracles but negative answers for other oracles. We also construct oracles for each k such that the relativized version of the above hierarchy is distinct up to the kth level but collapses from the kth level onwards. The behavior of these relativized classes under the operation of complementation is also investigated. The results we obtain indicate that the power of even "small" amounts of nondeterminism in polynomial-bounded machines is not amenable to the traditional methods of analysis. On the other hand, in Kintala [5] related questions in some other familiar classes of computations have turned out to be tractable.

Note. Some readers may prefer to allow $O(g(n))$ binary nondeterministic moves instead of $g(n)$ c-ary nondeterministic moves in Definition 1.1. All our results in this paper will carry over with this definition also. We have, however, chosen the above definition because (i) it appears unnatural to us to restrict the machines to just two choices at any step and (ii) the proofs would not be simplified if the latter definition were used since the "fan-out" constant c would be replaced by a constant c' to be applied to $g(n)$.

Some of the results in this paper are contained in Chapter 3 of [5]. Related results were first presented in [6]. Many of the proof methods used here are analogous to those of Baker, Gill and Solovay [1].

2. The model and the definitions. The model for computations in this paper is the *query machine* which is a nondeterministic multitape Turing machine with an additional work tape called the *query tape*, and three distinguished states, called the *query state*, the *yes state*, and the *no state*. The action of a query machine is similar to that of a Turing machine with the following extension. When a query machine enters its query state, the next operation of the machine is determined by an oracle. An oracle for a set X will place the query machine into its "yes" state if the binary string written on the query tape is an element of X; otherwise the oracle places the machine into the "no" state. We will identify an oracle for a set X with the set X itself and shall deal only with recursive oracles. The *language* accepted by a query machine with oracle X is the set of input strings for which some possible computation of the machine halts in one of the designated *accepting* states. Henceforth, we fix $\{0, 1\}$ as the alphabet in which all the languages are encoded. A query machine is *polynomial-time bounded* if there is a polynomial $p(n)$ such that every computation of the machine on every input of length n halts within $p(n)$ steps, whatever oracle X is used.

A c-ary nondeterministic move of a machine is a move in which the number of choices for the next step of the machine is c. Such a nondeterministic move is sometimes said to have a *fan-out* of c. Any query machine M can be so designed that all the nondeterministic moves made by M have the same fan-out c for some constant c, which depends on M. The relativized version of Definition 1.1 can now be provided.

DEFINITION 2.1. For a given function $g(n)$ and any oracle X, let $\mathcal{P}^X_{g(n)} = \{L | L \subseteq \{0.\ 1\}^*$ and there is a constant c such that L is accepted by a polynomial-time bounded query machine with oracle X making at most $g(n)$ c-ary nondeterministic moves$\}$. We also define $\mathcal{NP}^X = \bigcup_{k=0}^{\infty} \mathcal{P}^X_{n^k}$.

[1] $\log n$ = the largest integer m such that $2^m \leqq n$.

A recursive enumeration of all nondeterministic polynomial-time bounded query machines can be obtained by listing all pairs $\langle Q, p\rangle$ where Q is a query machine and p is a polynomial and attaching a $p(n)$-clock to Q. The new machine Q' will halt whenever its computation exceeds $p(n)$ steps. We will use such an enumeration as our "standard enumeration" and let M_i^X denote the ith machine in this enumeration.

For each i, we can effectively determine a c_i and an a_i such that c_i is the fan-out of the nondeterministic moves made by M_i^X and $p_i(n) = a_i + n^{a_i}$ is a strict upper bound on the length of any computation by M_i^X, for any oracle X.

Given a time-constructible function $g(n)$ and an oracle X, we let $M_{i,g(n)}^X$ denote the query machine which results by attaching a $g(n)$-time clock to M_i^X.[2] This clock stops M_i^X if $g(n)$ nondeterministic moves are exceeded. We observe that c_i bounds the fan-out of $M_{i,g(n)}^X$ and that $p_i(n)$ still bounds its run time since the $g(n)$-time counter runs in parallel with the main computation of M_i^X and makes no nondeterministic moves of its own.

For the readability of our statements, we introduce the following additional terminology:

DEFINITION 2.2. Let

(1) $ML_{i,k}^X \triangleq M_{i,(\log n)^k}^X$;

(2) $\mathcal{PL}_k^X \triangleq \mathcal{P}_{(\log n)^k}^X$;

(3) $\mathcal{PL}^X \triangleq \bigcup_{k=1}^{\infty} \mathcal{PL}_k^X$.

For any oracle X,

$$\mathcal{P}^X = \mathcal{PL}_1^X \subseteq \mathcal{PL}_2^X \subseteq \cdots \subseteq \mathcal{PL}_k^X \subseteq \cdots \mathcal{PL}^X \subseteq \mathcal{P}_n^X \subseteq \mathcal{NP}^X.$$

We call the preceding hierarchy as the "relativized $\mathcal{PL}$-hierarchy."

A language A is said to be *polynomial-time reducible* to B if there is a function f computable by a deterministic Turing machine in time bounded by some polynomial, such that $x \in A$ if and only if $f(x) \in B$. (This definition is that of Karp [4].) For a given class $\mathcal{C}$ of languages, a language B is said to be *$\mathcal{C}$-complete if* $B \in \mathcal{C}$ and A is a polynomial-time reducible to B for every $A \in \mathcal{C}$. As mentioned in the introduction, many $\mathcal{NP}$-complete languages are in $\mathcal{P}_n$. We will exhibit $\mathcal{PL}_k$-complete languages for all $k \geqq 1$.

3. $\mathcal{PL} = ?\mathcal{PL}_{k+1}$ relativized.

Baker, Gill, and Solovay [1] have constructed an oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$. It is obvious that for any such A, $\mathcal{PL}_k^A = \mathcal{PL}_{k+1}^A$ for every $k \geqq 1$. We will construct here oracles for which these classes differ.

DEFINITION 3.1. For every $k \geqq 1$ and all oracles X, let

$$L_k(X) = \{w \mid (\exists y)[|y| = (\log(|w|))^k \text{ and } wy \in X]\}.$$

Clearly, $L_k(X) \in \mathcal{PL}_k^X$, since one can nondeterministically guess y, then check whether $wy \in X$.

THEOREM 3.2. *For every $k \geqq 2$, there is an oracle B_k such that $\mathcal{PL}_{k-1}^{B_k} \subsetneqq \mathcal{PL}_k^{B_k}$.*

Proof. We shall define the required oracle B_k in such a way that $L_k(B_k) \notin \mathcal{PL}_{k-1}^{B_k}$. Since $L_k(B_k) \in \mathcal{PL}_k^{B_k}$, the theorem will follow immediately. B_k will be constructed in $wy \in X$.

[2] A function $g(n)$ is said to be *time-constructible* if there is a *real-time*-bounded deterministic Turing machine which, given inputs of length n, will produce outputs of length exactly $g(n)$. Functions of the form $(\log n)^k$, $\log^k(n)$, n, n^k, etc., are all time-constructible. See [5], [6] for related discussions.

stages. We denote by $B_k(i)$ the finite set of strings placed into B_k prior to stage i. Let $B_k(0)=\varnothing$, $n_{-1}=0$ and start at stage 0.

Stage i. Choose a sufficiently large n_i so large that

(i) $n_i > n_{i-1}$,
(ii) $n_i+(\log n_i)^k > p_{i-1}(n_{i-1})$,
(iii) $c_i^{(\log n_i)^{k-1}} p_i(n_i) < 2^{(\log n_i)^k}$

where p_i and c_i are as described in Definition 2.2. The first two requirements can obviously be satisfied, and one can note that $2^{(\log n_i)^k}=(2^{\log n_i})^{(\log n_i)^{k-1}}$. Then a suitable n_i can always be found.

Simulate the machine $M=ML_{i,k-1}^{B_k(i)}$ on input $x=0^{n_i}$. If M accepts x in any of its possible computations, then place no string into B_k at this stage. If M rejects x, then add some string of the form xy such that $|y|=(\log |x|)^k$ and xy is not queried during any possible computation of M on x. Such a string exists because there are at most $c_i^{(\log n_i)^{k-1}}$ possible computations, each of length at most $p_i(n_i)$. Therefore, the number of queries the machine could have made in all its possible computations is $c_i^{(\log n_i)^{k-1}} \cdot p_i(n_i) < 2^{(\log n_i)^k}$ by condition (iii). So some such xy is available. Set $B_k(i+1)=B_k(i)\cup\{xy\}$ and go to stage $i+1$.

The computation of $ML_{i,k-1}^{B_k}$ on input $x=0^{n_i}$ is the same whether B_k or $B_k(i)$ is used as an oracle because conditions (i) and (ii) guarantee that no string queried by $ML_{i,k-1}^{B_k}$ in any computation of x is later added to B_k (strings are obviously never deleted from B_k). This is true because no string queried is longer than the run-time bound $p_i(n_i)$, while strings added, if any, after stage i are of length at least $n_{i+1}+(\log n_{i+1})^k > p_i(n_i)$. At stage i, we ensure that $ML_{i,k-1}^{B_k}$ does not recognize $L_k(B_k)$, since, by construction, $ML_{i,k-1}^{B_k}$ in any computation of x is later added to B_k (strings are obviously never deleted $|y|=(\log(|x|))^k$, that is, if and only if $x\in L_k(B_k)$. Hence $L_k(B_k)\notin \mathscr{PL}_{k-1}^{B_k}$. □

We can modify the above construction to obtain a uniform oracle B for all k by using the familiar dovetailing methods. We omit details.

THEOREM 3.3. *There is an oracle B such that* $\mathscr{PL}_{k-1}^B \subsetneqq \mathscr{PL}_k^B$ *for every* $k\geqq 2$, *i.e.*,

$$\mathscr{P}^B=\mathscr{PL}_1^B \subsetneqq \mathscr{PL}_2^B \subsetneqq \cdots \subsetneqq \mathscr{PL}_k^B \subsetneqq \cdots \subsetneqq \mathscr{PL}^B.$$

The above results can be interpreted in the following manner. The question $\mathscr{P}=?\ \mathscr{NP}$ asks whether providing "full" nondeterminism to a class of polynomial-time bounded machines increases the class of languages accepted by them. Baker, Gill, and Solovay [1] have shown that the relativized $\mathscr{P}=?\ \mathscr{NP}$ question has an affirmative answer for some oracles but a negative answer for other oracles, which is taken to be further evidence of the difficulty of the $\mathscr{P}=?\mathscr{NP}$ question. By the same token, our observations indicate that the refined questions as defined in the Introduction are also difficult in the sense that neither familiar diagonalization methods nor simulation methods are applicable to resolve the original or the refined questions of $\mathscr{P}=?\mathscr{NP}$. The preceding arguments do not however discount the possible use of information counting methods, such as those initiated by Hartmanis and Stearns [3]. We have noted in Kintala [5] and Kintala and Fischer [6] that even those methods are probably not applicable.

An interesting property of the relativized $\mathscr{PL}$-hierarchy is exhibited by the following theorem which shows that problems concerning this hierarchy do not automatically reduce to equivalent problems in $\mathscr{P}=?\mathscr{NP}$.

THEOREM 3.4. *For every* $k\geqq 2$, *there is an oracle* D_k *such that*

$$\mathscr{P}^{D_k}=\mathscr{PL}_1^{D_k} \subsetneqq \mathscr{PL}_2^{D_k} \subsetneqq \cdots \subsetneqq \mathscr{PL}_k^{D_k}=\cdots=\mathscr{PL}^{D_k}=\mathscr{NP}^{D_k}.$$

Proof. For any given m, and any oracle X, define

$$\hat{L}_m(X) = \{w|(\exists y)[|y| = (\log(|w|))^m; |wy| \text{ is even}; wy \in X]\}.$$

Note that this definition adds only the restriction "$|wy|$ is even" to the definition for $L_k(X)$. Thus, it is immediate that $\hat{L}_m(X) \in \mathcal{PL}_m^X$. We shall construct here the required oracle D_k in such a way that:

(1) For all d and for any given x such that $|x| \geqq d$ and $p_d(|x|) < |x|^{(\log|x|)^{1/k}}$, $M_d^{D_k}$ accepts x is and only if for $y = 0^d 1x10^t$, such that $|y| = p_d(|x|)+1$, $(\exists w)[|w| = (\log|x|)^k + \varepsilon$, $\varepsilon = 0$ or 1, $|yw|$ is odd, and $yw \in D_k]$.

(2)$\hat{L}_m(D_k) \notin \mathcal{PL}_{m-1}^{D_k}$ for every m such that $2 \leqq m \leqq k$.

Requirement (1) will guarantee $\mathcal{PL}_k^{D_k} = \mathcal{NP}^{D_k}$ as follows: Given any language $L \in \mathcal{NP}^{D_k}$, there is an index d such that L is accepted by $M_d^{D_k}$. We can then construct a machine $M = ML_{j,k}^{D_k}$ for some j which, given x, constructs $y = 0^d 1x10^t$ if $|x| \geqq d$ and $2|x|+2 < p_d(|x|) < |x|^{(\log|x|)^{1/k}}$ so that $|y| = p_d(|x|)+1$. This is possible except perhaps for a finite number of x. M then guesses a string w such that $|w| = (\log|x|)^k + \varepsilon$, where $\varepsilon = 0$ or 1, so that $|yw|$ is odd. M then queries D_k and accepts x if and only if $yw \in D_k$. If w is such that the above conditions on $|x|$ are not satisfied, then M accepts x if and only if it is in a finite table stored in M. Thus, information about the acceptance of strings in $\mathcal{NP}^{D_k}$ is encoded into strings in D_k of odd lengths.

Requirement (2) is satisfied by "attacking" the classes $\mathcal{PL}_1^{D_k}, \cdots, \mathcal{PL}_{k-1}^{D_k}$ in a cyclic manner through some strings in D_k of even length. This will imply $\mathcal{PL}_{m-1}^{D_k} \subsetneq \mathcal{PL}_m^{D_k}$ for $2 \leqq m \leqq k$.

We will say that a string $y = 0^d 1x10^t$ is *admissible* if $|x| \geqq d$ and $|y| \leqq x|^{(\log|x|)1/k}$. As before, D_k will be constructed in stages. Let $D_k(i)$ denote the set of strings added to D_k prior to stage i. During the course of construction, some strings will be *reserved* for $\bar{D}_k$. An index $e = \langle j, m\rangle$ (for some canonical enumeration of the pairs of the form $\langle j, m\rangle$ such that $j \geqq 0$ and $2 \leqq m \leqq k$) will be *canceled* at some stage n_e when we ensure that $ML_{j,m-1}^{D_k}$ does not accept $\hat{L}_m(D_k)$. Set $D_k(0) = \varnothing$, $n_{-1} = 0$, and start at stage 0.

Stage i: Execute the following two routines.

Routine A (Towards requirement (1)). For every string y of length i, if $y = 0^d 1x10^t$ is admissible, simulate all the computations of $M_d^{D_k(i)}$ on input x for up to $i-1$ steps. In any such computation only strings of length less than i are queried.

For each such y and the associated d and x, if any computation of $M_d^{D_k(1)}$ accepts x, then place some string of the form yw into D_k where

(i) $w \in \{0, 1\}^*$; $|w| = (\log|x|)^k + \varepsilon$ where $\varepsilon = 0$ or 1 so that $|yw|$ is odd and

(ii) yw has not been reserved for $\bar{D}_k$ in an earlier stage.

(We will ensure in condition (iv) of the following routine, which is the only routine reserving strings for $\bar{D}_k$, that such w is available, if needed. It is of course possible that no strings will be added to D_k at this stage by Routine A.)

Routine B (Towards requirement (2)). Let $e = \langle j, m\rangle$ be the least uncanceled index. If i is such that $i + (\log i)^m$ is odd, or if any of the following four conditions is not satisfied, then skip this routine and go to stage $i+1$. Otherwise, cancel e at this stage and choose $n_e = i$ as follows: suppose $e-1 = \langle g, h\rangle$ for some $g \geqq 0$ and $2 \leqq h \leqq k$.

(i) $i > p_g(n_{e-1})$;

(ii) no string of length $\geqq i$ is reserved for $\bar{D}_k$,

(iii) $c_j^{(\log i)^{m-1}} \cdot p_j(i) < 2^{(\log i)^m}$(Recall Definition 2.2);

(iv) there is no admissible $y = 0^d 1x10^t$ with $|y| \geqq i$ such that there is an $\varepsilon = 0$ or 1 so that $ML_{j,m-1}^{D_k(i)}$, in all its possible computations on input 0^i, queries *all* strings of the form yw where $w \in \{0, 1\}^*$ and $|w| = (\log|x|)^k + \varepsilon$.

Simulate the machine $M = ML_{j,m-1}^{D'}$, where $D' = D_k(i) \cup$ {the odd length strings you just added by Routine A in this stage}, on input $z = 0^i$. In this simulation, reserve for $\bar{D}_k$ all strings of length at least i queried during any possible computation of M on z and which are not members of D'. If M accepts z then add no element to D_k in this routine at this stage But, if M rejects z, then add to D_k some string of the form zv such that $|v| = (\log i)^m$ and zv is not queried during any possible computation of M on z. Such a string exists because of conditions (i) and (iii) of this routine. Cancel the index e and go to stage $i+1$.

End of stage i.

It is easy to see that for a given $e = \langle j, m\rangle$, conditions (i), (ii) and (iii) of Routine B will be satisfied for large enough i. Condition (iv) will also be satisfied because of the following:

For large enough i and for any admissible $y = 0^d 1x10^t$ such that

$$i \leqq |y| \leqq |x|^{(\log|x|)^{1/k}} = 2^{(\log|x|)^{(k+1)/k}})$$

$ML_{j,m-1}^{D_k(i)}$, will reserve at most

$$\begin{aligned} c^{(\log i)^{m-1}} \cdot p_j(i) &\leqq c^{(\log(2^{(\log|x|)^{(k+1)/k}}))^{m-1}} \cdot p_j(2^{(\log|x|)^{(k+1)/k}}) \\ &\leqq c_j^{(\log|x|)^{(k+1)(m-1)/k}} \cdot p_j(2^{(\log|x|)^{(k+1)/k}}) \\ &< 2^{(\log|x|)^k} \quad \text{for large enough x} \end{aligned}$$

since $(k+1)(m-1)/k < k$ if $2 \leqq m \leqq k$ and $k \geqq 2$.

Hence, every index e is eventually canceled, thus satisfying requirement (2). It is straightforward to see that Routine A works for requirement (1).

Our construction given above is based on Robertson's [8] observations that $(\exists X)[\mathcal{P}^X \subsetneqq \mathcal{P}_n^X = \mathcal{NP}^X]$. His proof, in turn, is similar to that of Theorem 4.5 in the next section.

4. Closure under complementation. We do not know whether $\mathcal{PL}_k$ is closed under complementation for any $k \geqq 2$. However in the relativized case we can exhibit oracle sets for each side of the question.

THEOREM 4.1. *For each $k \geqq 2$, there is an oracle E_k such that $\mathcal{PL}_k^{E_k}$ is not closed under complementation.*

Proof. The proof is by a construction very similar to that of Theorem 3.2. □

It is obvious that if $\mathcal{P}^A = \mathcal{NP}^A$, then $\mathcal{PL}_k^A$ is closed under complementation for all k, since $\mathcal{P}^X$ is closed under complementation for all oracles X. Thus, the closure of a particular $\mathcal{PL}_k^X$ under complementation depends on the oracle X. However, we can exhibit an oracle F_k such that $\mathcal{PL}_k^{F_k}$ *is closed* under complementation but $\mathcal{P}^{F_k} \neq \mathcal{PL}_k^{F_k}$.

LEMMA 4.2 (Constant speed-up). *For any time-constructible $g(n)$ and any oracle X, $\mathcal{P}_{g(n)}^X = \mathcal{P}_{g(n)/2}^X$.*

Proof. Clearly, $\mathcal{P}_{g(n)/2}^X \subseteq \mathcal{P}_{g(n)}^X$. Suppose $L \in \mathcal{P}_{g(n)}^X$. Let $M = M_{i,g(n)}^X$ be a machine accepting L and making $g(n)$ nondeterministic moves of fan-out $c = c_i$. Then construct a new machine M' which initially makes $g(n)/2$ c^2-ary nondeterministic moves to write a string of length $g(n)/2$ on a special guess tape using a special alphabet $\{\gamma_{ij} | 1 \leqq i \leqq c; 1 \leqq j \leqq c\}$. Then M' simulates M; whenever M needs to make a nondeterministic move, the nondeterminism in coded form is available on this guess tape with two c-ary nondeterministic moves available on each square of this tape. Thus $L \in \mathcal{P}_{g(n)/2}^X$. □

We shall use the following languages in the construction of F_k. Observe that if a language A is $\mathcal{C}$-complete for a class $\mathcal{C}$, then $\bar{A}$ is (co-$\mathcal{C}$)-complete where co-$\mathcal{C}$ is the class of complements of the languages in $\mathcal{C}$.

DEFINITION 4.3. For every $k \geqq 1$, and any oracle X, let $A_k(X) = \{0^i 1x10^t |$ some computation of $ML_{i,k}^X$ accepts x in no more than t steps}.

LEMMA 4.4. *$A_k(X)$ is $\mathcal{PL}_k^X$-complete. Moreover, $\mathcal{PL}_k^X$ is closed under complementation if and only if $\overline{A_k(X)} \in \mathcal{PL}_k^X$.*

Proof. Clearly, $A_k(X) \in \mathcal{PL}_k^X$. Suppose $L \in \mathcal{PL}_k^X$, say L is accepted by $ML_{d,k}^X$ for some d. Set $f(x) = 0^d 1x10^{p_d(|x|)}$. Then $f(x)$ is computable in polynomial time. Now $x \in L$ if and only if $ML_{d,k}^X$ accepts x in no more than $p_d(|x|)$ steps making no more than $(\log |x|)^k$ nondeterministic moves. So $x \in L$ if and only if $f(x) \in A_k(X)$. Hence $A_k(X)$ is $\mathcal{PL}_k^X$-complete.

If $\mathcal{PL}_k^X$ is closed under complementation then obviously $\overline{A_k(X)} \in \mathcal{PL}_k^X$. Conversely, suppose $\overline{A_k(X)} \in \mathcal{PL}_k^X$. Let $\overline{A_k(X)}$ be accepted by a machine $M = ML_{j,k}^X$ for some j. For any $L \in \mathcal{PL}_k^X$, since L is polynomial-time reducible to $A_k(X)$, there exists a function f such that $x \in \bar{L}$ if and only if $f(x) \in \overline{A_k(X)}$ and $|f(x)| \leqq |x|^b$ for some constant b. Construct a machine N to accept $\bar{L}$ in the following manner. Given a string x, N transforms x to $f(x)$ and then simulates M on $f(x)$. Obviously the number of nondeterministic moves made by N is bounded by $(\log |f(x)|)^k \leqq b^k \cdot (\log |x|)^k$. Invoking Lemma 4.2, we can now infer that $\bar{L} \in \mathcal{PL}_k^X$. □

THEOREM 4.5. *For every $k \geqq 2$, there is an oracle F_k such that*

(1) *$\mathcal{PL}_k^{F_k}$ is closed under complementation and*

(2) *$\mathcal{P}^{F_k} \subsetneq \mathcal{PL}_k^{F_k}$.*

Proof. For the given k and any oracle X, define

$$L'_k(X) = \{w \,|\, |w| = 2^{m+1} \text{ for some } m \geqq 0;\ (\exists y)[|y| = m^k;\ wy \in X]\}.$$

Clearly, $L'_k(X) \in \mathcal{PL}_k^X$. F_k will be constructed in such a way that

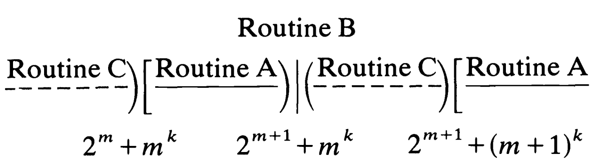
(1) for any string u, $u \in \overline{A_k(F_k)}$ if and only if $(\exists v)[|v| = (\log |u|)^k$ *and* $uv \in F_k]$

(2) $L'_k(F_k) \notin \mathcal{P}^{F_k}$.

Then $\overline{A_k(F_k)} \in \mathcal{PL}_k^{F_k}$ from (1), so that by Lemma 4.4 $\mathcal{PL}_k^{F_k}$ will be closed under complementation; and since $L'_k(F_k) \in \mathcal{PL}_k^{F_k}$ we can infer from (2) that $\mathcal{P}^{F_k} \neq \mathcal{PL}_k^{F_k}$.

As usual, F_k will be constructed in stages. At stage i, we decide the membership in F_k of all strings of length i. In the course of construction, some strings will be reserved for $\bar{F}_k$, that is, designated as nonmembers of F_k. An index e will be *canceled* at some stage when we ensure that $M_{e,0}^{F_k}$ (i.e., the eth machine of the class of deterministic polynomial-bounded query machines) does not recognize $L'_k(F_k)$. Let $F_k(i)$ denote the set of strings placed into F_k prior to stage i; $n_{-1} = 0$, $F_k(0) = \varnothing$; and start at stage 0.

Stage i: If $i = 2^m + m^k + j$ for some m, j such that $0 \leqq j < 2^m$, then go to Routine A. If $i = 2^{m+1} + m^k$ for some $m \geqq 0$ then go to Routine B. Else go to Routine C. Observe that for every integer $n \geqq 1$, $n = 2^m + j$ for m and j such that $0 \leqq j < 2^m$. The following "integer line," for sufficiently large m, might be helpful to the reader in determining which routines are executed at various stages of the construction of F_k.



Routine A. Here $i = 2^m + m^k + j$ for some $m \geqq 0$ and $0 \leqq j < 2^m$.

Notice that $\log (2^m + j) = m$. For every string z of length i, not reserved for $\bar{F}_k$ at an earlier stage, determine the prefix u of z having length $2^m + j$. If $u = 0^d 1x10^t$ then place z into F_k if and only if $ML_{d,k}^{F_k(i)}$ does not accept x in fewer than t steps making no more than $(\log |x|)^k$ nondeterministic moves. If u is not of the above form, then place z into F_k. Go to Stage $i+1$.

Routine B. Here $i = 2^{m+1} + m^k$ for some $m \geqq 0$.

Let e be the least uncanceled index. Choose $n_e = 2^{m+1}$ if and only if the following conditions are satisfied:

(i) No string of length $\geqq i$ is reserved for $\bar{F}_k$;

(ii) $n_e > p_{e-1}(n_{e-1})$;

(iii) $p_e(n_e) < 2^{(\log(n_e)-1)^k}$.

If 2^{m+1} does not satisfy the above conditions then go to Routine C. Otherwise, simulate the machine $M = M_{e,\mathfrak{d}}^{F_k(i)}$ on input $x = 0^{n_e}$ and reserve for $\bar{F}_k$ all strings of length $\geqq i$ queried during the computation of M on x. If M accepts x then add no element to F_k at this stage. But if M rejects x then add to F_k some string of the form xv such that $|v| = m^k$. Thus $|xv| = i$. Such a string exists because M is deterministic and hence could have queried at most $p_e(n_e) < 2^{(\log(n_e)-1)^k} = 2^{m^k}$ strings in its computation on x. Cancel index e and go to stage $i+1$.

Routine C. Add no element to F_k at this stage and go to stage $i+1$.

End Stage i.

Every index e is eventually canceled because for a given e and some sufficiently large m conditions (i), (ii), (iii) will be satisfied after index $(e-1)$ is canceled. When index e is canceled at stage i, our construction guarantees that $M_{e,\mathfrak{d}}^{F_k}$ does not recognize $L'_k(F_k)$, satisfying requirement (2).

At any stage i of the form $(2^{m+1} + m^k)$, fewer than 2^{m^k} strings are reserved for $\bar{F}_k$. Thus, for any j such that $0 \leqq j < 2^m$, fewer than $2^{0^k} + 2^{1^k} + \cdots + 2^{(m-1)^k} < 2^{m^k}$ strings of length $2^m + m^k + j$ are reserved for $\bar{F}_k$ before stages of the form $i = 2^m + m^k + j$. Therefore, every string u of length $2^m + j$ is the prefix of at least one string z of length $(2^m + j) + m^k$ which is never reserved for $\bar{F}_k$. Moreover the computations in Routine A when processing $u = 0^d 1 x 1 0^t$ at stage $2^m + j + m^k$ never query strings of length greater than $2^m + j$. The memberships of the strings of length not exceeding $2^m + j$ in F_k are determined at earlier stages. Hence, by construction, any string u of length $2^m + j$ is in $\overline{A_k(F_k)}$ if and only if u is the prefix of a string of length $2^m + j + m^k$ in F_k. Therefore, $\overline{A_k(F_k)} \in \mathcal{PL}_k^{F_k}$, satisfying requirement (1). □

Acknowledgment. The authors would like to thank Professors Edward Robertson and Joel Seiferas for many helpful discussions. They are indebted to Professor Seiferas for pointing out a serious mistake in an earlier version of Theorem 3.4. They are also grateful to the referees for their careful reading of the paper and their helpful comments and suggestions.

REFERENCES

[1] I. Baker, J. Gill and R. Solovay, *Relativization of the $P = ?$ NP question*, SIAM J. Comput., 4 (1975), pp. 431–442.

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

[3] J. Hartmanis and R. E. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[4] R. M. Karp, *Reducibilities among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds. Plenum Press, New York-London, 1972, pp. 85–104.

[5] C. M. R. Kintala, *Computations with a restricted number of nondeterministic steps*, Ph.D. dissertation, Pennsylvania State University, University Park, 1977.

[6] C. M. R. Kintala and P. C. Fischer, *Computations with a restricted number of nondeterministic steps*, Proc. 9th Annual Symp. on Theory of Computing (1977), pp. 178–185.

[7] V. Pratt, *Every prime has a succinct certificate*, SIAM J. Comput., 4 (1975), pp. 214–220.

[8] E. L. Robertson, private communication, 1977.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0004 $01.00/0

ON THE COMPLEXITY OF COMPOSITION AND GENERALIZED COMPOSITION OF POWER SERIES*

R. P. BRENT† AND J. F. TRAUB‡

Abstract. Let $F(x)=f_1x+f_2x^2+\cdots$ be a formal power series over a field Δ. Let $F^{[0]}(x)=x$ and for $q=1,2,\cdots$, define $F^{[q]}(x)=F^{[q-1]}(F(x))$. The obvious algorithm for computing the first n terms of $F^{[q]}(x)$ is by the composition analogue of repeated squaring. This algorithm has complexity about $\log_2 q$ times that of a single composition. Brent showed that the factor $\log_2 q$ can be eliminated in the computation of the first n terms of $(F(x))^q$ by a change of representation, using the logarithm and exponential functions. We show the factor $\log_2 q$ can also be eliminated for the composition problem, unless the complexity of composition is quasi-linear.

$F^{[q]}(x)$ can often, but not always, be defined for more general q. We give algorithms and complexity bounds for computing the first n terms of $F^{[q]}(x)$ whenever it is defined.

We conclude the paper with some open problems.

Key words. composition, fast algorithms, formal power series, symbolic computation, generalized composition, functional equations, Schroeder function, iteration, similarity transformations

1. Introduction. Let

$$F(x)=f_1x+f_2x^2+\cdots \tag{1.1}$$

be a formal power series over a field Δ. Let $F^{[0]}(x)=x$ and for $q=1,2,\cdots$, define the *q-composite* of F by

$$F^{[q]}(x)=F^{[q-1]}(F(x)). \tag{1.2}$$

The q-composite may also be called the q-iterate. Let $H(x)$ be the reversion of $F(x)$, i.e., the power series inverse to $F(x)$ under composition. For $q=1,2,\cdots$, define

$$F^{[-q]}(x)=H^{[q]}(x). \tag{1.3}$$

As we shall see below, the q-composite of F can often (but not always) be defined for more general q. If q is not an integer, we shall call $F^{[q]}(x)$ a *generalized q-composite.* We confine ourselves to the case that $F^{[q]}(x)$ is a power series. One important special case of generalized composition is $q=1/r$, where r is an integer. Then $G=F^{[1/r]}(x)$ is an rth root of F under composition, and satisfies the equation $G^{[r]}(x)=F(x)$.

Let

$$F_n(x)=f_1x+\cdots+f_nx^n, \tag{1.4}$$

$$G(x)=F^{[q]}(x)=g_1x+g_2x^2+\cdots, \tag{1.5}$$

$$G_n(x)=g_1x\cdots+g_nx^n. \tag{1.6}$$

Given q and $F_n(x)$, we want to compute $G_n(x)$.

In this paper we shall give algorithms and complexity bounds for computing $G_n(x)$ whenever it is defined. For integer q these algorithms are asymptotically faster than the obvious algorithms.

* Received by the editors May 23, 1978, and in revised form December 18, 1978. This research was supported in part by the National Science Foundation under Grant MCS75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422. The work of the first author was also supported in part by the National Science Foundation under Grant 1-442427-21164-2 at the University of California at Berkeley.

† Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania. Presently at Computer Science Department, Australian National University, Canberra, Australia.

‡ Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213.

We discuss the last point. Let $\mathrm{COMP}_1(n)$ denote the complexity of computing the first n terms of $F(F(x))$, and let q be a power of two. Then the obvious algorithm for computing $G_n(x)$ is by the composition analogue of "repeated squaring," and has complexity $\mathrm{COMP}_1(n) \lg q$. (We shall denote $\log_2$ by lg.) Can we eliminate the multiplicative factor of $\lg q$?

An analogous problem is that of computing $R_n(x)$, the first n terms of $(F(x))^q$. Asymptotically in n, the complexity of forming $R_n(x)$ is the same as the complexity of a single multiplication of two polynomials of degree n. This follows from the observation that if $A(x)$ is a power series with constant term unity, then $(A(x))^q \equiv \exp(q \ln A(x))$. This may be viewed as a change of representation of $A(x)$ to a new representation where multiplication is replaced by addition, followed by the inverse change of representation. Brent (1976) showed that the change of representation could be computed "fast."

This suggests asking whether there is a change of representation which reduces composition to multiplication. We shall see that there is, at least in the "regular" case (see § 3). Furthermore, the change of representation can be computed "fast." *This enables us to eliminate the multiplicative factor of* $\lg q$ (unless the complexity of composition is quasi-linear). In addition we shall show (§§ 4–6) that even in the "nonregular" cases we can still eliminate this factor. A bonus is that our algorithms apply for non-integer q (so long as $F^{[q]}(x)$ is a well-defined power series).

The problem of composition and generalized composition occurs in many applications including branching processes, asymptotic analysis, difference equations, numerical analysis, and dynamical systems. See, for example, Aczél (1966), Cherry (1964), de Bruijn (1970), Feller (1957), Harris (1963), Henrici (1974), Knuth (1969), Kuczma (1968), Levy and Lessman (1961), and Melzak (1973). The study of composition (often called iteration) may be viewed as a major subfield of mathematics. See Aczél (1966), Gross (1972), and Kuczma (1968) for very extensive bibliographies. However, little attention seems to have been given to the development of algorithms for computing $F^{[q]}(x)$ when $F(x)$ is a given power series.

The following conventions are adopted below. We deal with formal power series; that is, we do not concern ourselves with convergence. Power series are denoted by upper case letters such as $A(x)$ or simply A, with coefficients denoted by the corresponding lower case letters such as a_i. If $A(x) = a_k x^k + a_{k+1}x^{k+1} + \cdots$, $a_k \neq 0$, then $\mathrm{ord}(A) = k$. It is convenient to define $\mathrm{ord}(0) = \infty$. If $\mathrm{ord}(B - C) \geqq k$ we write $B = C + O(x^k)$. The polynomial $b_0 + b_1 x + \cdots + b_{k-1}x^{k-1}$ is denoted either by $B(x) \bmod x^k$ or by $B_{k-1}(x)$. It is convenient to define $\gamma(n, q) = O(\delta(n, q))$ to mean $|\gamma(n, q)| \leqq K|\delta(n, q)|$ for all sufficiently large integer n for all q under consideration.

We summarize the remainder of the paper. Our complexity model is specified in § 2. In § 3 we study the "regular" case when the multiplier f_1 is such that $f_1 \neq 0$, $f_1^m \neq 1$, $m = 1, 2, \cdots$. In the following three sections we consider the cases $f_1 = 0$; $f_1 = 1$; $f_1^m = 1$, integer $m > 1$, but $f_1 \neq 1$, respectively.

In each of §§ 3, 4, and 5 we define an "auxiliary" function, demonstrate it can be computed fast by "divide and conquer," and show how it can be used to compute $F^{[q]}$. The case studied in § 6 can be reduced to that of § 5. In the concluding section we state a theorem (Theorem 7.1) summarizing our results, state the defining equations for all cases, and mention some open problems.

2. Complexity model. In this section we state our complexity model and summarize the complexity results needed below. We assume that scalar arithmetic operations are performed exactly and have unit cost. Thus our time bounds are invalid if, for example, exact rational arithmetic is used. However, our algorithms should still be useful in this case.

Given power series $A(x)$ and $B(x)$, the time required to compute $A(x)B(x) \bmod x^n$ is denoted by MULT (n). If ord $(B) \geqq 1$, the time required to compute $A(B(x)) \bmod x^n$ is denoted by COMP (n). We assume that MULT (n) and COMP (n) satisfy certain plausible regularity conditions (see Brent and Kung (1978, § 1)). Then Brent and Kung (1978) show

$$\text{(2.1)} \qquad \text{COMP}\,(n) = O(\min\,(n^{(1+r)/2}, (n \lg n)^{1/2}\,\text{MULT}\,(n))),$$

if matrix multiplication has complexity $O(n^r)$. If the field Δ is such that fast algorithms like the FFT are available, then

$$\text{(2.2)} \qquad \text{MULT}\,(n) = O(n \lg n)$$

(see Borodin and Munro (1975)), and it follows from (2.1) that

$$\text{(2.3)} \qquad \text{COMP}\,(n) = O((n \lg n)^{3/2}).$$

The bounds in this paper will be expressed in terms of the complexity function

$$\text{(2.4)} \qquad \text{COMP}_2\,(n) = \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j\,\text{COMP}\,(\lceil 2^{-j} n \rceil).$$

Assume (with notation as in Knuth (1976))

$$\text{(2.5)} \qquad \text{COMP}\,(n) = \Theta(n^\alpha s(n)),$$

where $\alpha \geqq 1$, and $s(n)$ is a monotonic increasing positive function. (For example, $s(n)$ might be $(\lg n)^\beta$ for some constant $\beta \geqq 0$.) Then, $s(n) = O(n^\varepsilon)$ for all $\varepsilon > 0$.

$$\text{(2.6)} \qquad \text{COMP}_2\,(n) = \begin{cases} O(\text{COMP}\,(n)), & \text{if } \alpha > 1, \\ O(\text{COMP}\,(n) \lg n), & \text{if } \alpha = 1. \end{cases}$$

If the field Δ is such that (2.3) holds, then $\alpha \leqq \frac{3}{2}$. If $\alpha > 1$, then COMP$_2$ (n) may be replaced by O(COMP (n)) in our bounds.

If $\alpha = 1$, we say COMP (n) is quasi-linear. If $\alpha = 1$ and q is a fixed integer, then "repeated squaring" is asymptotically faster than our algorithms. Of course, if q is not an integer, then "repeated squaring" is not an alternative to our algorithms. If $\alpha > 1$, our result (that we can eliminate the multiplicative factor of $\lg q$) holds for all fields of characteristic zero and all finite fields of characteristic p greater than n.

If f_1^q is defined, we denote the complexity of computing f_1^q by POWER (q). If q is a positive integer, then POWER $(q) = O(\lg q)$. To eliminate POWER (q) terms we sometimes assume that f_1^q is given.

In Brent (1976) it is shown that the complexity of computing $\ln(1 + A(x)) \bmod x^n$ is O(MULT (n)) for any power series A, ord $(A) > 0$. Using Brent's results it can be shown that the complexity of computing $(B(x))^q \bmod x^n$ is O(MULT (n)+ POWER (q)). By Brent and Kung (1978, Lemma 4.2) MULT $(n) = O$(COMP (n)), so we can absorb MULT (n) into COMP (n) in our analyses.

Recall that COMP$_1$ (n) was defined as the complexity of computing the first n terms of $F(F(x))$. It can be shown, by means similar to the proof of Brent and Kung (1978) that the complexity of reversion and composition are asymptotically equal, that COMP $(n) = O$(COMP$_1$ (n)).

3. The regular case. In this section we study the computation of $F^{[q]}(x)$ when $f_1 \neq 0$, $f_1^m \neq 1$, $m = 1, 2, \cdots$. We call this the *regular* case. Define the Schroeder function $S(x)$ by

$$S(F(x))=f_1S(x) \qquad \operatorname{ord}(S)=1, \qquad s_1=1. \tag{3.1}$$

$S(x)$ exists and is unique (Schroeder (1871), Kuczma (1968, Chap. 6)). See also Parker (1977). It is easy to prove that, for all integer q,

$$F^{[q]}(x)=S^{[-1]}(f_1^qS(x)). \tag{3.2}$$

$S(x)$ and $S^{[-1]}(x)$ play the role that the logarithm and exponential functions play in computing $(F(x))^q$ fast. They reduce self-composition to scalar powering.

Equation (3.1) has an interesting matrix interpretation. To the formal power series $F(x)$ and $S(x)$, we may associate infinite matrices M_F and M_S, respectively (see, for example, Henrici (1974, p. 45)). Then $S^{[-1]}(x)$ is associated with M_S^{-1}. It is easy to show that (3.1) corresponds to a matrix similarity transformation which transforms M_F to a diagonal matrix with diagonal elements f_1^k, $k=1,2,\cdots$. The conditions $f_1\neq 0$, $f_1^m\neq 1$ ensure that the eigenvalues of M_F are all distinct.

If q is not an integer but q and the scalar f_1 are such that f_1^q is defined, then (3.2) may be used to define $F^{[q]}$. We shall use the "divide and conquer" strategy to compute $S(x)$ fast and then show how to compute $F^{[q]}$ from (3.2) in total time $O(\text{COMP}_2\,(n)+\text{POWER}\,(q))$.

Although we wish to solve the functional equation (3.1), to make the "divide and conquer" strategy work we embed (3.1) in the more general linear functional equation

$$A(x)W(F(x))-B(x)W(x)-C(x)=0, \tag{3.3}$$

where W is the unknown. Note that this equation includes reversion as a special case. The "divide and conquer" algorithm introduced to solve (3.3) may therefore be used to revert power series. This algorithm is different from the one derived by Newton iteration and given in Brent and Kung (1978).

Lemma 3.1 gives the basis for a "divide and conquer" algorithm for solving (3.3). The proof is by substitution. Lemma 3.2 gives sufficient conditions for the existence of a formal solution, and Lemma 3.3 gives an upper bound on the time required to compute an approximate solution.

LEMMA 3.1. *If n, p are nonnegative integers,* $\operatorname{ord}(F)\geqq 1$,

$$A(x)U(F(x))-B(x)U(x)-C(x)=x^nR(x) \tag{3.4}$$

and

$$A(x)(F(x)/x)^nV(F(x))-B(x)V(x)+R(x)=O(x^p), \tag{3.5}$$

then

$$A(x)W(F(x))-B(x)W(x)-C(x)=O(x^{n+p}) \tag{3.6}$$

where

$$W(x)=U(x)+x^nV(x). \tag{3.7}$$

Remark 3.1. If Lemma 3.1 is applied for $n=p=2^j$, $j=0,1,2,\cdots$, we have an algorithm for approximating $W(x)$ which is quadratically convergent in the sense of Kung and Traub (1978).

LEMMA 3.2. *If* $\operatorname{ord}(F)\geqq 1$,

$$a_0f_1^m\neq b_0 \quad \textit{for } m=1,2,3,\cdots \tag{3.8}$$

and

$$a_0 = b_0 \quad \textit{implies} \quad c_0 = 0 \tag{3.9}$$

then there is a formal power series W, *satisfying* (3.3), *with* ord $(W) = 0$ *unless* $c_0 = 0$ *and* $a_0 \neq b_0$.

Proof. We shall construct $w_0, w_1, \cdots$ such that $W(x) = \sum_{j=0}^{\infty} w_j x^j$ satisfies (3.3). We let

$$W_m(x) = \sum_{j=0}^{m} w_j x^j \tag{3.10}$$

and show by induction on m that, for some power series $R_{m+1}(x)$,

$$A(x)W_m(F(x)) - B(x)W_m(x) - C(x) = x^{m+1}R_{m+1}(x) = O(x^{m+1}). \tag{3.11}$$

Let

$$w_0 = \begin{cases} 1 & \text{if } a_0 = b_0, \\ c_0/(a_0 - b_0) & \text{otherwise.} \end{cases} \tag{3.12}$$

Then (3.11) holds for $m = 0$, starting the induction. Assuming that (3.11) holds for $m \geqq 0$, we define

$$w_{m+1} = \frac{R_{m+1}(0)}{b_0 - a_0 f_1^{m+1}} \tag{3.13}$$

and apply Lemma 3.1 (with $n = m + 1$, $p = 1$, $U = W_m$, $V = w_{m+1}$) to deduce that (3.11) holds with m replaced by $m + 1$. Thus, the result follows by induction on m. □

LEMMA 3.3. *Suppose that* $w_0, \cdots, w_{n-1}$ *can be found in time* $t(n)$ *whenever the conditions of Lemma* 3.2 *apply. Then*

$$t(2n) \leqq 2t(n) + \text{COMP}(2n) + O(\text{MULT}(n)). \tag{3.14}$$

Proof. In time $t(n)$ we find $u_0, \cdots, u_{n-1}$ such that (3.4) holds for some power series $R(x)$, where $U(x) = \sum_{j=0}^{n-1} u_j x^j$. Compute $U(F(x)) \bmod x^{2n}$ in time COMP $(2n)$, and then find

$$R(x) = \frac{A(x)U(F(x)) - B(x)U(x) - C(x)}{x^n} \bmod x^n \tag{3.15}$$

in time $O(\text{MULT}(n))$. [Note: MULT $(2n) = O(\text{MULT}(n))$.]

Since ord $(F) \geqq 1$, $F(x)/x$ is a power series, and by an algorithm given in Brent (1976) we can compute $(F(x)/x)^n \bmod x^n$, and thus

$$\tilde{A}(x) = A(x)(F(x)/x)^n \bmod x^n, \tag{3.16}$$

in time $O(\text{MULT}(n))$. Now (3.5) with $p = n$ is just

$$\tilde{A}(x)V(F(x)) - B(x)V(x) + R(x) = O(x^n),$$

so we can find $v_0, \cdots, v_{n-1}$ in time $t(n)$. Using Lemma 3.1, we take

$$w_j = \begin{cases} u_j & \text{if } 0 \leqq j < n, \\ v_{j-n} & \text{if } n \leqq j < 2n, \end{cases}$$

and the result follows. □

COROLLARY 3.1. *With the notation of Lemma* 3.3,

$$t(n) = O(\text{COMP}_2(n)). \tag{3.17}$$

Proof. This follows from Lemma 3.3, the definition of $\mathrm{COMP}_2(n)$, and the fact that $\mathrm{MULT}(n) = O(\mathrm{COMP}(n))$. □

COROLLARY 3.2. *If* $\mathrm{ord}(F) = 1$ *and* $f_1^m \neq 1$ *for* $m = 1, 2, \cdots$, *then we can compute the first n coefficients,* $s_0, \cdots, s_{n-1}$ *of the Schroeder function* $S(x)$ *satisfying* (3.1) *in time* $O(\mathrm{COMP}_2(n))$.

Proof. We solve a special case of (3.3), namely

$$(F(x)/x)W(F(x)) - f_1W(x) = 0, \tag{3.18}$$

to obtain $w_0, \cdots, w_{n-2}$ by the method of Lemma 3.2. Then $S(x) = xW(x)$ satisfies (3.1) mod x^n, so $s_0 = 0$ and $s_j = w_{j-1}$ for $j = 1, \cdots, n-1$. □

THEOREM 3.1. *Assume* $\mathrm{ord}(F) = 1$, $f_1^m \neq 1$ *for* $m = 1, 2, \cdots$. *Let* f_1^q *be defined and let*

$$G(x) = F^{[q]}(x). \tag{3.19}$$

Then $g_0, \cdots, g_{n-1}$ *can be computed in time*

$$O(\mathrm{COMP}_2(n) + \mathrm{POWER}(q)). \tag{3.20}$$

Proof. Using the method of Corollary 3.2, we compute $S_{n-1}(x) = \sum_{j=1}^{n-1} s_jx^j$ such that $s_1 \neq 0$ and

$$S_{n-1}(F(x)) = f_1S_{n-1}(x) + O(x^n) \tag{3.21}$$

in time $O(\mathrm{COMP}_2(n))$. Now

$$S_{n-1}(G(x)) = f_1^qS_{n-1}(x) + O(x^n), \tag{3.22}$$

and thus

$$G(x) = S_{n-1}^{[-1]}(f_1^qS_{n-1}(x)) + O(x^n). \tag{3.23}$$

Using the method of Brent and Kung (1978), we can compute $S_{n-1}^{[-1]}(x)$ mod x^n in time $O(\mathrm{COMP}(n))$, and the $g_0, \cdots, g_{n-1}$ are obtained from (3.23) in time $\mathrm{COMP}(n) + \mathrm{POWER}(q)$. The result follows. □

Remark 3.2. The condition $f_1^m \neq 1$ is necessary so that the divisor in (3.13) is nonzero. Thus, we need only assume that $f_1^m \neq 1$ for $m = 1, 2, \cdots, n-2$. If F is a formal power series over a finite field with characteristic p, then it is necessary to assume $n \leqq p$.

The proofs above are constructive and give the following two algorithms.

ALGORITHM 3.1. The algorithm $\mathscr{A}(A, B, C, F, W, m)$ finds $w_0, \cdots, w_{m-1}$ such that $W(x)$ satisfies (3.3). It is defined recursively by:

```
**if** $m = 1$ **then** {use equation (3.12) to define $w_o$} **else**
{$n \leftarrow \lceil m/2 \rceil$;
$\mathscr{A}(A, B, C, F, U, n)$;
Compute $R$ using equation (3.15);
Compute $\tilde{A}$ using equation (3.16);
$\mathscr{A}(\tilde{A}, B, -R, F, V, n)$;
**for** $j \leftarrow 0$ step 1 **until** $n-1$ **do** {$w_j \leftarrow u_j$; $w_{n+j} \leftarrow v_j$}}.
```

ALGORITHM 3.2. The following algorithm computes $G(x) = F^{[q]}(x)$ if the conditions of Theorem 3.1 apply:

1. Take $A(x) = F(x)/x$, $B(x) = f_1$, $C(x) = 0$ and find $w_0, \cdots, w_{n-2}$ such that $W(x)$

satisfies (3.3) by applying $\mathcal{A}(A, B, C, F, W, n-1)$ (see Algorithm 3.1).

2. Let $s_0 = 0$, $s_j = w_{j-1}$ for $j = 1, \cdots, n-1$, and compute $S^{[-1]}(f_1^q S(x)) \bmod x^n$ using the composition and reversion algorithms of Brent and Kung (1978).

4. Multiplier zero. In this section we study the case $f_1 = 0$. Since the problem is trivial if $F(x) \equiv 0$, we can assume $\operatorname{ord}(F) = k$, $1 < k < \infty$. We define auxiliary power series $S(x)$ by

$$S(F(x)) = f_k(S(x))^k, \qquad \operatorname{ord}(S) = 1, \qquad s_1 = 1. \tag{4.1}$$

This reduces to Schroeder's equation (3.1) if $k = 1$. By induction on q we have, for all positive integer q,

$$F^{[q]}(x) = S^{[-1]}\{f_k^{(k^q-1)/(k-1)}[S(x)]^{k^q}\}. \tag{4.2}$$

Remark 4.1. The restriction to positive integer q is essential here. For example, take $F = x^3$. Then $F^{[q]}$ does not exist as a power series for $q = -1$ or $q = \frac{1}{2}$.

The following lemmas reduce the solution of (4.1) to problems solved in the previous section.

LEMMA 4.1. *If* $\operatorname{ord}(F) = k > 1$ *the equation.*

$$W(F(x)) - kW(x) + \{(k-1) + \ln[F(x)/(f_k x^k)]\} = 0 \tag{4.3}$$

has a solution $W(x)$, *and* $w_0, \cdots, w_{n-1}$ *can be computed in time* $O(\mathrm{COMP}_2(n))$.

Proof. Lemmas 3.1 to 3.3 are applicable to (4.3), so $W(x)$ exists and $w_0, \cdots, w_{n-1}$ can be computed in time $O(\mathrm{COMP}_2(n))$ by the method used in the proof of Lemma 3.3. □

LEMMA 4.2. *If* $\operatorname{ord}(F) = k > 1$ *and* $W(x)$ *satisfies* (4.3), *then*

$$S(x) = x \exp(W(x) - 1) \tag{4.4}$$

satisfies (4.1).

Proof. Substitute $W(x) = 1 + \ln(S(x)/x)$ in (4.3). From (3.12), $w_0 = 1$, so $S(x)$ is a power series. □

Using the algorithm of Brent (1976) we can compute the first n coefficients of

$$[S(x)/x]^{k^q} = \exp[k^q(W(x)-1)]$$

in time $O(\mathrm{MULT}(n))$ once $w_0, \cdots, w_{n-1}$ are known. We can also compute $f_k^{(k^q-1)/(k-1)}$ in time $\mathrm{POWER}((k^q-1)/(k-1))$. Then, using a slight modification of the composition and reversion algorithms of Brent and Kung (1978) we have:

THEOREM 4.1. *Assume* $\operatorname{ord}(F) = k > 1$, $q \geqq 1$ *is a positive integer, and*

$$G(x) = F^{[q]}(x)/x^{k^q}. \tag{4.5}$$

Then $g_0, \cdots, g_{n-1}$ *can be computed in time*

$$O(\mathrm{COMP}_2(n) + \mathrm{POWER}((k^q-1)/(k-1))). \tag{4.6}$$

5. Multiplier unity. Now we consider the case that the multiplier f_1 is equal to unity. We define an auxiliary function T by

$$T(F(x)) = F'(x)T(x), \qquad \operatorname{ord}(T) = \operatorname{ord}(F(x) - x). \tag{5.1}$$

$T(x)$ exists and is unique up to a scaling factor (Kuczma (1968, Lemma 9.4)). Let $G(x) = F^{[q]}(x)$. Then we show below that $G(x)$ may be computed from the equation

$$T(G(x)) = G'(x)T(x). \tag{5.2}$$

Remark 5.1. T may also exist if $f_1 \neq 1$. If F is such that the Schroeder function S exists, then $T(x) = cS(x)/S'(x)$, where c is a nonzero constant.

Example 5.1. If $F(x) = 2x + x^2$, then $S(x) = \ln(1+x)$, $T(x) = (1+x)\ln(1+x)$, $F^{[q]}(x) = (1+x)^{2^q} - 1$. If $F(x) = x/(1-x)$, then $T(x) = x^2$.

Although we wish to solve the functional equation (5.1), as before we need to embed (5.1) in a more general equation. Throughout this section we define d by $F(x) = x + f_d x^d + \cdots$, $f_d \neq 0$, and let k be any integer greater than d. Then we shall solve

$$x^{1-d}[(F(x)/x)^k Y(F(x)) - F'(x)Y(x)] - A(x) = 0 \tag{5.3}$$

for $Y(x)$.

Lemma 5.1 gives the basis for a "divide and conquer" algorithm for solving (5.3). Lemma 5.2 gives sufficient conditions for the existence of a formal solution, and Lemma 5.3 gives an upper bound on the time required to compute an approximate solution. Lemma 5.4 establishes (5.2) and gives a sufficient condition for G to be uniquely defined.

LEMMA 5.1. *Let n, p be nonnegative integers. If*

$$x^{1-d}[(F(x)/x)^k U(F(x)) - F'(x)U(x)] - A(x) = x^n R(x) \tag{5.4}$$

and

$$x^{1-d}[(F(x)/x)^{k+n} V(F(x)) - F'(x)V(x)] + R(x) = O(x^p), \tag{5.5}$$

then

$$x^{1-d}[(F(x)/x)^k W(F(x)) - F'(x)W(x)] - A(x) = O(x^{n+p}) \tag{5.6}$$

where

$$W(x) = U(x) + x^n V(x). \tag{5.7}$$

Proof. By direct substitution. Note that since $F(x) = x + f_d x^d + \cdots$, the terms in square brackets in (5.4) to (5.6) have ord $\geqq d-1$. □

LEMMA 5.2. *There is a formal power series $Y(x)$ such that*

$$x^{1-d}[(F(x)/x)^k Y(F(x)) - F'(x)Y(x)] = A(x). \tag{5.8}$$

Proof. We shall construct $y_0, y_1, \cdots$ such that $Y(x) = \sum_{j=0}^{\infty} y_j x^j$ satisfies (5.8). Recall our assumption that $k > d = \operatorname{ord}(F(x) - x)$. Take

$$y_0 = \frac{a_0}{(k-d)f_d} \tag{5.9}$$

and let

$$Y_n(x) = \sum_{j=0}^{n} y_j x^j. \tag{5.10}$$

Thus

$$x^{1-d}[(F(x)/x)^k Y_{n-1}(F(x)) - F'(x)Y_{n-1}(x)] - A(x) = x^n R_n(x) \tag{5.11}$$

is true for $n = 1$ (where R_n is some power series). Define

$$y_n = \frac{-R_n(0)}{(k+n-d)f_d} \tag{5.12}$$

for $n \geqq 1$. Using Lemma 5.1 with $p = 1$, it is straightforward to prove that (5.11) holds for all $n \geqq 1$, by induction on n. Thus, the result follows. □

LEMMA 5.3. *Suppose that $y_0, \cdots, y_{n-1}$ can be found in time $t_2(n)$ whenever the conditions of Lemma* 5.2 *apply. Then*

$$t_2(2n) \leqq 2t_2(n) + \mathrm{COMP}\,(2n+d-1) + O(\mathrm{MULT}\,(n)). \tag{5.13}$$

Proof. In time $t_2(n)$ we find $u_0, \cdots, u_{n-1}$ such that (5.4) holds for some power series $R(x)$, if $U(x) = \sum_{j=0}^{n-1} u_j x^j$. Compute $U(F(x)) \bmod x^{2n+d-1}$ and then $R(x) \bmod x^n$ from (5.4). Then find $v_0, \cdots, v_{n-1}$ such that $V(x)$ satisfies (5.5) with $p = n$ (this takes time $t_2(n) + O(\mathrm{MULT}\,(n))$). From Lemma 5.1 we can take

$$y_j = \begin{cases} u_j & \text{if } 0 \leqq j < n, \\ v_{j-n} & \text{if } n \leqq j < 2n, \end{cases}$$

so we get $y_0, \cdots, y_{2n-1}$ in time $2t_2(n) + \mathrm{COMP}\,(2n+d-1) + O(\mathrm{MULT}\,(n))$ as required. □

COROLLARY 5.1. *With the notation of Lemma* 5.3, $t_2(n) = O(\mathrm{COMP}_2\,(n))$.

COROLLARY 5.2. *There exists a formal power series $T(x)$ such that ord $(T) = d$ and*

$$T(F(x)) = F'(x)T(x). \tag{5.14}$$

Moreover, $t_d, \cdots, t_{n-1}$ can be found in time $O(\mathrm{COMP}_2\,(n))$.

Proof. If

$$A(x) = x^{-2d}[F'(x)x^d - (F(x))^d] = (f_{d+1} - f_2^2) + \cdots \tag{5.15}$$

and

$$x^{1-d}[(F(x)/x)^{d+1} Y(F(x)) - F'(x)Y(x)] = A(x) \tag{5.16}$$

then

$$T(x) = x^d + x^{d+1}Y(x) \tag{5.17}$$

satisfies (5.14). Thus, the result follows from Lemma 5.2 and Corollary 5.1. □

LEMMA 5.4. *Let q be an integer, T satisfy* (5.14), *and*

$$G(x) = F^{[q]}(x). \tag{5.18}$$

Then

$$T(G(x)) = G'(x)T(x), \tag{5.19}$$

and the power series $G(x)$ is uniquely determined by (5.19) *and the condition*

$$\mathrm{ord}\,(G(x) - x - qf_d x^d) > d. \tag{5.20}$$

Proof. It is easy to prove (5.19) by induction for positive q, and the result for negative q then follows. It is also easy to prove by induction that (5.20) holds if G is defined by (5.18). From Lemma 9.4 of Kuczma (1968) the solution of (5.19) satisfying (5.20) is unique, so the result follows. □

One $T(x)$ is known, we can solve (5.19) for $G(x)$, using the "initial condition" (5.20). Since (5.19) is a nonlinear differential equation for G, we can use a Newton-type method as described in Brent and Kung (1978). The algorithms are given below. First we summarize the result:

THEOREM 5.1. *Assume $f_1 = 1$ and let $G = F^{[q]}(x)$. Then $g_0, \cdots, g_{n-1}$ can be computed in time $O(\mathrm{COMP}_2\,(n))$.*

Proof. First find $t_d, \cdots, t_{n-1}$ such that $T(x)$ satisfies (5.1), as in Corollary 5.2, in time $O(\mathrm{COMP}_2\,(n))$. Then solve (5.19) and (5.20) by Algorithm 5.3 below (in time $O(\mathrm{COMP}\,(n))$ to find $g_0, \cdots, g_{n-1}$. □

Remark 5.2. Note that q need not be an integer in Theorem 5.1. Kuczma (1968, Thm. 9.15] considers the question of when $F^{[q]}(x)$ is analytic. See also Baker (1964) and Szekeres (1964).

ALGORITHM 5.1. The algorithm $\mathfrak{B}(A, F, Y, k, d, n)$ finds $y_0, \cdots, y_{n-1}$ such that $Y(n)$ satisfies (5.8). It is assumed that $n > 0$, $a_0, \cdots, a_{n-1}$ and $f_1, \cdots, f_{d+n-1}$ are given, and that the conditions stated after Example 5.1 are satisfied. $\mathfrak{B}(A, F, Y, k, d, n)$ is defined recursively by:

```
**if** $n = 1$ **then** {define $y_0$ by (5.9)}
**else** {$p \leftarrow \lceil n/2 \rceil$;
    $\mathfrak{B}(A, F, U, k, d, p)$;
    Compute $U(F(x)) \bmod x^{d+2p-1}$;
    Compute $R(x) \bmod x^p$ from (5.4) with $n$ replaced by $p$;
    $\mathfrak{B}((-R, F, V, k+p, d, p)$;
    **for** $j \leftarrow 0$ step 1 **until** $j-1$ **do**
        $\{y_j \leftarrow u_j;\ y_{p+j} \leftarrow v_j\}\}$.
```

ALGORITHM 5.2. The algorithm $\mathfrak{S}(F, T, d, n)$ finds $t_d, \cdots, t_{n-1}$ such that $T(x)$ satisfies (5.14). It is assumed that $f_1, \cdots, f_{n-1}$ are given and that the conditions of Corollary 5.2 are satisfied.

```
$Y \leftarrow 0;\ T \leftarrow 0;\ t_d \leftarrow 1$;
**if** $n > d+1$ **then** {compute $A(x) \bmod x^{n-d-1}$ from (5.15);
    $\mathfrak{S}(A, F, Y, d+1, d, n-d-1)$;
    **for** $j \leftarrow d+1$ **until** $n-1$ **do** $t_j \leftarrow y_{j-d-1}\}$.
```

ALGORITHM 5.3. The following algorithm computes $g_0, \cdots, g_{n-1}$, such that $G(x) = F^{[q]}(x)$. It is assumed that $t_d, \cdots, t_{n-1}$ have been computed using Algorithm 5.2.

```
$G \leftarrow x + qf_d x^d$;
$k \leftarrow 1$;
**while** $k + d < n$ **do**
    $\{k \leftarrow \min(2k, n-d)$;
    $R \leftarrow \dfrac{T(G(x)) - G'(x)T(x)}{x^d T(x)} \bmod x^{k-1}$;
    $U \leftarrow \dfrac{d}{x} - \dfrac{T'(G(x))}{T(x)} \bmod x^{k-2}$;
    $E \leftarrow \exp\left(\int_0^x U(y)\,dy\right) \bmod x^{k-1}$;
    $V \leftarrow \dfrac{1}{E(x)} \int_0^x E(y)R(y)\,dy \bmod x^k$;
    $G \leftarrow G + x^d V \bmod x^{k+d}\}$.
```

Remark 5.3. It can be verified that all the quantities appearing on the lefthand sides in Algorithm 5.3 are indeed power series.

6. Multiplier nontrivial root of unity. In this section we consider the only remaining case: $f_1 \neq 1$, $f_1^m = 1$ for some integer $m > 1$. By Remark 3.2 we may assume $m \leqq n-2$. We also assume q is an integer.

Remark 6.1. The restriction to integer q is essential here. For example, let $F = -x + x^2 + x^3$. There is no formal power series for $F^{[1/2]}(x)$. That is, there is no power series $G(x)$ such that $G^{[2]}(x) = F(x)$ (Kuczma (1968, p. 304)).

In what follows we shall use the following algebraic relations:

$$F^{[p+q]}(x) = F^{[p]}(F^{[q]}(x)), \tag{6.1}$$

$$F^{[pq]}(x) = R^{[p]}(x), \quad \text{where } R(x) = F^{[q]}(x), \tag{6.2}$$

for integer p, q. If q is negative we compose $F^{[-1]}$ instead of F, so without loss of generality we may assume that q is positive. Let

$$q = mr + s, \tag{6.3}$$

where $r \geqq 0$, $0 \leqq s < m$. We can evaluate $M = F^{[m]} = x + \cdots$, and $F^{[s]}$ by the obvious "squaring" method in time $O(\text{COMP}(n) \lg m) = O(\text{COMP}(n) \lg n)$. Then, using the method of § 5, we can evaluate $F^{[mr]} = M^{[r]}$ in time $O(\text{COMP}_2(n))$. Finally, $F^{[q]} = F^{[mr]}(F^{[s]})$ may be evaluated by performing one composition. (An additional reversion is required if $q < 0$.) Thus we have established

THEOREM 6.1. *Assume* ord $(F) = 1$, $f_1 \neq 1$, $f_1^m = 1$ *for some* m *such that* $1 < m \leqq n-2$, q *integer, and let* $G = F^{[q]}$. *Then* $g_1, \cdots, g_{n-1}$ *can be evaluated in time* $O(\text{COMP}(n) \lg m + \text{COMP}_2(n))$.

Remark 6.2. If Δ is the real field (so the only roots of unity are ± 1) then Theorem 6.1 shows that $g_1, \cdots, g_{n-1}$ can be evaluated in time $O(\text{COMP}_2(n))$.

7. Summary and open problems. From Theorems 3.1, 4.1, 5.1, and 6.1 we have

THEOREM 7.1. *Let* $F(x)$ *be a formal power series, ord* $(F) \geqq 1$, *and let* $G(x) = F^{[q]}(x)$. *If* q *satisfies the following conditions*:

(i) *If* ord $(F) > 1$, *then* q *is a positive integer*;

(ii) *If the multiplier* f_1 *is a nontrivial root of unity, then* q *is an integer*;

(iii) f_1^q *is defined*;

and if f_1^q *is given, then* $g_1, \cdots, g_n$ *can be computed in time* $O(\text{COMP}_2(n))$ *and this bound is independent of* q.

Different defining equations are used for the various cases we have had to consider. For the reader's convenience we summarize them here. As before, $G = F^{[q]}$.

I. Regular case: $f_1 \neq 0$, $f_1^m \neq 1$, $m = 1, 2, \cdots$. Define S by $S(F(x)) = f_1 S(x)$, ord $(S) = 1$. Then $G(x) = S^{[-1]}(f_1^q S(x))$.

II. $f_1 = 0$. Define S by $S(F(x)) = f_k(S(x))^k$, ord $(S) = 1$, $s_1 = 1$. Then $G(x) = S^{[-1]}\{f_k^{(k^q-1)/(k-1)}[S(x)]^{k^q}\}$.

III. $f_1 = 1$. Define T by $T(F(x)) = F'(x)T(x)$, and ord $(T) = $ ord $(F(x) - x)$. Then determine $G(x)$ from $T(G(x)) = G'(x)T(x)$ and (5.20).

IV. $f_1 \neq 1$, $f_1^m = 1$ for some integer $m > 1$. This can be reduced to case III.

It is possible to compute G using the same functional equation for cases I–III. Define $U(x)$ by

$$U(F(x)) = \frac{F'(x)}{\text{ord}\,(F)}\, U(x), \qquad \text{ord}\,(U(x)) = \text{ord}\,(F(x) - x). \tag{7.1}$$

$U(x)$ exists and is unique up to a scaling factor. In fact, in cases I and II we have

$$U(x) = cS(x)/S'(x), \tag{7.2}$$

and in case III we have $U(x) = c'T(x)$, for some nonzero constants c and c'. Also, it is easy to prove that G satisfies

$$U(G(x)) = \frac{G'(x)}{[\text{ord}\,(F)]^q} U(x). \tag{7.3}$$

Although a unified treatment of cases I–III using (7.1) and (7.3) would be possible, it is simpler to use the Schroeder function $S(x)$ of (3.1) in case I and the generalized Schroeder function of (4.1) in case II, for then G is given explicitly by (3.23) or (4.2) instead of implicitly as a certain solution of (7.3). Also, in proving properties of algorithms for the computation of G by either method, it is natural to consider cases I–III separately.

The techniques of §§ 3 and 5 can be applied to far more general nonlinear functional equations. We shall report on this elsewhere.

To conclude we list some open problems suggested by the results of the paper.

1. If the field Δ is such that $\text{MULT}\,(n) = O(n \lg n)$ then the fastest algorithm known for composition is $O((n \lg n)^{3/2})$. No nontrivial lower bound is known.

a. Is composition harder than multiplication? (It is at least as hard.)

b. Although there are only n inputs and n outputs, the best upper bound known is $O((n \lg n)^{3/2})$. This is comparable to matrix multiplication where there are $2n^2$ inputs and n^2 outputs but the best upper bound known (Pan (1978)) is $O(n^{2.79})$. Can the Brent–Kung upper bound be reduced?

c. Is $\alpha > 1$ in the notation of (2.5)? An affirmative answer would show that $\text{COMP}_2\,(n) = O(\text{COMP}\,(n))$.

2. Brent and Kung (1978) showed that, for the reversion problem $R(x) = F^{(-1)}(x)$, the complexity of computing $R_n(x)$ is $O(\text{COMP}\,(n))$. Consider computing $R_n(x_0)$ for a scalar x_0. This problem has n inputs and one output. Brent and Kung (1978) showed its complexity to be $O(\text{MULT}\,(n))$. If $G(x) = F^{[q]}(x)$, what is the complexity of computing $G_n(x_0)$? Is it less than the complexity of computing $G_n(x)$?

3. What are the numerical properties of our algorithms? For example, we expect the computation of the Schroeder function to be ill-conditioned if f_1^m is close to 1 for some $m \leqq n-2$; see (3.13). Cherry (1964) discusses this problem in conjunction with a problem in dynamical systems.

4. What are the complexity bounds for exact arithmetic over the rational field?

Acknowledgment. We are deeply indebted to M. L. Fredman who pointed out to us the critically important idea of using the Schroeder function to effect a change of representation in the regular case. We thank D. E. Knuth who pointed out to us that a single functional equation, (7.1), can be used for three of the cases. Finally, we thank M. Sapsford and M. Sieveking for their careful reading of the manuscript.

REFERENCES

J. ACZÉL (1966), *Lectures on Functional Equations and Their Applications*, Academic Press, New York.

I. N. BAKER (1964), *Fractional iteration near a fixpoint of multiplier 1*, J. Austral. Math. Soc., 4, pp. 143–148.

A. BORODIN AND I. MUNRO (1975), *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York.

R. P. BRENT (1976), *Multiple-precision zero-finding methods and the complexity of elementary function evaluation*, Analytic Computational Complexity, J. F. Traub, ed., Academic Press, New York, pp. 151–176.

R. P. BRENT AND H. T. KUNG (1978), *Fast algorithms for manipulating formal power series*, Department of Computer Science Report, Carnegie–Mellon University, 1976. Also J. Assoc. Comput. Mech., 25, pp. 581–595.

T. M. CHERRY (1964), *A Singular Case of Iteration of Analytic Functions: A Contribution to the Small-Divisor Problem*, Nonlinear Problems of Engineering, W. F. Ames, ed., Academic Press, New York, pp. 29–50.

N. G. DE BRUIJN (1970), *Asymptotic Methods in Analysis* (Third Edition), North-Holland Publishing Company, Amsterdam.

W. FELLER (1957), *An Introduction to Probability Theory and its Applications*, vol. I, Second Edition, John Wiley, New York.

F. GROSS (1972), *Factorization of Meromorphic Functions*, U.S. Government Printing Office, Washington, DC.

T. E. HARRIS (1963), *The Theory of Branching Processes*, Springer-Verlag, Berlin.

P. HENRICI (1974), *Applied and Computational Complex Analysis*, vol. 1, John Wiley, New York.

D. E. KNUTH (1969), *The Art of Computer Programming*, vol. 2, Addison-Wesley, Reading, MA.

——— (1976), *Big omicron and big omega and big theta*, SIGACT News 8, no. 2, pp. 18–24.

M. KUCZMA (1968), *Functional Equations in a Single Variable*, PWN-Polish Scientific Publishers, Warsaw.

H. T. KUNG AND J. F. TRAUB (1978), *All algebraic functions can be computed fast*, Department of Computer Science Report, Carnegie–Mellon University, 1976. Also J. Assoc. Comput. Mech., 25, pp. 245–260.

H. LEVY AND F. LESSMAN (1961), *Finite Difference Equations*, Pitman, London.

Z. A. MELZAK (1973), *Companion to Concrete Mathematics*, John Wiley, New York.

V. PAN (1978), *Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations*, 19th Annual Symposium on Foundations of Computer Science, IEEE Computer Society.

D. S. PARKER, JR, (1977), *Nonlinear recurrences and parallel computation*, High Speed Computer and Algorithm Organization, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds., Academic Press, New York, pp. 317–320.

E. SCHROEDER (1871), *Über iterierte Funktionen*, Math. Ann., 3, pp. 296–322.

G. SZEKERES (1964), *Fractional iteration of entire and rational functions*, J. Austral. Math. Soc., 4, pp. 129–142.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/09-01-0005 $01.00/0

THE SEMANTICS OF CALL-BY-VALUE AND CALL-BY-NAME IN A NONDETERMINISTIC ENVIRONMENT*

M. C. B. HENNESSY†

Abstract. Nondeterministic recursive procedures are considered in which parameters may be passed by use of call-by-value or one of two different formulations of call-by-name. These procedures are given an operational semantics via an evaluation mechanism. We define a denotational semantics using so-called nondeterministic domains, which are function spaces endowed with two partial orders. An operational characterization of equality of procedures under this denotational semantics is then given.

Key words. recursive programs, nondeterminism, call-by-value, call-by-name, mathematical semantics, computation rules, full-abstraction

Introduction. The problem of defining an adequate semantics for recursive definitions which allow various types of parameter-passing mechanisms has generated a considerable amount of interest in the literature (see [1], [2], [3], [20]). Consider for example the well-known recursive definition

$$F\langle X, Y\rangle \Leftarrow \mathrm{IF} X = 0 \mathrm{THEN} 0 \mathrm{ELSE} F\langle X-1, F\langle X, Y\rangle\rangle.$$

Interpreted as a fix point equation over the flat cpo of nonnegative integers it has as its least solution

$$f(x, y) = \begin{cases} 0 & \text{if } x = m \text{ for any nonnegative integer } m, \\ \perp & \text{otherwise ("}\perp\text{" means undefined).} \end{cases}$$

This also happens to coincide with the computed function if a call-by-name (or outside-in) evaluation mechanism is used. However, if a call-by-value (or inside-out) evaluation mechanism is used the computed function is

$$f_v(x, y) = \begin{cases} 0 & \text{if } x = 0, \\ \perp & \text{otherwise.} \end{cases}$$

In [20] the conclusion is drawn that the call-by-value evaluation mechanism is incorrect.

This situation can, however, be viewed from a different perspective. Both these evaluation mechanisms exist a priori and we can pose the problem of finding a denotational semantics which adequately reflects the operational behavior of programs under each of these mechanisms. Since the two mechanisms lead, in general, to different computed functions we have to define a call-by-name denotational semantics to model the call-by-name mechanism and a call-by-value denotational semantics to model the call-by-value mechanism. As will be seen below such denotational semantics can be defined and are similar in many ways. Indeed the only difference is that different methods are used to compose functions. Therefore we extend the notion of recursive definition so as to allow the procedures to be called using either the call-by-name or call-by-value mechanisms.

In fact the recursive definitions used will be nondeterministic. Thus we can form definitions such as

$$F\langle X\rangle \Leftarrow X \textbf{ or } F\langle S(X)\rangle$$

* Received by the editors, April 14, 1977, and in revised form January 12, 1979.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Now at Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland EH8 9NW.

and the presence of a term T_1 **or** T_2 indicates that we can choose to evaluate T_1 or to evaluate T_2. In the presence of such nondeterminism a dilemma occurs as to the exact meaning of the call-by-name mechanism. Consider for example the definition

$$F\langle X\rangle \Leftarrow \mathrm{IF}Z(X)\mathrm{THEN}X\mathrm{ELSE}2$$

where Z is a test for 0. To evaluate the term $F\langle 0$ **or** $1\rangle$ we must decide how to apply the procedure to the parameter '0 **or** 1'. One approach is to consider '0 **or** 1' as a set and a procedure is applied to a set by applying it to an (arbitrary) element of the set. Thus one would get two possible outcomes by

$$F\langle 0\ \mathbf{or}\ 1\rangle \to F\langle 0\rangle \to \cdots \to 0$$

$$F\langle 0\ \mathbf{or}\ 1\rangle \to F\langle 1\rangle \to \cdots \to 2.$$

Another approach is to use the body substitution rule of ALGOL. In this case we would get not only the possible outcomes 0, 2 but also 1 via the evaluation

$$F\langle 0\ \mathbf{or}\ 1\rangle \to \mathrm{IF}Z(0\ \mathbf{or}\ 1)\mathrm{THEN}0\ \mathbf{or}\ 1\mathrm{ELSE}2$$

$$\to \mathrm{IF}Z(0)\mathrm{THEN}0\ \mathbf{or}\ 1\mathrm{ELSE}2$$

$$\to 0\ \mathbf{or}\ 1 \to 1.$$

Rather than make a choice between these two approaches we allow both. The former will be called *call-time choice* because all choices in the parameters must be made when a call to the procedure is made. The latter will be called *run-time choice* because in this case choices may be made at any time during the execution of a call to a procedure. Thus for the remainder of the paper 'call-by-name' will not be used. However for deterministic programs run-time choice and call-time choice both coincide with the usual notion of call-by-name.

In this paper we give a mathematical or denotational semantics of recursive definitions which allow call-by-value, call-time choice and run-time choice parameter-passing mechanisms. A knowledge of the mathematical approach to semantics is assumed (see [12], [19]). In § 2 the programming language (i.e. recursive definitions) and evaluation mechanisms are defined. In § 3 the mathematical model is introduced and in § 4 we investigate the relationship between the mathematical and denotational semantics. In § 1 we isolate the necessary mathematical definitions and constructs which are needed throughout the paper. The reader may wish to skip over this section on first reading and refer to its contents when required. The results in this paper were originally reported in [9] and outlined in [7].

Related work. The relationship between call-by-value and call-by-name for deterministic programs has been discussed by many authors. For example in [20], [13] call-by-name is considered correct and call-by-value incorrect. These opinions are discussed at length in [1] where the author shows that the call-by-value computed functions are also least fixpoints with respect to a partial order. However the partial order is defined operationally. In [18] an attempt is made to give a first-order reduction of call-by-name to call-by-value but the results seem invalidated because of the confusion between call-time choice and run-time choice. Call-by-name and call-by-value are discussed in a different setting in [17].

Nondeterministic programs are discussed in an operational setting in [4], [11] and in a denotational setting in [5], [16]. This paper owes much to the original approach to nondeterminism in [5]. However the models used can be obtained by using the powerdomain construction of [16]. More details concerning the actual choice of model, from the point of view of nondeterminism, may be obtained in [10].

1. Mathematical preliminaries. Let SORT be a set of elementary sorts. We will always assume that tr $\in$ SORT. A *sort* is any vector of elementary sorts. Elementary sorts and sorts of length one will usually be identified. A *type* is any expression of the form $m \to n$, where m, n are sorts. Elementary sorts will usually be denoted by symbols such as s, t, etc., sorts by m, n, k, etc. and types by a, b, c.

A *partial order* or *po* is a tuple $\langle D, \leqq\rangle$ where D is a nonempty set and $\leqq$ is a reflexive, transitive and anti-symmetric ordering on D. A $\leqq$-chain in D is a sequence of elements $\{x_n \in D : n \geqq 0\}$ such that $x_n \leqq x_{n+1}$. A po D is a *complete po* or *cpo* if (i) D has a least element $\perp$ with respect to $\leqq$ and (ii) every $\leqq$-chain X in D has a least upper bound in D, denoted $\bigvee\{X\}$. $\langle D, \leqq\rangle$ will be denoted by D if $\leqq$ is clear from the context.

A relation $R \subseteq X \times Y$ is *bounded* if for every $x \in X$ there is at most a finite number of $y \in Y$ such that $\langle x, y\rangle \in R$. R is *total* if for every $x \in X$ there is at least one y such that $\langle x, y\rangle \in R$. If X and Y are cpo's then R is *complete* if whenever $\{x_n : n \in N\}$, $\{y_n : n \in N\}$ are $\leqq$-chains, with limits x, y respectively, such that $\langle x_n, y_n\rangle \in R$ for all n then $\langle x, y\rangle \in R$. If R is a total complete function then it is said to be *continuous*. A (total) function f from X to Y, as above, is *monotonic* if $x_1 \leqq x_2$ in X implies $f(x_1) \leqq f(x_2)$ in Y.

A *nondeterministic domain*, written *nda*, is a triple $\langle D, \leqq, \subseteq\rangle$ such that

(i) $\langle D, \leqq\rangle$ is a cpo,

(ii) $\langle D, \subseteq\rangle$ is a po in which every two elements $d_1, d_2 \in D$ have a least upper bound $d_1 \cup d_2 \in D$,

(iii) $\cup$ is $\leqq$-continuous i.e. (continuous with respect to $\leqq$),

(iv) $\subseteq$ is $\leqq$-complete.

Suppose $\langle D_1, \leqq_1, \subseteq_1\rangle$, $\langle D_2, \leqq_2, \subseteq_2\rangle$ are both nda's. Let $[D_1, D_2]$ denote the set of $\subseteq$-monotonic, $\leqq$-continuous (total) functions from D_1 to D_2.

PROPOSITION 1.1. *$D_1 \times D_2$ and $[D_1, D_2]$ are both nda's under the pointwise induced partial orderings.*

Proof. The proof is by standard techniques. □

A *flat cpo* is a denumerable cpo D such that $\forall x, y \in D, x \leqq y$ implies $x = y$ or $x = \perp$. For any sort $m = \langle s_1, \cdots, s_k\rangle$ let D_m denote $D_{s_1} \times \cdots \times D_{s_k}$, where each D_{s_i} is a flat cpo. Let $\mathbb{D}_m$ be the set of k-tuples $\langle X_1, \cdots, X_k\rangle$ such that $\forall i$, $1 \leqq i \leqq k$,

(i) $X_i \subseteq D_{s_i}$,

(ii) $X_i \neq \phi$,

(iii) $|X_i| = \infty \to \perp \in X_i$.

The elements of $\mathbb{D}_m$ will usually be denoted by the symbols X, Y, Z and the ith-component of Y will be denoted by $e_i(Y)$. An element $X \in \mathbb{D}_m$ is called a *value* if it is a vector of singleton sets. If *any one* of these singleton sets contains $\perp$ it is called an *undefined value*. Otherwise it is a *defined value*. Values are very similar to elements of D_m and to emphasise this similarity they will be denoted by lower case symbols v, w etc. Moreover we will write $v \in X$ in place of $v \subseteq X$. If a is the type $(m \to n)$, $\mathbb{D}_a$ will denote the nda $[\mathbb{D}_m, \mathbb{D}_n]$.

For X, $Y \in \mathbb{D}_m$ let $X \subseteq Y$ if $e_i(X) \subseteq e_i(Y)$, $1 \leqq i \leqq k$. Note that $X \subseteq Y$ if and only if for every value v, $v \in X$ implies $v \in Y$. Let $X \leqq Y$ if $\forall i$, $1 \leqq i \leqq k$,

(i) $\forall x \in e_i(X) \exists y \in e_i(y), x \leqq y$,

(ii) $\forall y \in e_i(X) \exists x \in e_i(Y), x \leqq y$.

PROPOSITION 1.2. *$\langle \mathbb{D}_m, \leqq, \subseteq\rangle$ is an nda.*

Proof. The proof follows in a straightforward manner from the definition if Lemma 1.3, which is stated below, is used. □

LEMMA 1.3. *For every value v and $\leqq$-chain $\{X_n : n \in N\}$ in $\mathbb{D}_m$, $v \in \bigvee\{X_n\}$ if and only if there exists a K such that $v \in X_n$, $\forall n \geqq K$.*

Let $R \subseteq D_m \times D_t$, t an elementary sort, be a total bounded relation. The $\cup$

extension of R is the function from $\mathbb{D}_m$ to $\mathbb{D}_t$ defined by

$$f_R(X) = \bigcup\{y \mid \langle x, y\rangle \in R \text{ for some } x \in X\}.$$

Note that f_R may or may not be in $[\mathbb{D}_m, \mathbb{D}_t]$. *If* $R \subseteq D_m \times D_t$ is a bounded relation it can be extended to a total bounded relation $R_\perp$ as follows:

(i) if $d_i = \perp$ for any i, then $\langle\langle d_1, \cdots, d_k\rangle, \perp\rangle \in R_\perp$,

(ii) if there does not exist a d such that

$$\langle\langle d_1, \cdots, d_k\rangle, d\rangle \in R \text{ then } \langle\langle d_1, \cdots, d_k\rangle, \perp\rangle \in R_\perp,$$

(iii) otherwise if $\langle\langle d_1, \cdots, d_k\rangle, d\rangle \in R$ then

$$\langle\langle d_1, \cdots, d_k\rangle, d\rangle \in R_\perp.$$

The *natural extension* of a bounded relation R is defined to be the $\bigcup$-extension of $R_\perp$.

LEMMA 1.4. *The natural extension of a bounded relation* $R \subseteq D_m \times D_t$ *is in* $[\mathbb{D}_m, \mathbb{D}_t]$.

Proof. For convenience we let f denote the natural extension of R. We must first prove that for every $X \in \mathbb{D}_m$, $f(X) \in \mathbb{D}_t$. It suffices to prove that $|f(X)| = \infty$ implies $\perp \in f(X)$. If $|e_i(X)|$ is finite for all i then since R is bounded $|f(\mathrm{X})|$ is finite. Therefore if $|f(X)| = \infty$ there exists an i such that $\perp \in e_i(X)$ and from the definition of natural extension it follows that $\perp \in f(X)$.

Let $X = V\{X_n : n \in N\}$. We now show that $f(X) = V\{f(X_n) : n \in N\}$. For any $d \neq \perp$,

$$\begin{aligned} d \in f(X) &\Leftrightarrow \exists e \in V\{X_n : n \in N\} \text{ such that } \langle e, d\rangle \in R. \\ &\Leftrightarrow \exists k \in N, e \in V\{X_n : n \in N\} \text{ such that } \langle e, d\rangle \in R \\ &\quad\ \text{and } e \in X_n \forall n \geqq k, \text{ from Lemma 1.3.} \\ &\Leftrightarrow \exists k \in N \text{ such that } d \in f(X_n) \forall n \geqq k \\ &\Leftrightarrow d \in V\{f(X_n) : n \in N\}, \text{ from Lemma 1.3.} \end{aligned}$$

Suppose $\perp \in f(X)$. Then $\langle x, \perp\rangle \in R_\perp$ for some $x \in X$. Now from the construction of $R_\perp$ it follows that if $\langle d, \perp\rangle \in R_\perp$ and $d' \leqq d$ then $\langle d', \perp\rangle \in R_\perp$. For each $n \in N$ there exists an $x_n \in X_n$, $x_n \leqq x$. Therefore $\langle x_n, \perp\rangle \in R_\perp$. So $\perp \in \mathrm{f}(\mathrm{X}_\mathrm{n})$ for each $n \in N$. From Lemma 1.3 it follows that $\perp \in V\{f(X_n) : n \in N\}$. Suppose $\perp \in V\{f(X_n) : n \in N\}$. Then for each n there exists $x_n \in X_n$ such that $\langle x_n, \perp\rangle \in R$. If $|X|$ is infinite there exists at least one i such that $|e_i(X)|$ is infinite. It follows that $\perp \in e_i(X)$ and therefore $\perp \in f(X)$. On the other hand if $|X|$ is finite then there exists an n such that $X_n = X$. Therefore $x_n \in X$ and $\perp \in f(X)$. □

An element $X \in \mathbb{D}_m$ is *finite* if for each i, $e_i(X)$ is a finite set. The next proposition states that continuous functions are completely determined by their effect on finite elements.

LEMMA 1.5. *If* $f, g \in [\mathbb{D}_m, \mathbb{D}_t]$ *and* $f(X_f) = g(X_f)$ *for every finite element* X_f *then* $f = g$.

Proof. We prove that $f(X) = g(X)$ for an arbitrary $X \in \mathbb{D}_m$. If $|e_i(X)| = \infty$ we let $e_i(X)$ be $\{n^i_0, n^i_1, n^i_2, \cdots\}$, where $n^i_0 = \perp$. Now define a sequence of finite sets $\{X_n : n \in N\}$ as follows:

(i) $X_0 = \{\langle\perp, \cdots, \perp\rangle\}$,

(ii) $$e_i(X_{k+1}) = \begin{cases} e_i(X) & \text{if } |e_i(X)| \neq \infty, \\ e_i(X_k) \cup \{n^i_{k+1}\} & \text{otherwise.} \end{cases}$$

Then $X_n \leqq X_{n+1}$ and it is easy to prove that $X = V\{X_n : n \in N\}$. Therefore $f(X) = V\{f(X_n) : n \in N\} = V\{g(X_n) : n \in N\} = g(X)$. □

2. Programming languages.

2.1. Syntax. For each $s \in \mathrm{SORT}$ let K^s be a set of data-constant symbols of sort s. Let $\{X_i : i \in N\}$ be a set of *data-constant* variables; let $G_1, \cdots, G_n$ be *constant function symbols*, and $F_1, \cdots, F_m$ be *variable function symbols*, each with a specific type of the form $(m \to s)$, s an elementary sort. Type and sort indications will be omitted whenever possible. The set of *terms* together with their associated types is defined as follows:

The notation $T \in a$ is used to denote T is a term of type a.

(i) $X_i \in (n \to s_i)$, where $n = \langle s_1, \cdots, s_i, \cdots, s_k \rangle$;

(ii) if $C \in K^s$, $C \in (n \to s)$ if $n = \langle s_1, \cdots, s, \cdots, s_k \rangle$;

(iii) if G is of type $(m \to s)$, where $m = \langle s_1, \cdots, s_k \rangle$ and $T_i \in (n \to s_i)$, $1 \leqq i \leqq k$, then $G(T_1, \cdots, T_k) \in (n \to s)$;

(iv) if $T_1, T_2 \in a$ then T_1 **or** $T_2 \in a$;

(v) if $T_1 \in (n \to tr)$, T_2, $T_3 \in (n \to s)$ then IF $(T_1, T_2, T_3) \in (n \to s)$;

(vi) if F_i is of type $(m \to s)$, where $m = \langle s_1, \cdots, s_k \rangle$ and $T_i \in (n \to s_i)$, $1 \leqq i \leqq k$, then $F_i^\gamma \langle T_1, \cdots, T_k \rangle \in (m \to s)$, $\gamma =$ 'r', 'c', 'v'.

A *program P* is a set of recursive definitions of the form

$$\begin{aligned} &\mathrm{F}_1\langle \mathrm{X}_1, \cdots, \mathrm{X}_{\mathrm{n}_1} \rangle \Leftarrow \mathrm{P}_1, \\ &\quad\vdots \\ &F_k\langle X_1, \cdots, X_{n_k} \rangle \Leftarrow P_k, \end{aligned} \tag{1}$$

where P_i is a term whose associated type is the same as that of F_i. For reasons of convenience the following restrictions are made:

(i) if X_j occurs in P_i then $1 \leqq j \leqq n_i$;

(ii) if F_i occurs in any P_j then $1 \leqq i \leqq k$;

(iii) every occurrence of X_j in P_i has associated with it the type $(m_i \to s_j)$ where $m_i = (s_1, \cdots, s_j, \cdots, s_k)$;

(iv) if $C \in K^s$ has an occurrence in P_i then it has associated with it the type $(m_i \to s)$, with m_i as above.

However, these conventions do not raise problems because for the most part types are ignored. In fact they will only be used in § 3.2 where we associated a function with every term. When the typing conventions are ignored the resulting programs are identical to the recursive definitions in [13], [20] except that the nondeterministic '**or**' is used and the superscripts r, c, v appear on the F_i's. These superscripts denote which parameter-passing mechanism is allowed.

A term T is called a *P-program term* if every F_i which appears in T appers in P. For the remainder of the paper we will assume the existence of some (arbitrary) program as in (1) and consequently a P-program term will be simply called a program term.

2.2. Operational semantics. We assume that in each K^s we have an 'error constant', e_s, which is meant to denote an error condition in a program. We also assume that $K^{\mathrm{tr}} = \{t, f, e_{\mathrm{tr}}\}$. Finally if $m = \langle s_1, \cdots, s_k \rangle$ let $K^m = K^{s_1} \times \cdots \times K^{s_k}$.

To define an evaluation mechanism we must have associated with each G of type $(m \to s)$ a total bounded relation over $K_m \times K_s$, which we denote by $E(G)$. Given such an association we can define an immediate reduction relation, $\overset{\circ}{\to}$, between *constant* terms, i.e., terms not involving any data-constant variables, as follows:

(i) *Data-manipulation.*

$$G\langle C_1, \cdots, C_k \rangle \overset{\circ}{\to} C' \quad \text{if } \langle\langle C_1, \cdots, C_k \rangle, C' \rangle \in E(G),$$

$$G\langle T_1, \cdots, T_i, \cdots, T_k \rangle \overset{\circ}{\to} G\langle T_1, \cdots, T_i', \cdots, T_k \rangle \quad \text{if } T_i \overset{\circ}{\to} T_i'.$$

(ii) *Testing.*

$$\mathrm{IF}\,(t, T_1, T_2,) \overset{\circ}{\to} T_1,$$

$$\mathrm{IF}\,(f, T_1, T_2) \overset{\circ}{\to} T_2,$$

$$\mathrm{IF}\,(e, T_1, T_2) \overset{\circ}{\to} e,$$

$$\mathrm{IF}\,(T_1, T_2, T_3) \overset{\circ}{\to} \mathrm{IF}\,(T'_1, T_2, T_3) \quad \text{if } T_1 \overset{\circ}{\to} T'_1.$$

(iii) *Choice.*

$$T_1 \textbf{ or } T_2 \overset{\circ}{\to} T_1,$$

$$T_1 \textbf{ or } T_2 \overset{\circ}{\to} T_2.$$

(iv) *Procedure calls.*

(a) *Run-time choice.* This is akin to the body-replacement rule of ALGOL 60 and consequently no restrictions are made on what may be passed to the procedure.

$$F^r_i\langle T_1, \cdots, T_k\rangle \overset{\circ}{\to} [T_1|X_1, \cdots, T_k|X_k]P_i$$

where the term on the right-hand side denotes the result of simultaneously substituting T_j for X_j in P_i for each j, $1 \leqq j \leqq k$. In programming terms this means that pointers to the 'code' for T_j are passed to the procedure P_i and each time P_i requires a value it 'runs the code' itself. Note that because of the nondeterminism we may get a different result each time T_j is evaluated.

(b) *Call-time choice.* Here P_i can be called only if it can be assured that whenever the T_j is evaluated no choices will have to be made—that is, the T_j are all *deterministic.* The exact definition of deterministic is somewhat tricky and is given in the Appendix.

$$F^c_i\langle T_1, \cdots, T_k\rangle \overset{\circ}{\to} [T_1|X_1, \cdots, T_k|X_k]P_i \quad \text{if } T_j \text{ is deterministic,} \quad 1 \leqq j \leqq k.$$

$$F^c_i\langle T_1, \cdots, T_i, \cdots, T_k\rangle \overset{\circ}{\to} F^c_i\langle T_1, \cdots, T'_i, \cdots, T_k\rangle \quad \text{if } T_i \overset{\circ}{\to} T'_i, \quad T_i \text{ not deterministic.}$$

(c) *Call-by-value.* This is the well-known and well motivated ALGOL 60 mechanism whereby procedures will only accept as parameters real data constants.

$$F^v_i\langle C_1, \cdots, C_k\rangle \overset{\circ}{\to} [C_1|X_1, \cdots, C_k|X_k]P_i,$$

$$F^v_i\langle T_i, \cdots, T_i, \cdots, T_k\rangle \overset{\circ}{\to} F^v_i\langle T_1, \cdots, T'_i, \cdots, T_k\rangle \quad \text{if } T_i \overset{\circ}{\to} T'_i.$$

Let $\to$ be the reflexive transitive closure of $\overset{\circ}{\to}$. If $T_1 \to T_2$ we say that T_1 *reduces to* T_2.

Note that the notions of 'call-by-value' etc. are no longer computation rules in the sense of [20]. That is they are not algorithms for specifying the next occurrence of an F_i to be expanded. Instead they specify certain characteristics that the actual parameters of a particular call to an F_i should have before that call can be made. The evaluation of a term always proceeds from the outside-in, subject to these constraints. But for example if each F_i has the superscript v then the net result is an inside-out (or call-by-value as in [20]) evaluation.

2.3. Examples. Many data-structures such as S-expressions, stacks, etc. (see [13]) can be formulated within this framework. We give as a simple example the natural numbers.

Let SORT $= \{i, \mathrm{tr}\}$ and let $K^i = \{k_n : n \in N\} \cup \{e_i\}$. The set of constant function symbols consist of $Z \in (i \to \mathrm{tr})$, $S, P \in (i \to i)$. The intended interpretation is that k_α is a notation for the natural number α, Z is a test for zero, S is the successor function and P

the predecessor function. With this interpretation in mind we define

$$E(S)\{\langle k_n, k_{n+1}\rangle : n \in N\} \cup \{e_i, e_i\},$$

$$E(P) = \{\langle k_{n+1}, k_n\rangle : n \in N\} \cup \{\langle k_0, e_i\rangle, \langle e_i, e_i\rangle\},$$

$$E(Z) = \{\langle k_{n+1}, f\rangle : n \in N\} \cup \{\langle k_0, t\rangle, \langle e_i, e_{\mathrm{tr}}\rangle\}.$$

Consider the program

$$F\langle X_1, X_2\rangle \Leftarrow \mathrm{IF}(Z(X_1), k_0, F^v\langle P(X_1), F^c\langle X_1, X_2\rangle\rangle). \tag{2}$$

Then $F^v\langle k_n, k_m\rangle \to k_0$ if and only if $n = 0$. However, if the program

$$F\langle X_1, X_2\rangle \Leftarrow \mathrm{IF}(Z(X_1), k_0, F^c\langle P(X_1), F^c\langle X_1, X_2\rangle\rangle) \tag{3}$$

is used then $F^v\langle k_n, k_m\rangle \to k_0$ for every n, $m \in N$. A simple example of a nondeterministic term is the program

$$F\langle X\rangle \Leftarrow X \textbf{ or } F^v\langle S(X)\rangle. \tag{4}$$

With this program the term $F^v\langle k_n\rangle$ can be reduced to k_m for any $m \geqq n$. Finally consider the program

$$\begin{aligned} &F_1\langle X\rangle \Leftarrow \mathrm{IF}\ (Z(X), k_0, F_1^r\langle S(X) \textbf{ or } F_2^r\langle X\rangle\rangle \textbf{ or } X),\\ &F_2\langle X\rangle \Leftarrow \mathrm{IF}\ (Z(X), k_0, F_2^r\langle F_1^r\langle X\rangle \textbf{ or } P(X)\rangle \textbf{ or } X). \end{aligned} \tag{5}$$

Then $F^v\langle k_{n+1}\rangle \to k_m$ for every n, $m \in N$.

3. The mathematical semantics.

3.1. Nondeterministic domains. In Scott's approach to semantics [19] it is usual to associate with each program a functional over some domain and to consider the meaning of the program to be the least fixpoint of this functional. To ensure the existence of such a fixpoint, domains can be taken to be cpo's over which the functionals associated with programs are continuous. Since we are dealing with unstructured data-types it is sufficient to consider flat cpo's. However, as pointed out in [6], [16], when dealing with nondeterministic domains it is necessary to consider some form of power-domain. Because we only use flat cpo's this construction is relatively simple and is given in § 1. In that section we also isolated what we deem to be the necessary properties of these power-domains (see [6]) and the resulting structure is called a *nondeterministic domain, or* nda.

Every program term $T \in (m \to s)$ will be interpreted as an element of an nda $[\mathbb{D}_m, \mathbb{D}_s]$. This interpretation is achieved by associating with every program (such as (1)) a $\leqq$-continuous $\subseteq$-monotonic functional over the relevant cross-product of function spaces and interpreting F_i as the ith-component of the least fixpoint of this functional. This least fixpoint is taken with respect to the ordering $\leqq$, called the *computational ordering*. The order ordering, $\subseteq$, is called the *subset ordering* and is used to model the nondeterminism. However, before these functionals are defined we must say exactly which nda's will be used and we must associate with each syntactic construct in the language a corresponding semantic construct.

For each $s \in \mathrm{SORT}$ let D_s be the flat cpo obtained by adjoining to K^s the undefined symbol $\perp_s$. Let $\mathbb{D}_s$ be the nda generated by D_s as in § 1. If m is a sort and a a type, let $\mathbb{D}_m$, $\mathbb{D}_a$ be the nda's as defined in § 1. If these subscripts are obvious from the context they will be omitted. We now consider the various semantic constructs used to interpret programs.

(i) *Datamanipulation.* We let g denote the natural extension of $E(G)$, for every constant function symbol G. g will be used to interpret G.

(ii) *Testing.* Let

$$\mathrm{dif}(x, y, z) = \begin{cases} y & \text{if } x = t, \\ z & \text{if } x = f, \\ \perp & \text{if } x = \perp, \\ e & \text{if } x = e. \end{cases}$$

Let a, b be the types $(m \to \mathrm{tr})$, $(m \to s)$ respectively. For $f_1 \in \mathbb{D}_a, f_2, f_3 \in \mathbb{D}_b$ define if (f_1, f_2, f_3) by

$$\text{if } (f_1, f_2, f_3)(X) = \bigcup\{\mathrm{dif}\,(x_1, x_2, x_3) \mid x_i \in f_i(X)\} \quad \forall X \in \mathbb{D}_m.$$

The functional if will be used to model the syntactic construct IF.

(iii) *Choice.* If f, $g \in \mathbb{D}_a$ then $f \cup g \in \mathbb{D}_a$. Moreover since $\mathbb{D}_a$ is an nda $\cup$ is $\leqq$-continuous. This functional will be the semantic counterpart to the syntact construct **or**.

(iv) *Tupling.* Let a_i be the type $(m \to s_i)$, $1 \leqq i \leqq k$ and let a be the type $(m \to n)$, where $n = \langle s_1, \cdots, s_k \rangle$. If $f_i \in \mathbb{D}_{a_i}$, $1 \leqq i \leqq k$, define $\langle f_1, \cdots, f_k \rangle \in \mathbb{D}_a$ by

$$\langle f_1, \cdots, f_k \rangle(X) = \langle f_1(X), \cdots, f_k(X) \rangle \quad \text{for all } X \in \mathbb{D}_m.$$

(v) *Procedure-calling.* Suppose we have assigned a 'meaning' f to a program module T_1 and a 'meaning' g to another module T_2. Then the meaning assigned to the composite module T_1 followed by T_2 is simply the composition of f and g. The composition gf is defined by $gf(X) = g(f(X))$. However, when different parameter-passing mechanisms are used the module T_2 may not accept arbitrary inputs from T_1. Instead these inputs must satisfy certain criteria, which are now considered.

(a) *Call-by-value.* If T_2 requests input from T_1, specifying call-by-value, then it will only accept as input data constants. Thus if T_1 can supply a vector, all of whose components are data constants, T_2 will hapily operate on them. Moreover if T_1 supplies a vector of inputs in which at least one component is not a data constant then T_1 will not stir. Instead T_1 will patiently wait for the entire vector to become defined. If this never happens the composite module will diverge. This leads to the following definition:

For $f \in \mathbb{D}_a$, $a = (m \to s)$ define f^v by

$$f^v(X) = \bigcup\{f(d) \mid d \in X, d \text{ a defined value}\}$$
$$\bigcup\{\perp \mid v \in X, v \text{ an undefined value}\}.$$

The functional: $f \mapsto f^v$ will be denoted by cv and is called *call-by-value operator.*

(b) *Call-time choice.* In this case T_2 will accept a vector as input from T_1 only if it is certain that each component of the vector is deterministic. This leads to the following definition of the *call-time choice operator* cc: $f \mapsto f^c$. For $f \in \mathbb{D}_a$, $a = (m \to s)$, *let* f^c be defined by

$$f^c(X) = \bigcup\{f(v) \mid v \in X, v \text{ a value}\}.$$

(c) *Run-time choice.* Here no restrictions are made on the input which T_2 will accept from T_1. Therefore the *run-time choice operator* rc: $f \mapsto f^r$ is simply the identity functional.

Reconsidering the modules T_1, T_2 above, if we specify that call-by-value is to be used the resulting composite module will be modeled by $g^v f$, whereas if call-time choice is specified $g^c f$ will be used.

The required properties of these propositions are summarized in the two following propositions.

PROPOSITION 3.1.1. (a) *If a, b, c are the types $(m \to n), (n \to k)$ and $(m \to k)$ respectively and if C denotes the composition operator then $C \in [\mathbb{D}_a \times \mathbb{D}_b, \mathbb{D}_c]$.*

(b) *If a is the type $(m \to s)$ then* cv, cc, rc $\in [\mathbb{D}_\mathrm{a}, \mathbb{D}_\mathrm{a}]$.

(c) *If a, b are the types $(m \to \mathrm{tr})$, $(\mathrm{m} \to \mathrm{s})$ respectively then* if $\in [\mathbb{D}_a \times \mathbb{D}_b \times \mathbb{D}_b, \mathbb{D}_b]$.

(d) *If a_i is the type $(m \to s_i)$, $1 \leqq i \leqq k$ and a is the type $(m \to n)$ where $n = \langle s_1, \cdots, s_k\rangle$ then $\langle \cdots \rangle \in [\mathbb{D}_{a_1} \times \cdots \times \mathbb{D}_{a_k}, \mathbb{D}_a]$.*

Proof. See Appendix. □

In consequent discussions the symbol C will not be used to denote composition. Instead the result of composing g with f will be simply denoted by gf. It is important to note that in the above proposition the types used are of the form $(m \to s)$, except in parts (a), (d). If more general types are allowed, i.e., types of the form $(m \to n)$, the results are no longer true. For example cc $\notin [\mathbb{D}_a, \mathbb{D}_a]$ if $a = (m \to n)$. This will not affect us as all terms have types of the form $(m \to s)$.

A function $f \in [\mathbb{D}_m, \mathbb{D}_s]$ is said to be a *value-function* if, for all $X \in \mathbb{D}_m$, $f(X)$ is a value in $\mathbb{D}_s$. It is said to be a *d-value function* if these values are all defined.

PROPOSITION 3.1.2. *Let f_1, f_2, f_3, f, g, h be functions over $\mathbb{D}$ of the appropriate type.*

(a)

$$\text{(i)} \quad (fg)^r h = f(g^r h),$$

$$\text{(ii)} \quad (f_1 \cup f_2)^r h = f_1^r h \cup f_2^r h,$$

$$\text{(iii)} \quad (\mathrm{if}\,(f_1, f_2, f_3))^r h = \mathrm{if}\,(f_1^r h, f_2^r h, f_3^r h).$$

(b) *If h is a value-function*

$$\text{(i)} \quad (fg)^c h = f(g^c h),$$

$$\text{(ii)} \quad (f_1 \cup f_2)^c g = f_1^c g \cup f_2^c g,$$

$$f^c(g_1 \cup g_2) = f^c g_1 \cup f^c g_2,$$

$$\text{(iii)} \quad (\mathrm{if}\,(f_1, f_2, f_3))^c h = \mathrm{if}\,(f_1^c h, f_2^c h, f_3^c h).$$

(c) If h is a d-value-function then (b) holds with v in place of c.

Proof. See Appendix. □

We can easily find examples of h which do not satisfy the restrictions of parts (b) and (c) and for which these statements are false.

3.2. The mathematical semantics. In this section we use the functionals of § 3.1 to define the semantics of programs and terms.

Referring to program (1) let a_i be the type of F_i and let $\mathbb{D}_p$ be $\mathbb{D}_{a_1} \times \cdots \times \mathbb{D}_{a_k}$. With each term $T \in a = (m \to s)$ we associate a functional $\lambda\langle f_1, \cdots, f_k\rangle.\mathcal{V}[T] \in [\mathbb{D}_p, \mathbb{D}_a]$ as follows:

(i) $T = X_i \in (m \to s)$, $\mathcal{V}[T] = \lambda x \in \mathbb{D}_m.x_i$,

(ii) $T = C \in (m \to s)$, $\mathcal{V}[T] = \lambda x \in \mathbb{D}_m.\{C\}$,

(iii) $T = T_1$ **or** T_2, $\mathcal{V}[T] = \mathcal{V}[T_1] \cup \mathcal{V}[T_2]$,

(iv) $T = G(T_1, \cdots, T_n)$, $\mathcal{V}[T] = g\langle \mathcal{V}[T_1], \cdots, \mathcal{V}[T_n]\rangle$,

(v) $T = F_i^\gamma\langle T_1, \cdots, T_n\rangle$, $\mathcal{V}[T] = f_i^\gamma\langle \mathcal{V}[T_1], \cdots, \mathcal{V}[T_n]\rangle$, $\gamma = r, c, v$.

From Proposition 3.1.1 it follows that $\lambda\langle f_1, \cdots, f_k\rangle.\mathcal{V}[T] \in [\mathbb{D}_p, \mathbb{D}_a]$. Considering the program (1), we let $\mathcal{M}(F_i)$ be the ith component of the least fixpoint of $\lambda\langle f_1, \cdots, f_k\rangle.\langle \mathcal{V}[P_1], \cdots, \mathcal{V}[P_k]\rangle$. Then for any term T we can define $\mathcal{M}(T)$ to be the result of applying the functional $\lambda\langle f_1, \cdots, f_k\rangle.\mathcal{V}[T]$ to the vector of functions $(\mathcal{M}(F_1), \cdots, \mathcal{M}(F_k))$.

Examples. For convenience we let id_i denote the ith projection function and C the function $\lambda x.\{C\}$. The exact types of these functions will be clear from the context.

The program (2) gives rise to the functional

$$\lambda f.\mathrm{if}\ (z\ \mathrm{id}_1, k_0, f^v\langle p\ \mathrm{id}_1, f^c\langle \mathrm{id}_1, \mathrm{id}_2\rangle\rangle)$$

and if T is the term $F^v\langle X_1, X_2\rangle$ then $\mathcal{M}(T)$ is the natural extension of

$$h(x, y) = \begin{cases} k_0 & \text{if } x = k_0, \\ e & \text{if } x = e. \end{cases}$$

The program (3) gives rise to the functional

$$\lambda f.\mathrm{if}(z\mathrm{id}_1, k_0, f^c\langle p\ \mathrm{id}_1, f^c\langle \mathrm{id}_1, \mathrm{id}_2\rangle\rangle)$$

and if T is the term $F^c\langle X_1, X_2\rangle$ then $\mathcal{M}(T)$ is the $\cup$-extension of

$$h(x, y) = \begin{cases} k_0 & \text{if } x = k_m \text{ for any } m, \\ e & \text{if } x = e, \\ \perp & \text{otherwise.} \end{cases}$$

The program (4) gives rise to the functional

$$\lambda f.f^v(id \cup f^v s\ id)$$

and $\mathcal{M}(F^v\langle X\rangle)$ is the natural extension of the relation

$$H = \{\langle k_m, k_n\rangle : m \geqq n\} \cup \{\langle e, e\rangle\}.$$

3.3. Equivalent definitions of $\mathcal{M}$. Let $T \in (m \to s)$ be a P-term. We define $\mathcal{M}_n(T)$, for all $n \geqq 0$, by induction on n.

(i) $n = 0: \mathcal{M}_0(T) = \lambda x \in \mathbb{D}_m.\{\perp_s\}$.

(ii) *Assume* $\mathcal{M}_n(T)$ has been defined for every term T. $\mathcal{M}_{n+1}(T)$ is defined by structural induction on T.

(a) $T = X_i \in (m \to s)$, $\mathcal{M}_{n+1}(T) = \mathrm{id}_i$,

(b) $T = C \in (m \to s)$, $\mathcal{M}_{n+1}(T) = \lambda x \in \mathbb{D}_m.\{C\}$,

(c) $T = T_1$ **or** T_2, $\mathcal{M}_{n+1}(T) = \mathcal{M}_{n+1}(T_1) \cup \mathcal{M}_{n+1}(T_2)$,

(d) $T = \mathrm{IF}\ (T_1, T_2, T_3)$, $\mathcal{M}_{n+1}(T) = \mathrm{if}\ (\mathcal{M}_{n+1}(T_1), \mathcal{M}_{n+1}(T_2), \mathcal{M}_{n+1}(T_3))$,

(e) $T = G\langle T_1, \cdots T_k\rangle$, $\mathcal{M}_{n+1}(T) = g^v\langle \mathcal{M}_{n+1}(T_1), \cdots, \mathcal{M}_{n+1}(T_k)\rangle$,

(f) $T = F_i^\gamma\langle T_1, \cdots, T_k\rangle$, $\mathcal{M}_{n+1}(T) = \mathcal{M}_n(P_i)^\gamma\langle \mathcal{M}_n(T_1), \cdots, \mathcal{M}_n(T_k)\rangle$, $\gamma = r, c, v$.

LEMMA 3.2.1. $\mathcal{M}_n(T) \leqq \mathcal{M}_{n+1}(T)$.

Proof. Since all of the functionals preserve $\leqq$ the proof is trivial. □

PROPOSITION 3.2.2. $\mathcal{M}(T) = V\{\mathcal{M}_n(T) : n \geqq 0\}$.

Proof. Let $\mathcal{M}(P)$ represent $\mathcal{M}(P_1), \cdots, \mathcal{M}(P_k)$ and similarly for $\mathcal{M}_n(P)$ and $\mathcal{V}[P]$. Also let $\mathbf{f}$ represent $\langle f_1, \cdots, f_k\rangle$. A simple proof by structural induction on T shows that $\mathcal{M}_{n+1}(T) = \lambda \mathbf{f}.\mathcal{V}[T](\mathcal{M}_n(P))$.

Now for any cpo $\langle D, \leqq\rangle$ and any continuous function $f : D \to D$ the least fixpoint of f is $V\{f_n : n \geqq 0\}$, where $f_0 = \perp$ and $f_{n+1} = f(f_n)$ (see [11]). If we apply this result to the functional $\lambda f.\mathcal{V}[P]$ we get that $\mathcal{M}(P) = V\{\mathcal{M}_n(P) : n \geqq 0\}$. Therefore

$$\begin{aligned}\mathcal{M}(T) &= \lambda \mathbf{f}.\mathcal{V}[T](\mathcal{M}(P))\\ &= \lambda \mathbf{f}.\mathcal{V}[T](V\{(\mathcal{M}_n(P)) : n \geqq 0\})\\ &= V\{\lambda \mathbf{f}.\mathcal{V}[T](\mathcal{M}_n(P)) : n \geqq 0\}\\ &= V\{\mathcal{M}_{n+1}(P) : n \geqq 0\}. \qquad \square\end{aligned}$$

Another useful characterization of $\mathcal{M}(T)$ is given by considering a sequence $\{T^n : n \geqq 0\}$ of recursion-free programs which approximate the behavior of T. Roughly speaking T^n is the result of restricting the depth of recursive nesting in T to at most n levels. If a call to a procedure is made at a lower level then the 'value' $\perp$ is returned. For an exact definition of T^n we must introduce into the syntax of the programming language the distinguished function symbol $\Delta \in a$ for every type a and the distinguished data-constant $\perp \in \mathrm{K}^s$ for every s. The evaluation mechanism is then augmented by the rules:

(i) $\Delta\langle T_1, \cdots, T_k\rangle \overset{\circ}{\rightarrow} \perp$ for every sequence $T_1, \cdots, T_k$,
(ii) $G\langle T_1, \cdots, T_k\rangle \overset{\circ}{\rightarrow} \perp$ if any $T_i = \perp, 1 \leqq i \leqq k$,
(iii) $F^v\langle T_1, \cdots, T_k\rangle \overset{\circ}{\rightarrow} \perp$ if any $T_i = \perp, 1 \leqq i \leqq k$.

The definition of $\mathcal{V}$ is also extended by defining

$$\mathcal{V}[\Delta\langle T_1, \cdots, T_k\rangle] = \lambda x \in \mathbb{D}_{m}.\{\perp\} \quad \text{if } \Delta \in (m \to s).$$

Given the program P (see (1)) and the P-program term T, we define, for every $n \geqq 0$, the program P^n and the terms T^n as follows:

(i) $T^0 = \Delta\langle \perp, \cdots, \perp\rangle$. P^0 is the program

$$\begin{aligned}&F_{1,1}\langle X_1, \cdots, X_{n_1}\rangle \Leftarrow P_1^0,\\ &\vdots\\ &F_{k,1}\langle X_1, \cdots, X_{n_k}\rangle \Leftarrow P_k^0.\end{aligned}$$

(ii) Assume T^n and P^n have been defined. The P^n-program term T^{n+1} *is defined by structural induction*:

(a) if T contains no occurrences of any F_i, $T^{n+1} = T$,
(b) $T = F_i^\gamma\langle T_1, \cdots, T_k\rangle$, $T^{n+1} = F_{i,n+1}^\gamma\langle T_1^n, \cdots, T_k^n\rangle$,
(c) $T = T_1$ **or** T_2, $T^{n+1} = T_1^{n+1}$ **or** T_2^{n+1},
(d) $T = G(T_1, \cdots, T_k)$, $T^{n+1} = G(T_1^{n+1}, \cdots, T_k^{n+1})$,
(e) $T = \mathrm{IF}\,(T_1, T_2, T_3)$, $T^{n+1} = \mathrm{IF}\,(T_1^{n+1}, T_2^{n+1}, T_3^{n+1})$.

P^{n+1} is the program obtained from P^n by adding the k equations

$$\begin{aligned}&F_{1,n+2}\langle X_1, \cdots, X_{n_1}\rangle \Leftarrow P_1^{n+1},\\ &\vdots\\ &F_{k,n+2}\langle X_1, \cdots, X_{n_k}\rangle \Leftarrow P_k^{n+1}.\end{aligned}$$

Example. Let P be the program

$$F_1\langle X\rangle \Leftarrow G(X, F_2^v\langle X, X\rangle),$$

$$F_2\langle X, Y\rangle \Leftarrow H(F_1^v\langle F_2^c\langle X, Y\rangle\rangle, X).$$

Then P^1 is the program

$$\begin{aligned}
&F_{1,2}\langle X\rangle \Leftarrow G(X, F^v_{2,1}\langle X, X\rangle),\\
&F_{2,2}\langle X, Y\rangle \Leftarrow H(F^v_{1,1}\langle \Delta\langle \bot, \bot\rangle\rangle, X),\\
&F_{1,1}\langle X\rangle \Leftarrow \Delta\langle \bot\rangle,\\
&F_{2,1}\langle X, Y\rangle \Leftarrow \Delta\langle \bot, \bot\rangle.
\end{aligned}$$

PROPOSITION 3.2.3. $\mathcal{M}(T) = V\{\mathcal{M}(T^n): n \geqq 0\}$.

Proof. From Proposition 3.2.2 it suffices to prove that $\mathcal{M}_n(T) = \mathcal{M}(T^n)$, $\forall n \geqq 0$. This is proven by induction on n. For $n = 0$ the proof is trivial and the case $k+1$ is proven by structural induction on T. We consider only one of the various cases. The others are similar. Suppose $T = F^\gamma_i\langle T_1, \cdots, T_m\rangle$. Then

$$\begin{aligned}
\mathcal{M}_{k+1}(T) &= \mathcal{M}_k(P_i)^\gamma\langle \mathcal{M}_k(T_1), \cdots, \mathcal{M}_k(T_m)\rangle\\
&= \mathcal{M}(P^k_i)^\gamma\langle \mathcal{M}(T^k_1), \cdots, \mathcal{M}(T^k_m)\rangle \quad \text{by induction}\\
&= \mathcal{M}(F^\gamma_{1,k+1}\langle T^k_1, \cdots, T^k_m\rangle) \quad \text{see Lemma 4.1.1}\\
&= \mathcal{M}(T^{k+1}). \quad \Box
\end{aligned}$$

4. Investigation of the mathematical semantics.

4.1. Properties of the model. In this subsection we give some useful properties of the mathematical model. For convenience we let T represent a sequence of terms $T_1, \cdots, T_k$ and X the sequence $X_1, \cdots, X_k$. Thus $[T|X]S$ denotes the result of substituting T_i for X_i in S, $1 \leqq i \leqq k$. Also $\mathcal{M}(T)$ will denote $\langle \mathcal{M}(T_1), \cdots, \mathcal{M}(T_k)\rangle$.

LEMMA 4.1.1. (application lemma).

(i) $\mathcal{M}(F^\gamma_i\langle T\rangle) = \mathcal{M}(P_i)^\gamma\mathcal{M}(T)$, $\gamma = r, c, v$.

(ii) $\mathcal{M}(G(T)) = g^v\mathcal{M}(T)$.

(iii) $\mathcal{M}(\text{IF }(T_1, T_2, T_3)) = \text{if }(\mathcal{M}(T_1), \mathcal{M}(T_2), \mathcal{M}(T_3))$.

(iv) $\mathcal{M}(T_1 \textbf{ or } T_2) = \mathcal{M}(T_1) \cup \mathcal{M}(T_2)$.

Proof. (i)

$$\begin{aligned}
&\mathcal{M}(F^\gamma_i\langle T\rangle)\\
&\quad = V\{\mathcal{M}_{n+1}(F^\gamma_i\langle T\rangle): n \geqq 0\} && \text{from Proposition 3.2.2}\\
&\quad = V\{\mathcal{M}_n(P_i)^\gamma\mathcal{M}_n(T): n \geqq 0\} && \text{by definition}\\
&\quad = (V\{\mathcal{M}_n(P_i): n \geqq 0\})^\gamma V\{\mathcal{M}_n(T): n \geqq 0\} && \text{since the } \gamma \text{ function preserves } V\\
&\quad = \mathcal{M}(P_i)^\gamma\mathcal{M}(T) && \text{from Proposition 3.2.2.} \quad \Box
\end{aligned}$$

Parts (ii), (iii) and (iv) are similar.

LEMMA 4.1.2 (substitution lemma). *If T_i, $1 \leqq i \leqq k$ are constant terms and S is any term then*

(i) $\mathcal{M}([T|X]S) = \mathcal{M}(S)^r\mathcal{M}(T)$,

(ii) $\mathcal{M}([T|X]S) = \mathcal{M}(S)^c\mathcal{M}(T)$ *if T_i are deterministic,* $1 \leqq i \leqq k$,

(iii) $\mathcal{M}([T|X]S) = \mathcal{M}(S)^v\mathcal{M}(T)$ *if T_i are data-symbols,* $1 \leqq i \leqq k$.

Proof. For the purposes of this proof we let $\mathcal{M}(L)$ be denoted by l, for any symbol L representing a term. We prove only part (ii).

(ii) We can easily show by structural induction that if T_i is deterministic and constant then t_i is a value-function. We omit the proof. We use structural induction on S

to prove the lemma.

(a) $S = C$, C a data-symbol. Then $s^c t = \lambda x.\{C\} = \mathcal{M}([T\,|\,X], S)$. Similarly if $S = X_i$.

(b) $S = S_1$ **or** S_2. Then

$$\begin{aligned} s^c t &= (s_1 \cup s_2)^c t && \text{from Lemma 4.1.1 (iv)} \\ &= s_1^c t \cup s_2^c t && \text{from Proposition 3.1.2 (b), (ii)} \\ &= \mathcal{M}([T|X]S_1) \cup \mathcal{M}([T|X]S_2) && \text{by induction hypothesis} \\ &= \mathcal{M}([T|X]S) && \text{from Lemma 4.1.1 (iv)} \end{aligned}$$

(c) $S = \text{IF}\ (S_1, S_2, S_3)$. Then

$$\begin{aligned} s^c t &= (\text{if}\ (s_1, s_2, s_3))^c t && \text{from Lemma 4.1.1 (iii)} \\ &= \text{if}\ (s_1^c t, s_2^c t, s_3^c t) && \text{from Proposition 3.1.2 (b), (iii)} \\ &= \text{if}\ (\mathcal{M}([T|X]S_1), \mathcal{M}([T|X]S_s), \mathcal{M}([T|X]S_3)) \\ & && \text{by induction hypothesis} \\ &= \mathcal{M}([T|X]S) && \text{from Lemma 4.1.1 (iii).} \end{aligned}$$

(d) $S = F_i^\gamma \langle S_1 \rangle$, where S_1 is a sequence of terms. Then

$$\begin{aligned} s^c t &= (p_i^\gamma s_1)^c t && \text{from Lemma 4.1.1 (i)} \\ &= p_i^\gamma (s_1^c t) && \text{from Proposition 3.1.2 (b), (i)} \\ & && \text{since } t \text{ is a value-function} \\ &= p_i \mathcal{M}([T|X]S_1) && \text{by induction hypothesis} \\ &= \mathcal{M}([T|X]S) && \text{from Lemma 4.1.1 (i).} \end{aligned}$$

(e) $S = G(S_1)$, where S_1 is a sequence of terms. The proof is similar to part (d). □

It should be pointed out that the substitution lemma is not in general true if the T_i are not constant. For example if S is the term IF $(Z(X), X, k_2)$ and T the term X, then $\mathcal{M}([T|X]S) \neq \mathcal{M}(S)^c \mathcal{M}(T)$.

As a simple consequence of these two results we have that the mathematical model is at least consistent with the evaluation mechanism.

PROPOSITION 4.1.3. *For any two constant terms T_1, T_2, $T_1 \to T_2$ implies $\mathcal{M}(T_1) \supseteq \mathcal{M}(T_2)$.*

Proof. If $T_1 \overset{\circ}{\to} T_2$ then we can prove by structural induction on T_1 that $\mathcal{M}(T_1) \supseteq \mathcal{M}(T_2)$. Four examples of the various cases are given. The remainder are similar.

(i) Suppose T_1 is IF (t, S_1, S_2). Then T_2 is S_1. Therefore

$$\begin{aligned} \mathcal{M}(T_1) &= \mathcal{M}(\text{IF}\ (t, S_1, S_2)) \\ &= \text{if}\ (\mathcal{M}(t), \mathcal{M}(S_1), \mathcal{M}(S_2)) && \text{from Lemma 4.1.1 (iii)} \\ &= \mathcal{M}(S_1) && \text{by the definition of 'if'.} \end{aligned}$$

(ii) Suppose T_1 is IF (T, S_1, S_2) and T_2 is IF (T', S_1, S_2). Then by definition $T \overset{\circ}{\to} T'$. So by induction hypothesis $\mathcal{M}(T) \supseteq \mathcal{M}(T')$. Therefore

$$\begin{aligned} \mathcal{M}(T_1) &= \text{if}\ (\mathcal{M}(T), \mathcal{M}(S_1), \mathcal{M}(S_2)) && \text{from Lemma 4.1.1 (iii)} \\ &\supseteq \text{if}\ (\mathcal{M}(T'), \mathcal{M}(S_1), \mathcal{M}(S_2)) && \text{since if preserves } \supseteq \\ &= \mathcal{M}(T_2) && \text{from Lemma 4.1.1 (iii).} \end{aligned}$$

(iii) Suppose T_1 is $F_i^\gamma\langle S\rangle$ where S is a sequence of terms and T_2 is $[S|X]P_i$. Then

$$\begin{aligned}\mathcal{M}(T_1)&=\mathcal{M}(P_i)^\gamma\mathcal{M}(S) && \text{from Lemma 4.1.1 (i)}\\ &=\mathcal{M}([S|X]P_i) && \text{by applying the appropriate section of Lemma 4.1.2.}\end{aligned}$$

(iv) Suppose T_1 is S_1 **or** S_2 and T_2 is S_2. Then

$$\begin{aligned}\mathcal{M}(T)&=\mathcal{M}(S_1)\cup\mathcal{M}(S_2) && \text{from Lemma 4.1.1 (iv)}\\ &\supseteq\mathcal{M}(S_2) && \text{from the definition of '}\cup\text{'.}\end{aligned}$$

Therefore $T_1\overset{\circ}{\to}T_2$ implies $\mathcal{M}(T_1)\supseteq\mathcal{M}(T_2)$. The proposition now follows by induction on the length of the derivation from T_1 to T_2. □

4.2. Characterization of equality in the model. Before considering which terms are given the same denotation via the mathematical semantics it is convenient to determine exactly the nature of $\mathcal{M}(T)$ for every constant term T. For every such term $T\in(m\to s)$, $\mathcal{M}(T)$ is always of the form $\lambda x\in\mathbb{D}_m.\alpha$ where $\alpha\in\mathbb{D}_s$. For convenience we will identify α and $\lambda x\in\mathbb{D}_m.\alpha$.

For any constant term T let EVAL $(T)=\{\mathrm{v}\in\mathrm{D_s}|\mathrm{T}\to\mathrm{v}\}$.

LEMMA 4.2.1. EVAL $(T^n)=\mathcal{M}(T^n)$, $\forall n\geqq 0$.

Proof. See Appendix. □

For any constant term T define LIMEVAL (T) by

(i) $C\in$ LIMEVAL (T) if $T^n\to C$ for some $n\geqq 0$,

(ii) $\perp\in$ LIMEVAL (T) if $T^n\to\perp$ for every $n\geqq 0$.

Thus LIMEVAL (T) contains all those data-constants which can be derived from the finite approximations to T together with $\perp$ if T can give rise to sequences of recursive calls of arbitrary length.

PROPOSITION 4.2.2. $\mathcal{M}(T)=$ LIMEVAL (T).

Proof.

$$\begin{aligned}C\in\text{LIMEVAL }(T)&\Leftrightarrow C\in\text{EVAL }(T^k) && \text{for some } k\\ &\Leftrightarrow C\in\mathcal{M}(T^k) && \text{from previous lemma}\\ &\Leftrightarrow C\in\mathcal{M}(T) && \text{from Proposition 3.2.3.}\\ \perp\in\text{LIMEVAL }(T)&\Leftrightarrow\perp\in\text{EVAL }(T^k) && \text{for every } k\\ &\Leftrightarrow\perp\in\mathcal{M}(T^k) && \text{for every } k \text{ from previous lemma}\\ &\Leftrightarrow\perp\in\mathcal{M}(T) && \text{from Proposition 3.2.3.}\end{aligned}$$ □

We can now consider what it means behaviorally for two terms to receive the same denotation, i.e., for $\mathcal{M}(T_1)$ to be equal to $\mathcal{M}(T_2)$. A natural definition of behavioral or operational equivalence which has been used in [6], [14], [20], is to say that two terms are operationally equivalent if and only if it is impossible to distinguish between them using the evaluation mechanism. Because of the nondeterminism it is necessary to take into consideration the ability of a term to diverge. If we use LIMEVAL then this ability will be taken into account via the presence or absence of $\perp$ because $\perp\in$ LIMEVAL (T) if T can give rise to arbitrary long sequences of recursive calls. This notion of operational equivalence may be formalized as follows:

For T_1, $T_2\in(m\to s)$ we say $T_1\equiv_{\mathrm{op}}T_2$ if for every sequence of constants terms S

$$\text{LIMEVAL }([S|X]T_1)=\text{LIMEVAL }([S|X]T_2).$$

THEOREM 4.2.3. $\mathcal{M}(T_1)=\mathcal{M}(T_2)\Leftrightarrow T_1\equiv_{\mathrm{op}}T_2$.

Proof. First suppose $\mathcal{M}(T_1)\neq\mathcal{M}(T_2)$. Then from Lemma 1.5 there exists a finite X_f such that $\mathcal{M}(T_1)(X_f)\neq\mathcal{M}(T_2)(X_f)$. Now for any finite set $F\in\mathbb{D}_s$ there exists a constant term S such that $\mathcal{M}(S)=F$. Now X_f above is of the form $\langle X_f^1,\cdots,X_f^m\rangle$, where each X_f^i is finite. Let S^i be the term corresponding to X_f^i and let $S=\langle S^1,\cdots,S^m\rangle$. Then $\mathcal{M}(T_1)'\mathcal{M}(S)\neq\mathcal{M}(T_2)'\mathcal{M}(S)$. From Lemma 4.1.2 $\mathcal{M}([S|X]T_1)\neq\mathcal{M}([S|X]T_2)$. It follows from Proposition 4.2.2 and the definition of $\equiv_{\text{op}}$ that $T_1\not\equiv_{\text{op}}T_2$.

Conversely suppose $T_1\not\equiv_{\text{op}}T_2$. Then for some vector of constant terms S, LIMEVAL $([S|X]T_1)\neq$ LIMEVAL $([S|X]T_2)$. By Proposition 4.2.2, $\mathcal{M}([S|X]T_1)\neq\mathcal{M}([S|X]T_2)$. Using Lemma 4.1.2 we see that $\mathcal{M}(T_1)'\mathcal{M}(S)\neq\mathcal{M}(T_2)'\mathcal{M}(S)$. Since S is a vector of constant terms $\mathcal{M}(S)=\lambda x.\alpha_s$ where $\alpha_s\in\mathbb{D}_m$. It follows that $\mathcal{M}(T_1)(\alpha_s)\neq\mathcal{M}(T_2)(\alpha_s)$, i.e., $\mathcal{M}(T_1)\neq\mathcal{M}(T_2)$. □

4.3. Discussion of operational equivalence. The use of LIMEVAL in the definition of operational equivalence may be justified by the observation that in any particular implementation of recursive languages there is a bound on the depth of recursion allowed. Thus LIMEVAL captures the behavior of programs over all possible implementations.

However, a more natural definition of operational equivalence could be given using the sets IEVAL (T) where IEVAL $(T)=$ EVAL $(T)\cup\{\perp|T$ can diverge$\}$. It is easy to show that in the absence of call-time choice IEVAL $(T)=$ LIMEVAL (T). But when call time choice is allowed then there is a difference.

Example.

$$F_1\langle X, Y\rangle\Leftarrow X \textbf{ or } Y,$$

$$F_2\langle X\rangle\Leftarrow F_2^v\langle X\rangle \textbf{ or } F_2^v\langle X\rangle.$$

Then consider the term $T=F_1^c\langle k_0, F_2^v\langle k_0\rangle\rangle$. Because $F_2^v\langle k_0\rangle$ can never be reduced to a deterministic term IEVAL $(T)=\{\perp\}$ whereas LIMEVAL $(T)=\{\text{k},\perp\}$. The latter follows because $\Delta\{\perp\}$ is deterministic.

The problem arises because certain procedures P_i which, when using call-time choice, do not actually use all of their parameters and therefore whether or not some such parameter is deterministic or not is immaterial. However, in our particular implementation of call-time choice these T_j must be reduced to a deterministic term even if they are not required. A solution is to implement call-time choice in a slightly different manner such that this problem does not arise. One such implementation is based on the delay rule in [19]. Here when evaluating $F_i^c\langle T\rangle$ a pointer to the storage for T is passed to the procedure P_i and if P_i needs a value for T it uses the pointer to evaluate it. However the difference between this and run-time choice is that here T is only ever evaluated once and if P_i subsequently needs a value for T the same value is used. In short, in run-time choice many copies of T are used whereas in compile-time choice only one copy is used.

We omit the proof that in this implementation of compile-time choice LIMEVAL coincides with the more natural IEVAL, as this implementation will be used in a more general setting in [8].

Appendix.

Definition of deterministic. Consider the program (1). F_i is said to be *directly dependent* on F_j if $i\neq j$ and F_j occurs in P_i. Let *dependent* be the transitive closure of directly dependent. F_i is said to be *directly nondeterministic* if '**or**' occurs in P_i or if some constant function symbol G occurs in P_i such that $E(G)$ is not a function. F_i is *nondeterministic* if it is directly nondeterministic or if there is some F_j on which F_i is dependent such that F_j is directly nondeterministic. A term T is nondeterministic if '**or**' occurs in T, if some nondeterministic F_i occurs in T, or if some G, such that $E(G)$ is not a function, occurs in T. T is *deterministic* if it is not nondeterministic.

Proof of Proposition 3.1.1. These results follow directly and laboriously from the various definitions. The only slight difficulty occurs in proving $\mathrm{cc} \in [\mathbb{D}_a, \mathbb{D}_a]$ and as an example we prove this.

We first show that if $f \in \mathbb{D}_a$ then $f^c \in \mathbb{D}_a$. For this it is sufficient to prove that (i) if $X, Y \in \mathbb{D}_m$ and $X \subseteq Y$ then $f^c(X) \subseteq f^c(Y)$ and (ii) if $X = \bigvee\{X_n\}$ in $\mathbb{D}_m$ then $f^c(X) = \bigvee\{f(X_n)\}$.

(i) Let $v \in f^c(X)$. Then there exists a $v' \in X$ such that $v \in f(v')$. Since $X \subseteq Y$, $v' \in Y$ and it follows from the definition of cc that $v \in f(Y)$. Therefore $f(X) \subseteq f(Y)$.

(ii)

$$\begin{aligned} v \in f^c(X) &\Leftrightarrow v \in f(v') && \text{for some } v' \in \bigvee\{X_n\} \\ &\Leftrightarrow v \in f(v') && \text{for some } v' \in X_n,\ \forall n \geqq N, \text{ for some } N \\ &\Leftrightarrow v \in f^c(X_n) && \forall n \geqq N, \text{ for some } N \\ &\Leftrightarrow v \in \bigvee\{f^c(X_n)\}. \end{aligned}$$

It remains to show that cc is $\subseteq$-monotonic and $\leqq$-continuous. The $\subseteq$-monotonicity is similar to (i) above and is omitted.

Let $f = \bigvee\{f_n\}$ in $\mathbb{D}_a$. We must show that $f^c = \bigvee\{f_n^c\}$. Let $X \in \mathbb{D}_m$ and $v \in f^c(X)$. Then there exists a $v' \in X$ such that $v \in f(v')$. But $f(v') = \bigvee\{f_n(v')\}$. Therefore there exists and N such that $v \in f_n(v')$ $\forall n \geqq N$. That is $v \in f^c(X)$ $\forall n \geqq N$. Therefore $v \in \bigvee\{f_n^c\}(X)$. Conversely suppose $v \in \bigvee\{f_n^c\}(X)$. If v is defined (i.e. $v \neq \perp$) then $v \in f_n^c(X)$ *for some n*. Thus $v \in f_n(v')$ for some $v' \in X$. Since $f_n \leqq f$ it follows that $v \in f(v')$ and therefore $v \in f^c(X)$. If $v = \perp$ then $v \in f_n^c(X)$ $\forall n \geqq 0$. Therefore for each n there exists a v^n such that $v \in f_n(v^n)$. For any element α of $\mathbb{D}_m$ let α_i be its ith component. We define the new value v' as follows: if $\perp \in X_i$ let $v_i' = \perp$. Otherwise X_i is finite. Therefore there must exist at least one c such that $v_i^n = c$ for infinitely many n. Let $v_i' = c$. Then $v' \leqq v^n$ for infinitely many n. Therefore $f_n(v') \leqq f_n(v^n)$ for infinitely many n. It follows that $\perp \in f_n(v')$ for infinitely many n and since $v' \in X$, $\perp \in f^c(x)$. □

Proof of Proposition 3.1.2. The proofs of the three parts (a), (b) and (c) are very similar and follow more or less directly from the definitions. Accordingly as examples we prove (c), (i) and (iii).

(c) (i) Let $X \in \mathbb{D}_m$ and let w denote the defined value $h(X)$. Then

$$\begin{aligned} v \in (fg)^v h(X) &\Leftrightarrow v \in (fg)^v(w) \\ &\Leftrightarrow v \in f(g(w)) \\ &\Leftrightarrow v \in f(g^v(w)) \\ &\Leftrightarrow v \in f(g^v h)X. \end{aligned}$$

Therefore $(fg)^v h = f(g^v h)$.

(c) (iii). Let X and w be as above. Then

$$\begin{aligned} & v \in \mathrm{if}\ (f_1^v h, f_2^v h, f_3^v h)(X) \\ &\Leftrightarrow v \in \mathrm{dif}\ (x_1, x_2, x_3) \quad \text{for some } x_i \in (f_i^v h)(X) \\ &\Leftrightarrow v \in \mathrm{dif}\ (x_1, x_2, x_3) \quad \text{for some } x_i \in f_i(w) \\ &\Leftrightarrow v \in \mathrm{if}\ (f_1, f_2, f_3)(w) \\ &\Leftrightarrow v \in \mathrm{if}\ (f_1, f_2, f_3)^v h(X). \end{aligned}$$

□

Proof of Lemma 4.2.1. Suppose $v \in \mathrm{EVAL}\,(T^n)$, $n \geqq 0$. Then $T^n \rightarrow v$. Therefore from Proposition 4.1.3, $\mathcal{M}(T^n) \supseteq \mathcal{M}(v) = \{v\}$. So $\mathcal{M}(T^n) \supseteqq \mathrm{EVAL}\,(T^n)$. To prove the converse we need a complicated induction hypothesis. A constant term T is said to be *okay* if $\mathrm{EVAL}\,(T) \supseteqq \mathcal{M}(T)$. Let Prop (n) be the property: for all terms $T_1, \cdots, T_k$ which are okay and any term S, $[T_1|X_1, \cdots, T_k|X_k]S^n$ is okay. We prove Prop (n) true by induction on n. The result will follow because if S is a constant term then $S = [T_1|X_1, \cdots T_k|X_k]S^n$. As usual the proof of Prop (0) is trivial. Assuming Prop (n) is true we prove Prop$(n+1)$ is true by structural induction on S. We will use the notational conventions introduced in the beginning of § 4.1.

(i) $S = X_i$. Then $[T|X]S^{n+1} = T_i$ and is therefore okay.

(ii) $S = G(S_1, \cdots, S_m)$. Then $[T|X]S^{n+1} = G([T|X]S_1^{n+1}, \cdots, [T|X]S_m^{n+1})$ and by induction $[T|X]S_i^{n+1}$ is okay. Now suppose $v \in ([T|X]S^{n+1})$. There are two cases:

(a) there exists values $v_1, \cdots, v_m$, $v_i \in \mathcal{M}([T|X]S_i^{n+1})$ and $v \in g(v_1, \cdots, v_m)$. Since $[T|X]S_i^{n+1}$ is okay $[T|X]S_i^{n+1} \rightarrow v_i$ and therefore $[T|X]S^{n+1} \rightarrow v$.

(b) $\perp \in ([T|X]S^{n+1})$ because $\perp \in ([T|X]S_i^{n+1})$ for some i. Then $[T|X]S_i^{n+1} \rightarrow \perp$ and therefore $[T|X]S^{n+1} \hookrightarrow \perp$.

(iii) $S = S_1$ **or** S_2. *The proof is similar to* (ii).

(iv) $S = F_i^\gamma(S_1, \cdots, S_m)$. There are three cases depending on what γ is.

(a) $\gamma =$ 'r'. Then $[T|X]S^{n+1} = F_{i,n+1}^r([T|X]S_1^n, \cdots [T|X]S_m^n)$. Therefore

$$\begin{aligned}\mathcal{M}([T|X]S^{n+1}) &= \mathcal{M}(P_i^n)^r(\mathcal{M}([T|X]S_1^n \cdots)) \\ &= \mathcal{M}([[[T|X]S_1^n|X_1], \cdots, [[T|X]S_m^n|X_m]]P_i^n) \quad \text{by Lemma 4.1.2 (i).}\end{aligned}$$

Now because Prop (n) is true $[T|X]S_i^n$ is okay. So, again using Prop (n), $[[\mathrm{T}|\mathrm{X}]\mathrm{S}_1|\mathrm{X}_1, \cdots]\mathrm{P}_\mathrm{i}^\mathrm{n}$ is okay. Now if $v \in ([T|X]S^{n+1})$ then $v \in ([[T|X]S_1|X_1, \cdots]P_i^n)$. Therefore $[[T|X]S_1|X_1, \cdots]P_i^n \rightarrow v$. It follows that $[T|X]S \rightarrow v$ since $[T|x]S \rightarrow [[T|X]S_1|X_1, \cdots]P_i^n$.

(b) $\gamma =$ 'v'. In this case we have that $\mathcal{M}([T|X]S^{n+1} = \mathcal{M}(P^v(\mathcal{M}([T|X]S_1^n, \cdots)$. Now suppose $v \in \mathcal{M}([T|X]S^{n+1})$. If there exists defined values $v_1, \cdots, v_n$ such that $v_i \in \mathcal{M}([T|X]S_i^n)$ and $v \in \mathcal{M}(P_i^n)\langle v_1, \cdots, v_m\rangle$, then using a proof similar to part (a) we can prove that $F_{i,n+1}^v\langle v_1, \cdots, v_m\rangle \rightarrow v$. Also by induction $[T|X]S_i^n \rightarrow v_i$. Therefore $[T|X]S^{n+1} \rightarrow v$. Otherwise for some i, $\perp \in \mathcal{M}([T|X]S_i^n)$. Then $\mathcal{M}([T|X]S) = \perp$ and by induction $[T|X]S_i^n \rightarrow \perp$. It follows that $[T|X]S \rightarrow \perp$.

(c) $\gamma =$ 'c'. The proof is similar to part (b). □

Acknowledgments. The author would like to thank E. A. Ashcroft under whose supervision this work was carried out. Thanks are also due to E. Egli whose ideas on nondeterminism as expressed in [5] are used throughout the paper and to J. Morris whose comments are incorporated in § 4.3. Finally I would like to thank M. Wang for typing the manuscript.

REFERENCES

[1] J. W. DE BAKKER, *Least fixpoints revisited,* Theoret. Comput. Sci., 2 (1976).

[2] J. W. DE BAKKER AND W. P. DE ROEVER, *A Calculus for recursive program schemes,* Automata, Languages and Programming, M. Nivat, ed., North-Holland, Amsterdam, 1972.

[3] J. M. CADIOU, *Recursive definitions of partial functions and their computations,* Ph.D. thesis, Stanford Univ., Stanford, CA, 1972.

[4] E. W. DIJKSTRA, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] H. EGLI, *A mathematical model for non-deterministic computations,* unpublished.

[6] M. HENNESSY AND E. A. ASHCROFT, *The semantics of non-determinism,* Third International Colloquium on Automata, Languages and Programming, Edinburgh, 1976.

[7] ———, *Parameter-passing mechanisms and non-determinism,* Ninth Annual ACM Symposium on the Theory of Computing, Boulder, 1977.

[8] M. HENNESSY, *Inside-out and outside-in evaluations*, to appear.

[9] ———, *A formal approach to parameter-passing mechanisms and non-determinism*, Ph.D. thesis, University of Waterloo, Waterloo, Ontario, 1976.

[10] M. HENNESSY AND E. A. ASHCROFT, *The semantics of a non-deterministic types λ-calculus*, Theoret. Comput. Sci., to appear.

[11] Z. MANNA, *The correctness of non-deterministic programs*, Artificial Intelligence, 1 (1970), pp. 1–26.

[12] ———, *The Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[13] Z. MANNA AND J. VUILLEMIN, *Fixpoint approach to the theory of computation*, Comm. ACM, 15 (1972), pp. 528–536.

[14] N. NEWEY, *Axioms and theorems for integers, lists and finite sets in L.C.F.*, Stanford memo AIM-184, Stanford Univ., Stanford, CA, 1972.

[15] G. D. PLOTKIN, *L.C.F. considered as a programming language*, Theoret. Comput. Sci., 1977, to appear.

[16] ———, *A powerdomain construction*, this Journal, 5 (1976) pp. 452–487.

[17] W. P. DE ROEVER, *Call-by-value versus call-by-name: a proof theoretic comparison*, Lecture Notes in Computer Science Vol. 28, Springer-Verlag, New York, 1975.

[18] ———, *First-order reduction of call-by-name to call-by-value*, Lecture Notes in Computer Science, Vol. 32, Springer-Verlag, 1975.

[19] D. SCOTT, *Outline of a mathematical theory of computation*, Oxford Mon. PRG-2, Oxford University Press, Oxford, England, 1970.

[20] J. VUILLEMIN, *Correct and optimal implementations of recursion in a simple programming language*, J. Comput. Systems Sci., 9 (1974), pp. 332–354.

[21] C. P. WADSWORTH, *The relation between computational and denotational properties for Scott's D_∞-models of the lambdacalculus*, this Journal, 5 (1976), pp. 488–521.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

ON THE OPTIMALITY OF LINEAR MERGE*

PAUL K. STOCKMEYER† AND F. FRANCES YAO‡

Abstract. Let $M(m, n)$ be the minimum number of pairwise comparisons which will always suffice to merge two linearly ordered lists of lengths m and n. We prove that $M(m, m+d)=2m+d-1$ whenever $m \geqq 2d-2$. This generalizes earlier results of Graham and Karp ($d=1$), Hwang and Lin ($d=2, 3$), Knuth ($d=4$), and shows that the standard linear merging algorithm is optimal whenever $m \leqq n \leqq \lfloor 3m/2 \rfloor + 1$.

Key words. merging, algorithm, paired comparisons

1. Introduction. Suppose we are given two linearly ordered sets A and B consisting of elements

$$a_1 < a_2 < \cdots < a_m$$

and

$$b_1 < b_2 < \cdots < b_n$$

respectively, where the $m+n$ elements are distinct. The problem of merging these sets into a single ordered set by means of a sequence of pairwise comparisons is of obvious practical interest, and several algorithms have been devised for handling it.

An intriguing theoretical problem is to determine $M(m, n)$, the minimum number of comparisons which will always suffice to merge the sets in a decision tree model [5]. Evaluating this function in general seems quite difficult, and values are known for only a few special cases, including $m \leqq 3$ ([1], [2], and [4]). In one direction, an upper bound for $M(m, n)$ is provided by a simple procedure variously referred to as the normal, standard, linear, or tape merge algorithm. Here the two smallest elements (initially a_1 and b_1) are compared, and the smaller of these is deleted from its list and placed on an output list. The process is repeated until one list is exhausted. It is easy to see that this algorithm requires $m+n-1$ comparisons in the worst case, so that

$$M(m, n) \leqq n+m-1.$$

Although better algorithms exist for many cases, R. L. Graham and R. M. Karp independently observed that this algorithm is optimal when $|n-m|$ is 0 or 1. That is, they showed that

$$M(m, m) = 2m-1$$

and

$$M(m, m+1) = 2m.$$

Later Hwang and Lin [3] proved that

$$M(m, m+2) = 2m+1 \quad \text{for } m \geqq 2$$

and

$$M(m, m+3) = 2m+2 \quad \text{for } m \geqq 4,$$

* Received by the editors June 2, 1978.

† Department of Mathematics and Computer Science, College of William and Mary, Williamsburg, Virginia 23185. The work of this author was supported in part by the Office of Naval Research under Contract N00014-76-C-0673, NR044-459, and by the William and Mary faculty research leave program.

‡ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by National Science Foundation Grant MCS-77-05313.

while Knuth [5, p. 204] verified that

$$M(m, m+4)=2m+3 \quad \text{for } m \geqq 6.$$

In this paper we generalize these results by proving that

$$M(m, m+d)=2m+d-1 \quad \text{for } m \geqq 2d-2.$$

Intuitively, this means that the standard merge algorithm is optimal, in the worst-case sense, whenever $m \leqq n \leqq \lfloor 3m/2 \rfloor + 1$.

2. Oracles. A lower bound for $M(m, n)$ will be produced by means of an "oracle", the proof technique utilized, for example, by Knuth [5, § 5.3.2]. In his formulation, when presented with a comparison a_i vs. b_j, an *oracle* announces which is larger and simultaneously chooses a *strategy* for answering further questions so as to force a large number of additional comparisons to be made. A useful lower bound is obtained from an oracle that has an effective strategy for dealing with any comparison it might encounter.

In addition to an oracle that provides a lower bound for $M(m, n)$, oracles are needed to furnish lower bounds for two other functions. Let $/M(m, n)$ be the number of comparisons required to merge two lists for which, unknown to the merger, a_1 is in fact greater than b_1. An oracle for this function must therefore make all pronouncements consistent with $a_1 > b_1$. Similarly, let $/M\backslash(m, n)$ be the number of comparisons required when a_1 is greater than b_1 and a_m is less than b_n, again unknown to the merger. Occasionally we shall use the notation $M\backslash(m, n)$ to denote the number of comparisons required to merge two lists when a_m is less than b_n. This is not another new function, though, since by symmetry we have $M\backslash(m, n) = /M(m, n)$.

To illustrate these definitions, suppose $m=2$ and $n=4$. It is well known that $M(2, 4)=5$. However, there is a way to perform this merge in only 4 comparisons if in fact $a_1 > b_1$, by first comparing a_1 with b_2. If $a_1 > b_2$, the problem reduces to $M(2, 2)$; otherwise, comparing a_1 with b_1 reduces the problem to $M(1, 3)$. Thus $/M(2, 4) \leqq 4$.

3. An example. We illustrate the use of Knuth's oracles, and the strategies available to them, by verifying that $M(4, 7) \geqq 10$. Assume that oracles for achieving $M(m, n)$ and $/M(m, n)$ exist whenever $m+n \leqq 10$ (see [5]). We consider four cases.

(i) First, suppose a merge algorithm begins by comparing a_1 with b_1. The oracle declares that $a_1 > b_1$, and requires that subsequent comparisons merge $\{a_1, a_2, a_3, a_4\}$ with $\{b_2, b_3, \cdots, b_7\}$, using an $M(4, 6)$ oracle. Thus $M(4, 7) \geqq 1 + M(4, 6) = 1 + 9 = 10$ in this case.

(ii) If a merge algorithm begins by comparing a_1 with b_j, with $j \geqq 2$, a more complex strategy is needed. The oracle declares that $a_1 < b_j$, and requires that later comparisons merge $\{a_1\}$ with $\{b_1\}$ and $\{a_2, a_3, a_4\}$ with $\{b_1, b_2, \cdots, b_7\}$, with the restriction that all future pronouncements are consistent with $a_1 < b_1 < a_2$. These restrictions ensure that information gained in merging one subproblem is of no help in the other, even though b_1 is in both. The situation is illustrated in Fig. 1. The top row is A, the bottom B, with smaller elements to the left. The dotted lines represent the

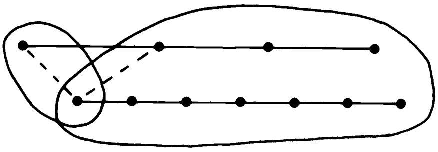


FIG. 1. $a_1 < b_j, j \geqq 2$.

restrictions the oracle imposes on itself, and the subproblems are encircled. With this strategy, the oracle can force at least $1+M\backslash(1,1)+/M(3,7)=1+1+8=10$ comparisons to be made in this case as well. Thus any algorithm which initially uses a_1 requires at least 10 comparisons.

(iii) An algorithm that first compares a_2 with b_j, with $j\leqq 3$, can be handled in a manner similar to (ii). The oracle declares that $a_2>b_j$ and requires that future comparisons merge $\{a_1\}$ with $\{b_1, b_2, b_3, b_4\}$ and $\{a_2, a_3, a_4\}$ with $\{b_4, b_5, b_6, b_7\}$, under the restrictions $a_1<b_4<a_2$. See Fig. 2. The number of comparisons required in this case is at least $1+M\backslash(1,4)+/M(3,4)=1+3+6=10$.

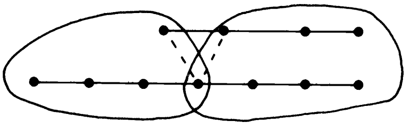


FIG. 2. $a_2>b_j, j\leqq 3$.

(iv) If the first comparison is a_2 vs. b_j with $j\geqq 4$, a simpler strategy will work. The oracle declares that $a_2<b_j$, and insists that later comparisons merge $\{a_1, a_2\}$ with $\{b_1, b_2, b_3\}$ and $\{a_3, a_4\}$ with $\{b_4, b_5, b_6, b_7\}$ as in Fig. 3. The number of comparisons required is at least $1+M(2,3)+M(3,4)=1+4+5=10$.

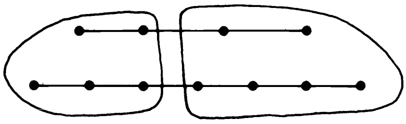


FIG. 3. $a_2<b_j, j\geqq 4$.

We have shown that any merge algorithm that begins with a comparison using either a_1 or a_2 requires at least 10 comparisons for this problem. By symmetry, the same is true for a_3 and a_4. Having considered all cases, we conclude that $M(4,7)\geqq 10$.

The two types of strategy illustrated above endow an oracle with sufficient power to prove our main result in the next section. In the "simple" strategy, the oracle answers the query and divides the merge problem into two disjoint unrestricted problems. In the "complex" strategy, there is an element of B in both subproblems, which are handled by suitably restricted oracles. Oracles for the functions $/M(m, n)$ and $/M\backslash(m, n)$ use the same strategies, with one or both subproblems inheriting the restrictions of the original. A subproblem may have one list empty in degenerate cases, as in case (i) above. In all cases, though, each subproblem contains fewer elements than the original problem, so that inductive proofs can be used.

4. The main result. The proof of our theorem is simplified by first establishing a few preliminary results.

LEMMA 1. (i) $/M\backslash(m, n)\leqq /M(m, n)\leqq M(m, n)$.

(ii) $/M(m+1, n+1)\geqq /M(m, n)+2$.

(iii) $/M\backslash(m+1, n+1)\geqq /M\backslash(m, n)+2$.

Proof. Part (i) is obvious; any merge algorithm performs at least as well on more restricted problems. In part (ii), an oracle for $/M(m+1, n+1)$ can make all pronouncements consistent with $b_1<a_1<b_2<a_2$, and force $\{a_2, a_3, \cdots, a_{m+1}\}$ to be merged with $\{b_2, b_3, \cdots, b_{n+1}\}$. Then the comparisons a_1 vs. b_1 and a_1 vs. b_2 can not be avoided. The proof of part (iii) is similar.

We are now ready to prove the main result. Although we are really interested only in part (a), bounds for all three functions must be proved simultaneously, as each oracle requires the help of at least one other.

THEOREM 1. (a) $M(m, m+d) \geqq 2m+d-1$ *for* $m \geqq 2d-2$;

(b) $/M(m, m+d) \geqq 2m+d-1$ *for* $m \geqq 2d-1$;

(c) $/M\backslash(m, m+d+2) \geqq 2m+d$ *for* $m \geqq 2d-1$.

Proof. If (b) and (c) are true for the threshold values $m = 2d-1$, then they are also true for $m > 2d-1$ by repeated application of Lemma 1 (ii) and (iii). Also, if (b) is true for $m \geqq 2d-1$ then Lemma 1 (i) implies that (a) is also true for $m \geqq 2d-1$. Thus it is sufficient to prove the theorem for the threshold values of m only, that is,

$$M(2d-2, 3d-2) \geqq 5d-5,$$

$$/M(2d-1, 3d-1) \geqq 5d-3$$

and

$$/M\backslash(2d-1, 3d+1) \geqq 5d-2.$$

The proof is by induction on d. The starting values for $1 \leqq d \leqq 3$ are given in Knuth [5, p. 203].

Part (a). Suppose an algorithm begins by comparing a_i with b_j, where $i = 2k-1$ and $j \leqq 3k-2$, for some integer k satisfying $1 \leqq k < d$. The oracle proclaims that $a_i > b_j$ and follows the simple strategy, yielding

$$\begin{aligned} M(2d-2, 3d-2) &\geqq 1 + M(2k-2, 3k-2) + M(2(d-k), 3(d-k)) \\ &\geqq 1 + (5k-5) + (5(d-k)-1) \\ &= 5d-5. \end{aligned}$$

If $i = 2k-1$ and $j \geqq 3k-1$, the oracle announces that $a_i < b_j$ and uses the complex strategy, with b_{3k-2} in both subproblems. This leads to

$$\begin{aligned} M(2d-2, 3d-2) &\geqq 1 + M\backslash(2k-1, 3k-2) + /M(2(d-k)-1, 3(d-k)+1) \\ &\geqq 1 + /M(2k-1, 3k-2) + /M\backslash(2(d-k)-1, 3(d-k)+1) \\ &\geqq 1 + (5k-4) + (5(d-k)-2) \\ &= 5d-5. \end{aligned}$$

This settles the case where i is odd. Reversing the order of the elements in A and B maps all points of A with even subscripts onto those with odd. Thus by symmetry we have handled the even case as well.

Part (b). Suppose the first comparison of an algorithm is a_i vs. b_j with $i = 2k-1$ and $j \leqq 3k-2$, where $1 \leqq k \leqq d$. The oracle proclaims that $a_i > b_j$ and uses the complex strategy, with b_{3k-1} in both subproblems. In this case we have

$$\begin{aligned} /M(2d-1, 3d-1) &\geqq 1 + /M\backslash(2k-2, 3k-1) + /M(2(d-k)+1, 3(d-k)+1) \\ &\geqq 1 + (5k-5) + (5(d-k)+1) \\ &= 5d-3. \end{aligned}$$

If $i = 2k-1$ and $j \geqq 3k-1$, the oracle announces that $a_i < b_j$. The simple strategy yields

$$\begin{aligned} /M(2d-1, 3d-1) &\geqq 1 + /M(2k-1, 3k-2) + M(2(d-k), 3(d-k)+1) \\ &\geqq 1 + (5k-4) + 5(d-k) \\ &= 5d-3. \end{aligned}$$

Now suppose $i = 2k$ and $j \leqq 3k$, with $1 \leqq k < d$. Choosing $a_i > b_j$, the oracle follows the complex strategy, leading to

$$\begin{aligned} /M(2d-1, 3d-1) &\geqq 1 + /M\backslash(2k-1, 3k+1) + /M(2(d-k), 3(d-k)-1) \\ &\geqq 1 + (5k-2) + (5(d-k)-2) \\ &= 5d-3. \end{aligned}$$

Otherwise, if $i = 2k$ and $j \geqq 3k+1$, the simple strategy with $a_i < b_j$ produces

$$\begin{aligned} /M(2d-1, 3d-1) &\geqq 1 + /M(2k, 3k) + M(2(d-k)-1, 3(d-k)-1) \\ &\geqq 1 + (5k-1) + (5(d-k)-3) \\ &= 5d-3. \end{aligned}$$

Part (c). Assume an algorithm begins a_i vs. b_j with $i = 2k-1$ and $j \leqq 3k-1$, where $1 \leqq k \leqq d$. The oracle picks $a_i > b_j$ and follows the simple strategy, yielding

$$\begin{aligned} /M\backslash(2d-1, 3d+1) &\geqq 1 + /M(2k-2, 3k-1) + M\backslash(2(d-k)+1, 3(d-k)+2) \\ &\geqq 1 + /M\backslash(2k-2, 3k-1) + /M(2(d-k)+1, 3(d-k)+2) \\ &\geqq 1 + (5k-5) + (5(d-k)+2) \\ &= 5d-2. \end{aligned}$$

The case $i = 2k-1$ and $j \geqq 3k$ is the mirror image of this case.

If $i = 2k$ and $j \leqq 3k+1$, with $1 \leqq k < d$, the simple strategy works again. The oracle declares $a_i > b_j$, and we have

$$\begin{aligned} /M\backslash(2d-1, 3d+1) &\geqq 1 + /M(2k-1, 3k+1) + M\backslash(2(d-k), 3(d-k)) \\ &\geqq 1 + /M\backslash(2k-1, 3k+1) + /M(2(d-k), 3(d-k)) \\ &\geqq 1 + (5k-2) + (5(d-k)-1) \\ &= 5d-2. \end{aligned}$$

Finally, the case $i = 2k$ and $j \geqq 3k+1$ is contained in the mirror image of this case.

In conclusion, we note that Knuth [5, p. 206] has made several conjectures concerning the behavior of $M(m, n)$, such as

$$M(m+1, n+1) \geqq M(m, n) + 2.$$

In view of Theorem 1, it seems reasonable to add

$$M(m+2, n+3) \geqq M(m, n) + 5$$

to the list.

Also, it would be interesting to know the precise range of m and n for which the linear merge algorithm is optimal. No instances have been found outside the range $m \leqq n \leqq \lfloor 3m/2 \rfloor + 1$, but cases as small as $m = 7$, $n = 12$ remain open.

Note added in proof. It has come to the authors' attention that essentially the same results have been proved independently by C. Christen in *On the optimality of the straight merging algorithm*, Publication number 296, Département d'informatique et recherce opérationelle, Université de Montréal.

REFERENCES

[1] R. L. GRAHAM, *On sorting by comparisons*, Computers in Number Theory, A. O. L. Atkin and B. J. Birch, eds., Academic Press, London, 1971, pp. 263–269.

[2] F. K. HWANG, *Optimal merging of* 3 *elements with n elements*, to appear.

[3] F. K. HWANG AND S. LIN, *Some optimality results in merging two disjoint linearly-ordered sets*, Bell Telephone Laboratories internal memorandum, 1970.

[4] ———, *Optimal merging of* 2 *elements with n elements*, Acta Informat., 1 (1971), pp. 145–158.

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

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/09001-0007 $01.00/0

BOUNDS FOR LIST SCHEDULES ON UNIFORM PROCESSORS*

YOOKUN CHO† AND SARTAJ SAHNI‡

Abstract. Bounds are derived for the worst case performance of list schedules relative to minimum finish time schedules for uniform processor systems. The tasks to be scheduled are assumed to be independent and only nonpreemptive schedules are considered.

Key words. list schedules, nonpreemptive schedules, uniform processors, independent tasks

1. Introduction. A uniform processor system consists of m, $m \geqq 1$, processors $P_1, P_2, \cdots, P_m$. Associated with each processor is a speed s_i, $s_i \geqq 1$. In one unit of time P_i can carry out s_i units of processing. Without loss of generality, we may assume $s_i \leqq s_{i+1}$, $1 \leqq i < m$ and $s_1 = 1$. We are given n independent tasks that are to be processed. Task i requires t_i units of processing (t_i is the *task time* of task i). If task i is assigned to P_j then t_i/s_j time units are needed to finish this task. A *nonpreemptive schedule* is an assignment of tasks to processors such that each task is assigned to exactly one processor. For each processor the order in which tasks are to be processed is also specified. If T_i is the set of tasks assigned to P_i then the *finish time* of P_i is $(\sum_{j \in T_i} t_j)/s_i$. The finish time of the schedule is the time at which all processors $\sum_{j \in T_i} t_j$ have finished processing.

For the case when $s_i = 1$, $1 \leqq i \leqq m$ the processor system defined above is known as a system of *identical processors.* It is well known that finding minimum finish time nonpreemptive schedules for identical processors with $m \geqq 2$ is *NP*-hard (see e.g. [7]). Several heuristics to obtain "near optimal" schedules for identical processors have been studied. Graham [4] has studied the performance of LPT schedules. In an LPT schedule tasks are assigned to processors in nonincreasing order of task times. Whenever a task is to be assigned, it is assigned to that processor on which it will finish earliest. Ties are broken arbitrarily and tasks are processed in the order assigned. Let $\hat{f}$ be the finish time of an LPT schedule for any given task set. Let f^* be the finish time of an optimal schedule. Graham [4] has shown that

$$\hat{f}/f^* \leqq 4/3 - 1/(3m).$$

Another heuristic studied by Graham is the list schedule. This scheduling rule differs from the LPT rule only in the order in which tasks are considered for assignment to processors. A list (or permutation) of the indices $1, 2, \cdots, n$ is provided. Tasks are considered in the order in which they appear on this list. If $\hat{f}$ is the finish time of a list schedule and f^* that of an optimal schedule for any given task set then it is known [3] that:

$$\hat{f}/f^* \leqq 2 - \frac{1}{m}.$$

Hence, for identical processor systems LPT schedules are better than arbitrary list schedules by only a constant factor. Note that an LPT schedule is a special case of a list schedule (i.e., the case when the tasks in the list are ordered in nonincreasing order of task times). Other heuristics for identical processor systems have been studied by Coffman, Garey and Johnson [1] and Sahni [9].

* Received by the editors July 25, 1978. This research was supported in part by the National Science Foundation under Grant MCS 76-21024.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455. Presently at Seoul National University, Korea.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

Another special case of a uniform processor system is when $s_i = 1, 1 \leqq i < m$ and $s_m > 1$. This case has been studied by Liu and Liu [8] and Gonzalez, Ibarra and Sahni [2]. Liu and Liu [8] considered a variation of the LPT rule defined here. They require a task to be assigned to that processor that becomes idle first rather than to the processor on which it will complete first. For this rule they show that

$$\hat{f}/f^* \leqq \begin{cases} 2(m-1+s_m)/(s_m+2) & \text{for } s_m \leqq 2, \\ (m-1+s_m)/2 & \text{for } s_m > 2. \end{cases}$$

Gonzalez, Ibarra and Sahni [2] show that for LPT schedules

$$\hat{f}/f^* \leqq \begin{cases} \dfrac{1+\sqrt{17}}{4}, & m=2, \\ 3/2-1/(2m), & m>2. \end{cases}$$

Finally, LPT schedules for general uniform processor systems have been analyzed by Gonzalez, Ibarra and Sahni [2]. They show that

$$\hat{f}/f^* \leqq 2m/(m+1).$$

Liu and Liu [8] have analyzed list schedules for the case when $s_i = 1, 1 \leqq i < m$ and $s_m < 1$. Their analysis assumes that the next task on the list is to be assigned to the first idle processor. Under this assumption they obtain the bound:

$$\hat{f}/f^* \leqq s_m + \frac{m-1}{s_m+m-1}.$$

A survey of similar results for other machine and task set models appears in [5]. In this paper we analyze arbitrary list schedules for the case of general uniform processor systems and also for the special case when $s_i = 1$, $1 \leqq i < m$ and $s_m > 1$.

In § 2 we show that for the general case

$$\hat{f}/f^* \leqq \begin{cases} (1+\sqrt{5})/2, & m=2, \\ 1+(\sqrt{2m-2})/2, & m>2. \end{cases}$$

For the case of $m \leqq 6$ the bounds given above are tight in the sense that there exist task sets and lists for which $\hat{f}/f^*$ equals the stated bound. For $m > 6$ we are unable to show the bound tight and suspect that it is not tight. We also present an example that shows that $\hat{f}/f^*$ is not bounded by any constant. Hence, while for identical processors LPT schedules are better than list schedules by only a constant factor, for general uniform processor systems the ratio of the finish time of an LPT schedule to that of an arbitrary list schedule is not bounded by any constant (but by some function of m).

In § 3 we consider the special case of $s_i = 1$, $1 \leqq 1 < m$ and $s_m > 1$. For this case we show that

$$\hat{f}/f^* \leqq \begin{cases} (1+\sqrt{5})/2, & m=2, \\ 3-4/(m+1), & m \geqq 3. \end{cases}$$

Furthermore, the bound is tight.

2. General uniform processor systems. In this section the following theorem is established:

THEOREM 1. *Let t_i, $1 \leqq i \leqq n$ be the task times of n independent tasks. Let s_i, $1 \leqq i \leqq m$ be the speeds of the m processors in the system. Let $\hat{f}$ be the finish time of the schedule obtained using any given list L and let f^* be the finish time of an optimal schedule for the* *task set. Then,*

$$\hat{f}/f^* \begin{cases} (1+\sqrt{5})/2, & m=2, \\ 1+(\sqrt{2m-2})/2, & m \geqq 3. \end{cases}$$

Furthermore, for $m \leqq 6$ there exists task sets, lists and processor systems for which $\hat{f}/f^*$ equals the above bound.

Theorem 1 will be proved in several steps. First, we derive some relationships between $\hat{f}/f^*$ and the s_i's and t_i's. In the following it will be assumed that $s_1 \leqq s_2 \leqq \cdots \leqq s_m$, $s_1 = 1$ and $f^* = 1$. We shall also assume that the tasks have been indexed so that the list is given by $L = (1, 2, 3, \cdots, n)$. Note that these assumptions do not affect the generality of the proofs. Any problem instance that violates one or more of these assumptions may be transformed into an equivalent problem instance satisfying all the assumptions by sorting the s_i's, reordering the tasks and appropriately scaling the s_i's and t_i's. We shall also assume that in case of a tie, the list scheduling algorithm assigns the task to the processor with highest index. Since our proof will be a proof by contradiction, it is necessary to develop relationships only for the smallest n (for any given m) for which the theorem may be violated. Thus, if $t_1, \cdots, t_n$ defines a task set with least n for which the theorem does not hold then it is easy to see that the finish time of the list schedule is determined by task n. To see this, observe that if $i < n$ determines the finish time then we can eliminate tasks $i+1, \cdots, n$ and consider only tasks $1, 2, \cdots, i$. For this set, $\hat{f}$ is unchanged and f^* is not increased. So, $\hat{f}/f^*$ does not decrease and we have a smaller instance violating the bounds of the theorem. Now, since task n finishes at $\hat{f}$, we can imagine the list schedule to look like Fig. 1. Here j is any index in the range $[1, m]$ and $\hat{f} = F_j + t_n/s_j$. Let F_i be the finish time of P_i, $1 \leqq i \leqq m$, before task n is scheduled.

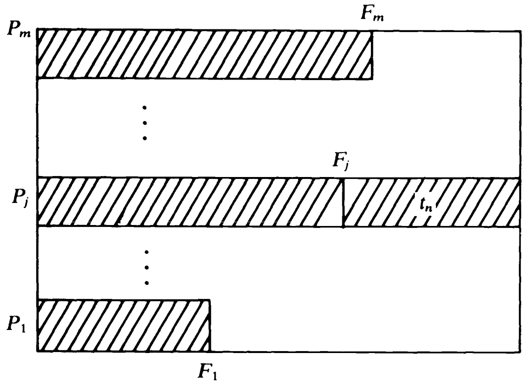


FIGURE 1

Since we are assuming $f^* = 1$, it follows that

$$\sum_1^n t_i \leqq \sum_1^m s_i, \tag{1}$$

$$t_n \leqq \max\{t_i\} \leqq s_m. \tag{2}$$

From the definition of list schedules, it follows that

$$F_i + t_n/s_i \geqq \hat{f}, \qquad 1 \leqq i \leqq m, \tag{3}$$

or $s_iF_i+t_n \geqq s_i\hat{f}$, $1 \leqq i \leqq m$. Using (3) with $i=m$ we obtain

$$s_m\hat{f} \leqq s_mF_m+t_n \leqq \sum_1^n t_i \leqq \sum_1^m s_i. \tag{4}$$

Hence, $\hat{f} \leqq \sum_i^m s_i/s_m$.

From (3), we obtain

$$s_i(\hat{f}-F_i) \leqq t_n, \qquad 1 \leqq i \leqq m, \tag{5}$$

Summing (5) for $1 \leqq i \leqq m$ and using (1), we get

$$\sum_1^m s_i\hat{f} - \sum_1^m s_iF_i \leqq (m-1)t_n + t_n$$

or

$$\begin{aligned}\sum_1^m s_i\hat{f} &\leqq (m-1)t_n + \sum_1^m s_iF_i + t_n \\ &\leqq (m-1)t_n + \sum_1^n t_i \\ &\leqq (m-1)t_n + \sum_1^m s_i.\end{aligned}$$

Hence,

$$\hat{f} \leqq 1 + (m-1)t_n \Big/ \sum_1^m s_i \tag{6a}$$

$$\leqq 1 + (m-1)s_m \Big/ \sum_1^m s_i. \tag{6b}$$

LEMMA 1. $\hat{f}/f^* \leqq (1+\sqrt{4m-3})/2$, $m \geqq 2$.

Proof. Assume the lemma is false. Consider the smallest n for which it is false. From the preceding discussion we may assume $f^*=1$ and that the list is $(1, 2, \cdots, n)$. Hence equations (1)–(6) hold.

Using x to represent the ratio $\sum_1^m s_i/s_m$ we obtain (7) from (4) and (6b).

$$\hat{f} \leqq \min\{x, 1+(m-1)/x\}. \tag{7}$$

The maximum value of the right hand side of (7) is obtained when

$$x = 1+(m-1)/x \quad \text{or} \quad x^2-x-(m-1)=0 \quad \text{or} \quad x=(1+\sqrt{4m-3})/2.$$

Hence, $\hat{f} \leqq (1+\sqrt{4m-3})/2$. So, the lemma must be true for all m, all task sets and all lists. □

When $m=2$, $(1+\sqrt{4m-3})/2=(1+\sqrt{5})/2$. This observation together with the following example proves Theorem 1 when $m=2$.

Example 1. Let $s_1=1$, $s_2=(1+\sqrt{5})/2$, $t_1=1$ and $t_2=(1+\sqrt{5})/2$. (Note that by assumption $L=(1, 2)$.) It is clear that $f^*=1$. In the list schedule however, both tasks 1 and 2 get assigned to P_2 and

$$\hat{f}=(t_1+t_2)/s_2=(1+\sqrt{5})/2.$$

When $m=3$, $(1+\sqrt{4m-3})/2=1+(\sqrt{2m}-2)/2=2$. This together with Example 2 establishes Theorem 1 for $m=3$.

Example 2. Consider, $s_1 = s_2 = 1$, $s_3 = 2$, $t_1 = t_2 = 1$ and $t_3 = 2$. Again, $f^* = 1$. In the list schedule all three tasks get assigned to P_3. Hence, $\hat{f} = 2$. Note that if a different tie breaking rule is used then we may replace s_3 and t_3 by $2+\varepsilon$. In this case $\hat{f} = (4+\varepsilon)/(2+\varepsilon)$ whuch approaches 2 as $\varepsilon \to 0$.

The bound of Lemma 1 is not tight for $m > 3$. This is established by obtaining a smaller bound. First, we derive some more inequalities. We readily observe that in every list schedule, task 1 is always assigned to P_m. Since we may assume $n > 1$, it follows that P_m always has at least one task (other than task n) assigned to it. Consider the status of the list schedule just before task n is assigned to a processor. Let t' be the task time of the last task assigned to P_m (note that this task cannot be task n as it has not yet been assigned). Let G_i be the finish time of P_i, $1 \leqq i \leqq m$ just before t' was assigned to P_m. It follows that $G_i \leqq F_i$, $1 \leqq i \leqq m$. Since t' was assigned to P_m, if follows that:

$$(8)\qquad F_i + t'/s_i \geqq G_i + t'/s_i \geqq G_m + t'/s_m = F_m, \qquad 1 \leqq i \leqq m.$$

Using t to denote t_n and substituting $i = m$ in (3) we obtain

$$s_m F_m + t \geqq s_m \hat{f} \quad \text{or} \quad F_m \geqq \hat{f} - t/s_m.$$

Substituting into (8) we get

$$(9)\qquad F_i + t'/s_i \geqq \hat{f} - t/s_m \quad \text{or} \quad s_i F_i + t' \geqq s_i \hat{f} - t s_i / s_m, \qquad 1 \leqq i \leqq m.$$

Further, it follows from $f^* = 1$ that

$$(10)\qquad t' + t \leqq s_{m-1} + s_m.$$

LEMMA 2. $\hat{f}/f^* \leqq 1 + (\sqrt{2m-2})/2$, $m \geqq 4$.

Proof. Suppose the lemma is not true. Consider the least n for which

$$(11)\qquad \hat{f}/f^* > 1 + (\sqrt{2m-2})/2.$$

We may assume $f^* = 1$ and that the list is $(1, 2, \cdots, n)$. Let t' and t be as defined before. We first recall the following inequality:

$$(12)\qquad \sum_1^m s_i F_i + t = \sum_1^n t_i \leqq \sum_1^m s_i.$$

From (3) with $i = m$ and $m-1$ we get

$$s_{m-1} F_{m-1} + s_m F_m + 2t \geqq \hat{f}(s_{m-1} + s_m).$$

So,

$$\sum_1^m s_i F_i + 2t \geqq \hat{f}(s_{m-1} + s_m).$$

This together with (2) and (12) yields

$$(13)\qquad \sum_1^m s_i + s_m \geqq \hat{f}(s_{m-1} + s_m) \quad \text{or} \quad \hat{f} \leqq 1 + \frac{\sum_1^m s_i - s_{m-1}}{s_{m-1} + s_m}.$$

Equations (11) and (13) together yield:

$$1 + \frac{\sqrt{2m-2}}{2} < 1 + \frac{\sum_1^m s_i - s_{m-1}}{s_{m-1} + s_m}$$

or

$$(s_{m-1}+s_m)<\frac{2}{\sqrt{2m-2}}\left(\sum_1^m s_i-s_{m-1}\right). \tag{14}$$

Summing up $m-1$ inequalities from (3) (i.e. $1<i\leqq m$) and inequality (9) with $i=1$, we get

$$\sum_1^m s_iF_i+(m-1)t+t'\geqq\sum_1^m s_i\hat{f}-s_1t/s_m.$$

Substituting (12) into the above equation, we get

$$\sum_1^m s_i+(m-2)t+t'\geqq\sum_1^m s_i\hat{f}-s_1t/s_m.$$

From this and (2) we get

$$\sum_1^m s_i+(m-2)t+t'+s_1\geqq\sum_1^m s_i\hat{f}. \tag{15}$$

Also, from (6b) and (11) we get

$$\frac{\sqrt{2m-2}}{2}<(m-1)s_m\Big/\sum_1^m s_i. \tag{16}$$

The next step is to show that if (11) holds then $t>s_{m-1}$. Suppose $t\leqq s_{m-1}$ then from (15) and the knowledge $t'\leqq\max\{t_i\}\leqq s_m$ we get

$$\sum_1^m s_i+(m-2)s_{m-1}+s_m+s_1\geqq\sum_1^m s_i\hat{f}.$$

Rearranging terms, we get

$$\hat{f}\leqq 2+(m-3)s_{m-1}\Big/\sum_1^m s_i. \tag{17}$$

This together with (11) gives

$$1+(\sqrt{2m-2})/2<2+(m-3)s_{m-1}\Big/\sum_1^m s_i$$

or

$$\frac{\sqrt{2m-2}-2}{2(m-3)}<\frac{s_{m-1}}{\sum_1^m s_i}. \tag{18}$$

Adding $1/(m-1)$ times (16) to (18) we get

$$\frac{\sqrt{2m-2}-2}{2(m-3)}+\frac{\sqrt{2m-2}}{2(m-1)}<\frac{s_{m-1}+s_m}{\sum_1^m s_i}. \tag{19}$$

Combining (19) and (14) we get

$$\frac{\sqrt{2m-2}-2}{2(m-3)}+\frac{\sqrt{2m-2}}{2(m-1)}<\frac{2}{\sqrt{2m-2}}\left(1-\frac{s_{m-1}}{\sum s_i}\right).$$

Simplifying, we get

$$\frac{s_{m-1}}{\sum_1^m s_i} < \frac{\sqrt{2m-2}-2}{2(m-3)}.$$

This contradicts (18). So, $t > s_{m-1}$. Hence, task n is scheduled on P_m in f^*. If t' is also on P_m then $t' \leqq s_m - t$. Otherwise, $t' \leqq s_{m-1}$. Hence,

$$t' \leqq \max\{s_{m-1}, s_m - t\}. \tag{20}$$

We shall now show that no matter which of s_{m-1} and $s_m - t$ is maximum we arrive at contradicting relations. So, $\hat{f} \leqq 1 + (\sqrt{2m} - 2)/2$.

Summing (3) with $i = m$ and $m-1$ and (9) with $1 \leqq i \leqq m-2$, we obtain

$$\sum_1^m s_i F_i + 2t + (m-2)t' \geqq \hat{f} \sum_1^m s_i - \sum_1^{m-2} s_i t/s_m.$$

Substituting from (2), (10), and (12) we reduce this to

$$\sum_1^m s_i + s_{m-1} + s_m + (m-3)t' + \sum_1^{m-2} s_i \geqq \hat{f} \sum_1^m s_i$$

or

$$\hat{f} \leqq 2 + (m-3)t' \Big/ \sum_1^m s_i. \tag{21}$$

First, let us consider the case $s_{m-1} \geqq s_m - t$. (20) yields $t' \leqq s_{m-1}$. Substituting into (21) we get

$$\hat{f} \leqq 2 + (m-3)s_{m-1} \Big/ \sum_1^m s_i.$$

This is the same as (17) and together with (11) and (16) can be used to derive (19) and arrive at a contradiction as before.

If $s_{m-1} < s_m - t$ then $t' \leqq s_m - t$ and $t < s_m - s_{m-1}$. Substituting into (21) we get

$$\hat{f} \leqq 2 + (m-3)(s_m - t) \Big/ \sum_1^m s_i. \tag{22}$$

Adding (22) and $(m-3)/(m-1)$ times (6a) we get

$$\frac{2m-4}{m-1}\hat{f} \leqq 2 + \frac{m-3}{m-1} + \frac{(m-3)s_m}{\sum_1^m s_i}.$$

Substituting for $\hat{f}$ from (11) we get

$$\frac{2m-4}{m-1}\left(1 + \frac{\sqrt{2m-2}}{2}\right) < 2 + \frac{m-3}{m-1} + \frac{(m-3)s_m}{\sum_1^m s_i}$$

or

$$\frac{2m-4}{m-1} + \frac{(2m-4)\sqrt{2m-2}}{2(m-1)} - 2 - \frac{m-3}{m-1} < \frac{(m-3)s_m}{\sum_1^m s_i}$$

or

$$\frac{(m-2)\sqrt{2m-2}}{m-1} - 1 < \frac{(m-3)s_m}{\sum_1^m s_i}$$

or

$$\frac{(m-2)\sqrt{2m-2}-(m-1)}{(m-1)(m-3)} < \frac{s_m}{\sum_1^m s_i}.$$

Combining with (4) we get

$$\hat{f} < \frac{(m-1)(m-3)}{(m-2)\sqrt{2m-2}-(m-1)}. \tag{23}$$

One may easily verify that the right hand side of (23) is no more than $1+(\sqrt{2m}-2)/2$ for $m \geqq 4$. This contradicts (11) and establishes the lemma. □

To complete the proof of Theorem 1 we need to show that the bound is tight for $m=4$, 5 and 6. The next three examples do this.

Example 3. $m=4$, $s_1=s_2=1$, $s_3=\sqrt{6}/2$, $s_4=s_3+1$. $t_1=t_2=.5$, $t_3=1$, $t_4=\sqrt{6}/2$ and $t_5=1+\sqrt{6}/2$. Clearly, $f^*=1$.
Figure 2(a) shows the list schedule. $\hat{f}=1+\sqrt{6}/2=1+(\sqrt{2m}-2)/2$.

Example 4. $m=5$, $s_1=s_2=s_3=1$, $s_4=\sqrt{2}$, $s_5=1+\sqrt{2}$. $t_1=t_2=t_3=1$, $t_4=\sqrt{2}$, $t_5=1+\sqrt{2}$. $f^*=1$. The list schedule is shown in Figure 2(b) and $\hat{f}=1+\sqrt{2}=1+(\sqrt{2m}-2)/2$.

Example 5. $m=6$, $s_1=s_2=s_3=s_4=1$, $s_5=\sqrt{10}/2$, $s_6=1+\sqrt{10}/2$. $t_1=t_2=.5$, $t_3=t_4=t_5=1$, $t_6=\sqrt{10}/2$ and $t_7=1+(\sqrt{10})/2$. Again, $f^*=1$, $\hat{f}=1+\sqrt{10}/2=1+(\sqrt{2m}-2)/2$ (see Fig. 2(c)).

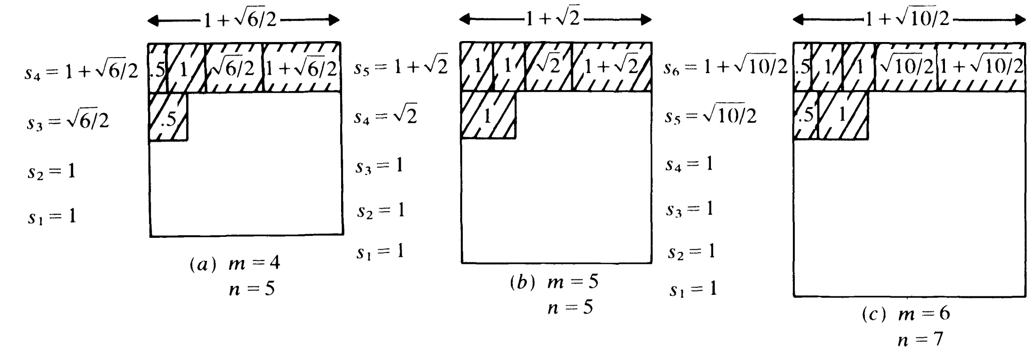
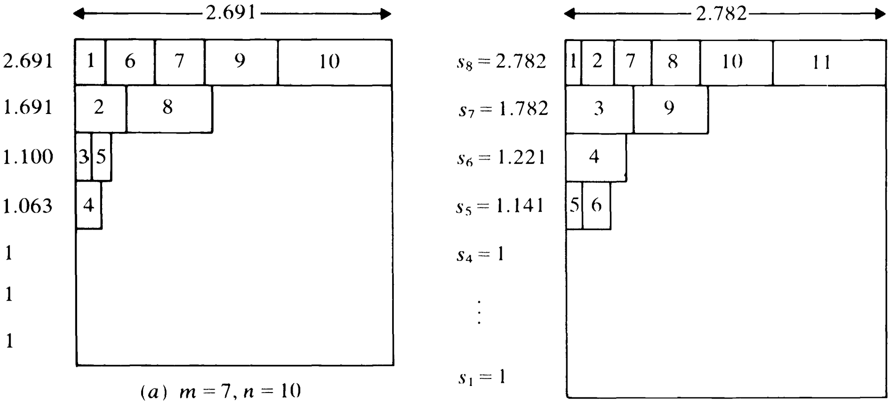


FIG. 2. *List schedules for Examples* 3, 4 *and* 5.

The question that naturally arises at this time is: What happens when $m>6$? Is it possible for $\hat{f}/f^*$ to get as large as the bound given in Theorem 1? We have been unable to generate examples achieving this bound for $m>6$. The worst examples we were able to generate for $m=7$ and 8 are given in Examples 6 and 7. These were arrived at by considering a certain job distribution pattern, deriving inequalities that led to a cubic equation. A root of this equation yielded s_m and the remaining numbers were obtained by back substitution.

Example 6. $m=7$. Let r be a real root of the cubic equation $4r^3-11r^2+r-1=0$. $s_1=s_2=s_3=1$, $s_4=r+1/r-2$, $s_5=r-r/(r-1)$, $s_6=r-1$, $s_7=r$. $t_1=r^2-3r+2-1/r$, $t_2=r^2-3r+r/(r-1)$, $t_3=1-t_1-t_4$, $t_4=r^2-3r+1$, $t_5=1-t_2$, $t_6=1$, $t_7=r+1/r-2$, $t_8=r-r/(r-1)$, $t_9=r-1$, $t_{10}=r$. For an optimal schedule, the list is (10, 9, 8, 7, 6, 5, 4, 3, 1, 2) Clearly $f^*=1$. An approximate value for r is 2.691. This yields $s_4=1.063$, $s_5=1.100$, $s_6=1.691$, $s_7=2.691$, $t_1=.797$, $t_2=.760$, $t_3=.033$, $t_4=.170$, $t_5=.240$, $t_6=1$, $t_7=1.063$, $t_8=1.100$, $t_9=1.691$ and $t_{10}=2.691$. For the list schedule consider the list $(1, 2, \cdots, 10)$. The resulting schedule is shown in Fig. 3(a) and $\hat{f} = 2.69 \ldots$. The bound of Theorem 1 when $m = 7$ is 2.732.

Example 7. $m = 8$. Let r be a real root of the cubic equation $4r^3 - 11r^2 - 1 = 0$. $s_1 = s_2 = s_3 = s_4 = 1$, $s_5 = r + 1/r - 2$, $s_6 = r - r/(r-1)$, $s_7 = r - 1$, $s_8 = r$. $t_1 = r^2 - 3r + 1 - 1/r$, $t_2 = 1$, $t_3 = r^2 - 3r + r/(r-1)$, $t_4 = r^2 - 2r - r/(r-1)$, $t_5 = 1 - t_3$, $t_6 = 1 - t_4 - t_1$, $t_7 = 1$ $t_8 = r + 1/r - 2$, $t_9 = r - r/(r-1)$, $t_{10} = r - 1$, $t_{11} = r$. For an optimal schedule, the list is $(11, 10, 9, 8, 7, 6, 5, 1, 4, 3, 2)$. Again $f^* = 1$. An approximate value for r is 2.782. Using this, we get $s_5 = 1.141$, $s_6 = 1.221$, $s_7 = 1.782$, $s_8 = 2.782$, $t_1 = .035$, $t_2 = 1$, $t_3 = .955$, $t_4 = .615$, $t_5 = .045$, $t_6 = .350$, $t_7 = 1$, $t_8 = 1.141$, $t_9 = 1.221$, $t_{10} = 1.782$, $t_{11} = 2.782$. Assume a list schedule is constructed using the list $(1, 2, \cdots, 11)$. The resulting schedule is shown in Figure 3(b) and $\hat{f} = 2.782 \ldots$. The bound of Theorem 1 is 2.87.



(*a*) $m = 7, n = 10$

(*b*) $m = 8, n = 11$

FIG. 3. *List schedules for Examples* 6 *and* 7.

While we have been unable to establish a tight bound for $m > 6$, we can show that the bound on $\hat{f}/f^*$ must increase as m increases. Hence, there is no constant k such that $\hat{f}/f^* \leqq k$ for all m. This result should be contrasted with the bound for identical processors which is itself bounded by 2.

THEOREM 2. *There exist task sets, uniform processor systems, and lists for which* $\hat{f}/f^* \geqq \lfloor (\log_2 (3m - 1) + 1)/2 \rfloor$.

Proof. Let $k = (\log_2 (3m - 1) + 1)/2$. We shall construct an example that achieves the above bound when m is such that k has integer values (i.e. when $m = 3, 11, 43, \cdots$). There are k sets of processors G_i, $1 \leqq i \leqq k$. Each processor in G_i has a speed of 2^i. $|G_i| = 2^{2k-2i-1}$, $1 \leqq i < k$ and $|G_k| = 1$. Thus, the total number of processors is $m = 1 + 2^1 + 2^3 \cdots + 2^{2k-3} = 2(4^{k-1} - 1)/3 + 1$.

We shall have k sets of tasks T_i, $1 \leqq i \leqq k$. The task time of a task in T_i, is 2^i, $1 \leqq i \leqq k$. Also, $|T_i| = 2^{2k-2i-1}$, $1 \leqq i < k$ and $|T_k| = 1$.

It is easily verified that $f^* = 1$. Consider the list in which all tasks appear as follows: T_1 tasks followed by T_2 tasks followed by T_3 tasks etc. The resulting list schedule is given in Fig. 4. The schedule consists of k columns. Each column contains tasks with identical task times. In column i, $1 \leqq i < k$ the processors in G_j, $i + 1 \leqq j \leqq k$ will be processing tasks with a task time of 2^i. The number of tasks on a processor in G_j, $i + 1 \leqq j \leqq k$ is 2^{j-i}. Thus, the total number of tasks scheduled on all processors in G_j will be $2^{j-i} * |G_j| = 2^{j-i} * 2^{2k-2j-1} = 2^{2k-j-i-1}$ for $i + 1 \leqq j < k$ and 2^{k-i} for G_k. Therefore the total number of tasks in column i is

$$\begin{aligned}\sum_{j=i+1}^{k-1} 2^{2k-j-i-1}+2^{k-i} &= \sum_{j=1}^{k-i-1} 2^{2k-2i-1-j}+2^{k-i}\\ &=2^{2k-2i-1}\sum_{j=1}^{k-i-1} 1/2^j+2^{k-i}\\ &=2^{2k-2i-1}(1/2((1/2)^{k-i-1}-1)/(1/2-1))+2^{k-i}\\ &=2^{2k-2i-1}(1-(1/2)^{k-i-1})+2^{k-1}\\ &=2^{2k-2i-1}-2^{k-i}+2^{k-i}=2^{2k-2i-1}.\end{aligned}$$

In column i, we need $2^{2k-2i-1}$ tasks with task time 2^i. We observe that the number of tasks with task time 2^i is exactly $2^{2k-2i-1}$ for $1\leqq i<k$. Thus, all tasks with the same task time are scheduled in one column, i.e., all tasks from T_i, $1\leqq i\leqq k$ will be processed in column i. Note that exactly k columns can be scheduled with the given tasks. It is clear that any task in column i, $2\leqq i\leqq k$ cannot be processed in column j, $1\leqq j\leqq i$. Hence, $\hat{f}=k$. □

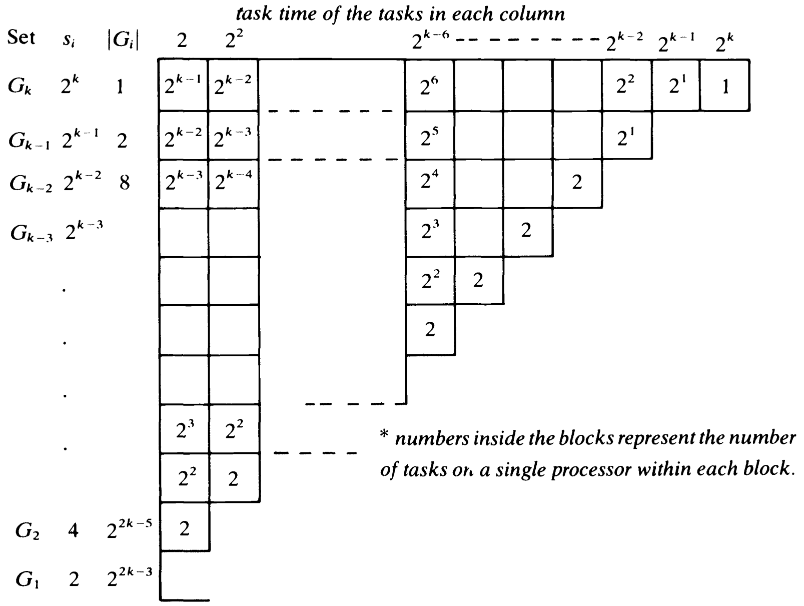


FIG. 4. *List schedule for Theorem* 2.

3. The case $s_i=1$, $1\leqq i<m$ and $s_m>1$. In this section the following theorem is established:

THEOREM 3. *If* $s_i=1$, $1\leqq i<m$ *and* $s_m>1$ *then*

$$\hat{f}/f^*\leqq\begin{cases}(1+\sqrt{5})/2, & m=2,\\ 3-4/(m+1), & m\geqq 3.\end{cases}$$

For each m, $m\geqq 2$ there exists task sets, lists and s_m for which $\hat{f}/f^*$ equals the above bound.

For $m=2$ and 3 the bounds for Theorems 1 and 3 are the same. Moreover, the examples given in the last section for $m=2$ and 3 can be used here too. So, the theorem needs to be proved only for $m>3$.

For $m>3$, consider any task set, m, s_m and list. Let $s=s_m$. Let $\hat{f}$ be the finish time of the corresponding list schedule and f^* the optimal finish time. We shall show that the following inequality holds.

$$\hat{f}/f^* \leqq 1+\frac{(m-1)}{s+m-1}\min\{s,2\}. \tag{24}$$

Since $s/(s+m-1)$ is an increasing function of s, it follows that the right hand side of (24) is maximized when $s=2$. Hence, (24) reduces to

$$\begin{aligned}\hat{f}/f^* &\leqq 1+\frac{2(m-1)}{m+1}\\ &=3-4/(m+1).\end{aligned}$$

We prove (24) by considering two cases. Let F_i be the finish time of P_i in the list schedule.

Case 1. $[\hat{f}=F_m]$. Let $F=\min_{i\neq m}\{F_i\}$. Clearly, the following inequality must hold:

$$(m-1)F+s\hat{f}\leqq\sum_1^n t_i$$

or

$$F\leqq\left(\sum_1^n t_i-s\hat{f}\right)\Big/(m-1). \tag{25}$$

Let t be the task time of the last task assigned to P_m. We may assume this is the last task in the list. If not, we can dispense with the remaining tasks and not reduce $\hat{f}/f^*$. We observe that $F+t\geqq\hat{f}$ and $sf^*\geqq t$. Using these and (25) we get:

$$sf^*\geqq\hat{f}-F\geqq\hat{f}-(\sum t_i-s\hat{f})/(m-1).$$

The preceding inequality together with the inequality $\sum t_i\leqq(s+m-1)f^*$ yields:

$$\hat{f}/f^*\leqq 1+s(m-1)/(s+m-1). \tag{26}$$

Now, let t' be the length of the last task assigned to P_m in the list schedule and which was not assigned to P_m in the optimal schedule. Clearly such a task exists as otherwise $\hat{f}=f^*$. Let $SUC(t')$ be the sum of the task lengths of the tasks assigned to P_m after t' in the list schedule. Clearly, $SUC(t')\leqq sf^*$ and $t'\leqq f^*$.
Also, $F+t'\geqq\hat{f}-SUC(t')/s$. Hence,

$$\begin{aligned}s(F+t')&\geqq s\hat{f}-SUC(t')\\ &\geqq s\hat{f}-sf^*\end{aligned}$$

or

$$F+f^*\geqq\hat{f}-f^* \quad\text{or}\quad F\geqq\hat{f}-2f^*$$

From this and (25) we obtain

$$\sum_1^n t_i\geqq(\hat{f}-2f^*)(m-1)+s\hat{f}. \tag{27}$$

Since $\sum t_i \leqq (s+m-1)f^*$, (27) results in (28)

$$\hat{f}/f^* \leqq 1 + \frac{2(m-1)}{s+m-1}. \tag{28}$$

Combining (26) and (28) we get (24).

Case 2. $[\hat{f} \neq F_m]$. In this case, $\hat{f} > F_m$. Without loss of generality, we may assume $\hat{f} = F_{m-1}$. Let t be the length of the last task assigned by P_{m-1}. As before, we can assume that this is the last task to be assigned to any processor. Note that no F_i, $i \neq m$ may be less than $\hat{f} - t$. If any F_i, $i < m-1$ is greater than $\hat{f} - t$ then we can decrease the processing requirements of the last few tasks assigned to P_i so that F_i is now equal to $\hat{f} - t$. This decrease will not change the list schedule but may decrease f^*. So $\hat{f}/f^*$ does not decrease and we will only be considering a worse case. Hence, we may assume $F = \hat{f} - t = F_i$, $1 \leqq i \leqq m-2$.

It is easy to see that $sF_m + \hat{f} + (m-2)F = \sum_1^n t_i \leqq (s+m-1)f^*$. Since $F + t = \hat{f}$ and $sF_m + t > s\hat{f}$, it follows that $sF_m + t + (m-2)(F+t) > (s+m-2)\hat{f}$. But, $sF_m + \hat{f} + (m-2)F = \sum t_i \leqq (s+m-1)f^*$. So, $(s+m-1)f^* - \hat{f} + (m-1)t > (s+m-2)\hat{f}$. This together with the equation $t \leqq sf^*$ yields

$$\hat{f}/f^* < 1 + \frac{s(m-1)}{s+m-1}. \tag{29}$$

Let t' be the length of the last task assigned to P_m in the list schedule and which is not assigned to P_m in f^*. If such a t' does not exist then $F_m \leqq f^*$. If the task with length t was on P_m in f^* then $\hat{f} \leqq F_m + t/s \leqq f^*$. Otherwise, $t \leqq f^*$ and $\hat{f} < F_m + t/s < 2f^*$. In either case $\hat{f}/f^* < 2$ and the theorem holds. So, we may assume t' exists. Let $SUC(t')$ be as defined in Case 1. Now, $F + t' \geqq \hat{f} - SUC(t')/s$. Since, $SUC(t') \leqq sf^*$, the previous inequality becomes $sF + st' \geqq s\hat{f} - sf^*$ or $F + t' \geqq \hat{f} - f^*$ or $F \geqq \hat{f} - 2f^*$.

Substituting $sF_m + t > \hat{f}$ and $\hat{f} - t = F$ into $sF_m + \hat{f} + (m-2)F \leqq (s+m-1)f^*$, we get

$$(m-1)F + s\hat{f} \leqq (s+m-1)f^*. \tag{30}$$

Using $F \geqq \hat{f} - 2f^*$ in (30) yields

$$\hat{f}/f^* \leqq 1 + \frac{2(m-1)}{s+m-1}. \tag{31}$$

(29) and (31) yield (24). This completes the proof for the upper bound of Theorem 3. The next example shows that the bound is tight.

Example 8. For any fixed m, $m \geqq 3$ let $n = (m-3)(m+1) + 3$ and $s_m = 2$, $s_i = 1$, $1 \leqq i < m$. The n task times are $t_i = 1/(m+1)$, $1 \leqq i \leqq n-3$, $t_{n-2} = t_{n-1} = 1$ and $t_n = 2$. By

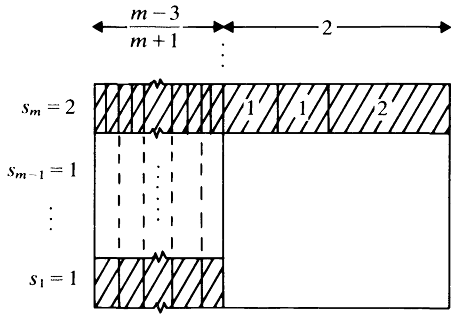


FIG. 5. *List schedule for Example* 8.

assigning task n to P_m, tasks $n-1$ and $n-2$ to P_{m-1} and P_{m-2} respectively and assigning $(m+1)$ of the remaining tasks to each of the remaining processors we get a schedule with finish time 1. It is easy to see that this is optimal and so $f^*=1$.

If the list for scheduling is $(1, 2, \cdots, n)$ then the resulting list schedule is as in Fig. 5. For every two tasks with index $\leqq n-3$ assigned to P_m, one task with index $\leqq n-3$ is assigned to each of the remaining processors. The finish time $\hat{f}$ is $(m-3)/(m+1)+4/2=3-4/(m+1)$.

REFERENCES

[1] 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.

[2] T. Gonzalez, O. H. Ibarra and S. Sahni, *Bounds for LPT schedules on uniform processors*, this Journal, 6 (1977), pp. 155–166.

[3] R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Tech. J., 45 (1966), pp. 1563–1581.

[4] ——— *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 263–269.

[5] R. L. Graham, E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, *Optimization and approximation in deterministic sequencing and scheduling: A Survey*, Mathematisch Centrum, Amsterdam, BW 82/.77, 1977.

[6] E. Horowitz and S. Sahni, *Exact and Approximate Algorithms for Scheduling Non-Identical Processors*, J. Assoc. Comput. Mach., (1976), pp. 317–327.

[7] ——— *Fundamentals of Computer Algorithms*, Computer Science Press, Maryland, 1978.

[8] J. W. S. Liu and C. L. Liu, *Bounds on Scheduling Algorithms for Heterogeneous Computing Systems*, Proc. IFIP, (1974), pp. 349–353.

[9] S. Sahni, *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23 (1976), pp. 116–127.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0008 $01.00/0

WORST CASE EXPONENTIAL LOWER BOUNDS FOR INPUT RESOLUTION WITH PARAMODULATION*

R. STATMAN†

Abstract. Input resolution with paramodulation is a theorem proving procedure complete for sets of unit clauses with equality. This procedure recommends itself because it is easy to implement, and several implementations are in use in more general theorem proving programs. In this note we show that input resolution with paramodulation requires, in the worst case, proofs of exponential length even though the satisfiability problem for sets of unit clauses can be solved in polynomial time.

Key words. input resolution, paramodulation, exponential, lower bounds

1. Introduction. General interest in the complexity of theorem proving procedures stems partly from Cook [5] and Cook and Reckhow [6] where it is shown that there is a polynomial bounded proof system for tautologies if and only if *NP* is closed under complementation. In Statman [4b] it is shown that there is a polynomial bounded intuitionistic proof s\ystem if and only if $NP = P$-space.

In this connection, particular interest in resolution based systems stems from Tseitin [7] where it is shown that "regular" resolution requires, in the worst case, proofs of exponential length on a class of sets of clauses whose satisfiability problem can be solved in polynomial time. Input resolution with paramodulation is a theorem proving procedure complete for sets of unit clauses with equality. This procedure recommends itself because it is easy to implement, and several implementations are in use in more general theorem proving programs (see e.g. Chang and Lee [1]). In this note we show that input resolution with paramodulation requires, in the worst case, proofs of exponential length even though the satisfiability problem for sets of unit clauses can be solved in polynomial time.

Our proof proceeds by simulating input resolution by a special class of nondeterministic pushdown stack automata. This class has not previously been discussed in the literature. We derive upper and lower time and space bounds for these automata, and these bounds apply automatically to input resolution. As a by product, we strengthen a result of Kozen [8].

2. Symmetric nondeterministic pushdown stack automata. A SNPDA is a nondeterministic pushdown stack automata without input all of whose transitions are reversible. More precisely, an SNPDA $M = (A, S, R)$ consists of a finite pushdown alphabet A, a finite set of states S, and a finite set of transition rules R. Each member of R has the form $s_1 \leftrightarrow as_2$ to be interpreted as meaning both (1) if in s_1 then push a and go into s_2, and (2) if in s_2 and a is topmost in the stack then pop a and go into s_1. Below we shall always assume that each member of $A \cup S$ is mentioned in R.

A configuration σs of M consists of a stack σ (possibly $= \phi$), where bottom to top = left to right, and a state s. An M computation from configuration τ_1 to configuration τ_n is a sequence $\tau_1, \cdots, \tau_n$ of configurations s.t. τ_{i+1} follows τ_i according to one of the rules of M.

The space of a computation γ is the maximum size of a stack in γ. The time of γ is the number of configurations in γ. The size of M, $|M|$, is the cardinality of R.

A SNPDA homomorphism h from $M_1 = (A_1, S, R_1)$ to $M_2 = (A_2, S_2, R_2)$ is a map $h: A_1 \to A_2$ and $h: S_1 \to S_2$ s.t. $s_1 \leftrightarrow as_2 \in R_1 \Rightarrow h(s_1) \leftrightarrow h(a)h(s_2) \in R_2$.

* Received by the editors June 26, 1978, and in revised form January 19, 1979.

† Department of Mathematics, Rutgers University, New Brunswick, New Jersey 08903.

3. Upper and lower bounds in runtime and space.

DEFINITIONS. Let $\gamma_1 = \alpha_1, \cdots, \alpha_n$ and $\gamma_2 = \beta_1, \cdots, \beta_m$ be computations and $\sigma \in A^*$

(1) $\gamma_1^{-1} = \alpha_n, \cdots, \alpha_1$;

(2) $\sigma \searrow \gamma_1 = \sigma\alpha_1, \cdots, \sigma\alpha_n$ (intuitively, "put σ below γ_1");

(3) $\gamma_1 \searrow \gamma_2 = \alpha_1, \cdots, \alpha_{n-1}, \sigma \searrow \gamma_2$ if $\sigma_n = \sigma\beta_1$
$= \gamma_1$ otherwise

(4) $\gamma_1 \swarrow \gamma_2 = \alpha_1, \cdots, \alpha_{n-1}, \sigma \searrow \gamma_2$ if $\sigma_n = \sigma\beta_1$
$= \gamma_2$ otherwise.

PROPOSITION. *Let $M = (A, S, R)$ be a SNPDA with $s', s'' \in S$; if there is an M computation from s' to s'' then there is one with space $\leqq |M| \cdot 2$.*

Our proof is motivated by the construction of minimal machines from finite automata. Call states s_1 and s_2 of M equivalent if there is an M-computation from s_1 to s_2. We shall construct a system of representatives for the equivalence classes of S in such a manner that there is an M-computation of space $\leqq |S|$ from each state to its representative. In particular, we shall construct an SNPDA M^* with states from S and an SNPDA homomorphism h from M onto M^* determined by certain cannonical M computations. M^* is deterministic with respect to pop. The definition of h insures that each state of M has a representative among the states of M^*. The determinism of M^* insures that these representatives are unique.

We proceed inductively constructing at stage n a set of computations Γ_n, computations $\gamma_n(s)$ for $s \in S$, and a computation γ_n.

Stage 1. $\Gamma_1 = S \cup \{s_1, as_2 : s_1 \leftrightarrow as_2 \in R\}$, $\gamma_1(s) = s$, and $\gamma_1 = \phi$.

Stage $k+1$. If there are computations $\delta_1, \delta_2 \in \Gamma_k$ from s_1 to σs and s_2 to σs resp. for $s_1 \neq s_2$, set $\gamma_{k+1} = \delta_1 \searrow \delta_2^{-1}$. Let $\gamma_{k+1}(s) = \gamma_k(s) \searrow \gamma_{k+1}$ and put $\Gamma_{k+1} = \{\gamma_{k+1}^{-1} \swarrow (\gamma \searrow \gamma_{k+1}) : \gamma \in \Gamma_k\}$. Otherwise stop.

Let $S_n = \{s \in S : s$ begins or ends a member of $\Gamma_n\}$; the following are easy to see.

(1) $\gamma_n(s)$ begins with s and ends in S_n, and γ_n and any $\gamma \in \Gamma_n$ begin in S and end with some σs for $s \in S$ and $\sigma = \phi$ or $\sigma \in A$.

(2) Space $(\gamma_n(s)) \leqq n$ and $\gamma \in \Gamma_n \Rightarrow$ space $(\gamma) \leqq n$.

(3) $|S_n| \leqq |S| - n + 1$.

Thus the construction stops at some $n_0 \leqq |S| \leqq |M| \cdot 2$. Let $S^* = S_{n_0}$, $\Gamma = \Gamma_{n_0}$ and $\gamma(s) = \gamma_{n_0}(s)$. Observe that

(4) if $\gamma \in \Gamma$ has both ends in S then they are the same.

(5) If $\gamma_1, \gamma_2 \in \Gamma$ end in the same configuration then they both begin with the same member of S^*.

Define an SNPDA $M^* = (A, S^*, R^*)$ where $s_1 \leftrightarrow as_2 \in R^* \Leftrightarrow_{df}$ there is a computation from s_1 to as_2 in Γ. Set $h(s) =_{df}$ end of $\gamma(s)$ for $s \in S$, then h is a SNPDA homomorphism of M onto M^* which is the identity on S^*. Now M^* is deterministic w.r.t. *pop* so if there is an M computation from s_1 to s_2 then $h(s_1) = h(s_2)$ so $\gamma(s_1) \searrow \gamma(s_2)^{-1}$ is an M computation from s_1 to s_2 with space $\leqq |M| \cdot 2$.

COROLLARY. *If there is an M computation from s_1 to s_2 then there is one of time $\leqq 2^{|M|((\log|M| \cdot 2)+1)}$.*

COROLLARY. *If there is an M computation from s_1 to s_2 and $|A| = 1$ then there is one of time $\leqq 4 \cdot |M|^2$.*

COROLLARY. *The following problem can be solved in polynomial time: Given SNPDA $M = (A, S, R)$ and $s_1, s_2 \in S$, is there an M computation from s_1 to s_2?*

Proof. The proof is by inspection of the proof of the proposition.

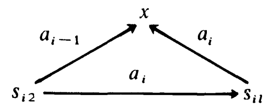
The reader may have observed that a straightforward dynamic programming argument gives the preceding corollary and a space bound of $|M|^2$ for the proposition.

However if this bound is plugged into the first corollary, the resulting time bound is not polynomial in the bound stated there.

Define SNPDA $M_n = (A_n, S_n, R_n)$ by $A_n = \{a_i : 0 \leqq i \leqq n+1\}$, $S_n = \{s_{ij} : 1 \leqq i \leqq n$ and $1 \leqq j \leqq 2\} \cup \{u, v, x, y\}$, and $R_n = \{s_{i1} \leftrightarrow a_i x, s_{i2} \leftrightarrow a_i s_{i1}, s_{i2} \leftrightarrow a_{i-1}x : 1 \leqq i \leqq n\} \cup \{y \leftrightarrow a_0 x, v \leftrightarrow a_n x, v \leftrightarrow a_{n+1}u, x \leftrightarrow a_{n+1}u\}$. We shall give an exponential lower bound on the time of an M_n computation from x to y. In order to motivate the proof let us represent SNPDA $M = (A, S, R)$ by a directed graph as follows: the vertices of the graph are the members of S, and the edges (labeled by members of A) are defined by $s_1 \xrightarrow{a} s_2$ is an edge if $s_1 \leftrightarrow as_2 \in R$. In this way we can always cross the edge from s_1 to s_2 by pushing a, but we can only cross the edge from s_2 to s_1 (contrary to the direction of the arrow) if a is at the top of the stack. The graph representing M_n is the union of the following:

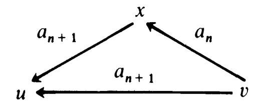
$y \xrightarrow{a_0} x$;

T_i;



for $1 \leqq i \leqq n$, and

T_{n+1};



In order to go from x to y one must first go around T_{n+1} counterclockwise leaving a stack a_n. In order to get rid of a_n one must go around T_n clockwise; thus one must go around T_{n+1} again to get a stack $a_n a_n$ before going around T_n, finally leaving a stack a_{n-1}. In order to get rid of a_{n-1} one must go around T_{n-1} clockwise; thus one must repeat the whole trip so far to get a stack $a_{n-1}a_{n-1}$ before going around T_{n-1}, finally leaving a stack a_{n-2}, etc. More precisely, let γ be a shortest M_n computation from x to y and set $\delta = x, a_{n+1}u, v, a_n x, a_n a_{n+1} u, a_n v, a_n a_n x, a_n s_{n1}, s_{n2}$. The following three lemmas provide a decomposition of γ.

LEMMA. *Suppose* $\gamma = \delta_1, \sigma x, \sigma a_{n+1} u, \delta_2$, *then* $\sigma x, \sigma a_{n+1}u, \delta_2 = (\sigma \searrow \delta, a_{n-1}x), \delta_4$ *for some* δ_4, *provided* σ *does not end in* a_n.

Proof. Since γ is shortest $\gamma = \delta_1, \sigma x, \sigma a_{n+1}u, \sigma v, \sigma a_n x, \cdots, \sigma a_n^k x, \sigma a_n^{k-1} s_{n1}, \sigma a_n^{k-2} s_{n2}, \sigma a_n^{k-2} a_{n-1} x, \delta_4$ for some δ_4, and $2 \leqq k$. We show $k = 2$. Suppose not; then δ_4 contains a part of the form $\sigma a_n w_1, \sigma w_2$ for $w_1, w_2 \in S_n$. Clearly, $w_1 \in \{x, s_{n1}\}$ so since γ is shortest $w_1 = s_{n1}$ and $w_2 = s_{n2}$. This contradicts the choice of γ.

LEMMA. $\gamma = x, \delta_n, s_{n,2}, a_{n-1}x, \delta_{n-1}, s_{n-1,2}, a_{n-2}x, \cdots, \delta_1, s_{1,2}, a_0 x, y$ *for some* $\delta_i 1 < i < n$.

Proof. We have $\gamma = \delta, a_{n-1}x, \eta, y$ for some η and this determines δ_n. Suppose now that $\gamma = x, \delta_n, s_{n,2}, a_{n-1}x, \cdots, a_k x, \delta_k, s_{k,2}, a_{k-1}x, \eta, y$. Now η does not begin with a state w for such a $w \in \{s_{k,2}, s_{k-1,1}\}$ and by shortness of γ, $w = s_{k-1,1}$ but then η begins with $s_{k-1,1}, a_{k-1}x$ contradicting the choice of γ. Thus either $k - 1 = 0$ and we are done or η contains $a_{k-1}w_1, w_2$ for $w_1, w_2 \in S_n$. As in the preceding lemma $w_1 = s_{k-1,1}$ and $w_2 = s_{k-1,2}$ is followed by $a_{k-2}x$. The lemma follows easily by induction.

Let $\gamma_k =_{df} x, \delta_n, s_{n,2}, a_{n-1}x, \cdots, a_k x, \delta_k, s_{k,2}$.

LEMMA. $a_k x, \delta_k, s_{k,2} = (a_k \searrow x, \eta_k, a_k x), a_k s_{k,1}, s_{k,2}$ *for suitable* η_k.

Proof. Since γ is shortest $a_k x, \delta_k, s_{k,2} = \lambda, a_k s_{k,1}, s_{k,2}$ for some λ. By shortness of γ, λ contains no part $a_k w_1, w_2$ for $w_1, w_2 \in S_n$ so $\lambda = a_k \searrow x, \eta_k, a_k x$ for suitable η_k.

PROPOSITION. *Any M_n computation from x to y takes time* $\geqq 2^{|M_n|_{n/3}+1} - 1$.

Proof. Since γ is shortest, time $(a_k x, \delta_k, s_{k,2}) \geqq$ time (γ_{k+1}) so time $(\gamma_k) \geqq 2 \cdot$ time $(\gamma_{k+1}) + 3$. Time $(\gamma_n) = 9$ so time $(\gamma) \geqq 6 \cdot 2^n - 1$.

Define SNPDA $M_{nm} = (A_{nm}, S_{nm}, R_{nm})$ by $A_{nm} = \{a\}$, $S_{nm} = \{s_i : 1 \leqq i \leqq n\} \cup \{t_i : 1 \leqq i \leqq m\} \cup \{r_i : 1 \leqq i \leqq m\} \cup \{x, y\}$, and $R_{nm} = \{s_{i+1} \leftrightarrow as_i : 1 \leqq i \leqq n\} \cup \{t_i \leftrightarrow ar_i, t_{i+1} \leftrightarrow ar_i : 1 \leqq i \leqq m\} \cup \{tm \leftrightarrow ar_m, x \leftrightarrow ar_m, y \leftrightarrow as_n, s_1 \leftrightarrow ax, t_1 \leftrightarrow ax\}$.

The following establishes that the space bounds, and the time bounds for $|A| = 1$ are optimal.

PROPOSITION. *Any M_{nm} computation from x to y takes time* $\geqq (nm+n)2+1$ *and space* $\geqq n+1$.

Proof. The proof is by inspection.

COROLLARY. *Any M_{n1} computation from x to y requires space* $\geqq |M_{n1}| - 3$.

COROLLARY. *Any $M_{nn/2}$ computation from x to y takes time* $\geqq |M_{nn/2}|^2 \cdot \frac{1}{4}$.

A linear sentence is an object of the form $\Gamma \to p$ where Γ is a finite set of propositional variables and p is a propositional variable. If $\Gamma = \{p_1, \cdots, p_n\}$ we write $p_1, \cdots, p_n \to p$ for $\Gamma \to p$ (Γ is the antecedent of $\Gamma \to p$). The interpretation of $p_1, \cdots, p_n \to p$ is $(p_1 \wedge \cdots \wedge p_n) \supset p$. (The use of "linear sentence" here is an abuse of the terminology in Gentzen [2].)

By using the path problem (see Jones and Laasser [3]) it is easy to see that the following problem is log-space complete for polynomial time.

Given a set L of linear sentences s.t. each antecedent of a member of L has size $\leqq 2$ and a propositional variable p, is $L \models p$?

A set L of linear sentences is normal if (1) each antecedent has size $\leqq 2$; (2) no variable occurs in more than two antecedents; (3) if a variable occurs in an antecedent of size $= 2$ it occurs in no other antecendent.

Suppose L satisfies (1), we define a set of linear sentences L^1 satisfying (1) and (3) as follows:

$$\to p \in L \Rightarrow \to p \in L^1,$$

$$p \to q \in L \Rightarrow p \to q \in L^1,$$

$$p, q \to r \in L \Rightarrow p \to p_s, q \to q_s, p_s, q_s \to r \in L^1, \quad \text{for } s = p, q \to r \text{ and new variables } p_s, q_s.$$

Clearly L^1 can be obtained from L in log-space and for $p \neq p_s$, $L \models p \Leftrightarrow L^1 \models p$. We now define L^2 satisfying (1), (2), and (3) as follows:

$$\to p \in L^1 \Rightarrow \to p \in L^2;$$

$$p, q \to rL^1 \Rightarrow p, q \to r \in L^2;$$

if $s_1 = p \to q_1, \cdots, s_n = p \to q_n$ are all members of L^1 with antecedent p then $p \to p_{s_n}, \cdots, p_{s_3} \to p_{s_2}, p_{s_3} \to q_s, p_{s_2} \to p_{s_1}, p_{s_2} \to q_s, p_{s_1} \to q_1 \in L^2$, for new variables $p_{s_1} \cdots p_{s_n}$.

It is easy to see that L^2 can be obtained from L^1 in log-space and for $p \neq p_s$, $L^1 \models p \Leftrightarrow L^2 \models p$. We obtain the following:

LEMMA. *The following problem is log-space complete for polynomial time. Given a normal set L of linear sentences and a variable p, is $L \models p$?*

Let L be a normal set of linear sentences. We define a SNPDA $M_L = (A_L, S_L, R_L)$ by $A_L = \{a, b\}$, $S_L = \{s_p s'_p s''_p : p \text{ occurs in a member of } L\} \cup \{x_l x'_l x''_l : l = p, q \to r \in L\}$, and

R_L defined by

$$\to p \in L \Rightarrow s_p \leftrightarrow as''_p,\ s'_p \leftrightarrow as''_p \in R_L$$

if $p \to q$, $p \to r$ are all the sentences in L with antecedent p then

$$s_q \leftrightarrow bs_p,\ s'_q \leftrightarrow bs'_p,\ s_r \leftrightarrow as_p,\ s'_r \leftrightarrow as'_p \in R_L.$$

$$l = p, q \to r \in L \Rightarrow x_l \leftrightarrow as_p,\ x''_l \leftrightarrow as'_p,\ x''_l \leftrightarrow as'_q,\ x'_l \leftrightarrow as_q,\ s'_r \leftrightarrow ax'_l,$$

$$s_r \leftrightarrow ax_l \in R_L.$$

Clearly M_L can be obtained from L in log-space.

LEMMA. *If there is an M_L computation from s_p to s'_p then $L \vDash p$.*

Proof. Suppose ν is a truth value assignment satisfying L. Let $\sim$ be the smallest equivalence relation on the states of M_L satisfying $\nu p = T \Rightarrow s_p \sim s'_p$, and, for $l = p, q \to r$, $\nu p = T \Rightarrow x_l \sim x''_l$, $\nu q = T \Rightarrow x'_l \sim x''_l$. Define SNPDA $M^* = (A_L, S^*, R^*)$ by $S^* = \{s/\sim : s \in S_L\}$, $R^* = \{s_1/\sim \leftrightarrow as_2/\sim,\ s_3/\sim \leftrightarrow bs_4/\sim : s_1 \leftrightarrow as_2,\ s_3 \leftrightarrow bs_4 \in R_L\}$, then the map h defined by $h(s) = s/\sim$, $h(a) = a$ and $h(b) = b$ is a SNPDA homomorphism from M_L onto M^*.

It is easy but tedious to verify that M^* is deterministic w.r.t. *pop*. Thus if there is an M^* computation from $s_1/\sim$ to $s_2/\sim s_1/\sim\ = s_2/\sim$. If $\nu p = F$ then $s_p \not\sim s^0_p$ so there is no M computation from s_p to s'_p.

LEMMA. $L \vDash p \Rightarrow$ *there is an M_L computation from s_p to s'_p.*

Proof. Define a truth value assignment by ν by $\nu p = T \Leftrightarrow$ there is an M_L computation from s_p to s'_p. It is easy to see that ν satisfies L. Thus if there is no M_L computation from s_p to s'_p then $L \nvDash p$.

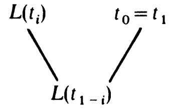
PROPOSITION. *The following problem is log-space complete for polynomial time. Given* SNPDA $M = (A, S, R)$ *and* $s_1, s_2 \in S$, *is there an M computation from s_1 to s_2?*

Note that the above proof shows that completeness is attained even if we restrict A to $|A| \leqq 2$, this sharpens the result of Kozen [8, p. 256, Cor. 3.1.10] to equations containing at most two unary function symbols (see below).

4. Applications to input paramodulation. Let C be a set of ground unit clauses with only unary function symbols, monadic predicate symbols, and $=$; an input refutation with resolution and paramodulation is defined as in Chang and Lee [1, p. 173] except

(1) arbitrary instances of $x = x$ are allowed, and

(2) paramodulations can have the form



where $L(t_{1-i})$ is obtained from $L(t_i)$ by replacing 0 or more occurrences of t_i by t_{1-i} for $0 \leqq i \leqq 1$.

More precisely, we consider a proof system with axioms $t = t$ for each term t, the rule of resolution (ground), and the rule of paramodulation (2). If C is a set of clauses an input derivation from C is a sequence of clauses $C_1, \cdots, C_n$ such that C_1 is an axiom or a member of C, and C_{i+1} follows from C_i and an axiom or a member of C by resolution or paramodulation. An input refutation of C is an input derivation from C ending in the empty clause. Below we shall represent input derivations in tree form making explicit both premisses of each inference. If C is a set of (ground) unit clauses which is unsatisfiable then there is an input refutation of C (see [1, Lemma 8.3, p. 176]).

C is called simple if

(1) each equation in C has the form $c_1 = fc_2$ for $c_1 \neq c_2$,

(2) each inequation in C has the form $c_1 \neq c_2$,

(3) each other literal in C has the form $\pm Pc$.

If C is not simple define a simple C^* as follows: Select a function symbol g from C, for each equation $E \in C$ a new constant c_E, for each occurrence t of a nonconstant term in C a new constant c_t, and set $c_c = c$. $C^* =_{df} \{c_{ft} = fc_t : ft \text{ an occurrence in } C\} \cup \{c_{t_1} = gc_{t_1=t_2}, c_{t_2} = gc_{t_1=t_2} : t_1 = t_2 \in C\} \cup \{c_{t_1} \neq c_{t_2} : t_1 \neq t_2 \in C\} \cup \{\pm Pc_t : \pm Pt \in C\}$.

Notice that C^* can be obtained from C in log-space.

LEMMA. C is satisfiable $\Leftrightarrow C^*$ is satisfiable.

Proof. The proof is routine.

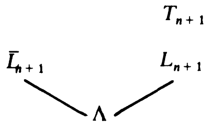
If C is simple we define an SNPDA $M_C = (A_C, S_C, R_C)$ by $A_C =$ function symbols in C, $S_C =$ constants in C, $R_C = \{c_1 \leftrightarrow fc_2 : c_1 = fc_2 \in C\}$. For SNPDA $M = (A, S, R)$ define $C_M = \{s_1 = as_2 : s_1 \leftrightarrow as_2 \in R\}$.

PROPOSITION. *Suppose C is simple and has an input refutation of length l then either*

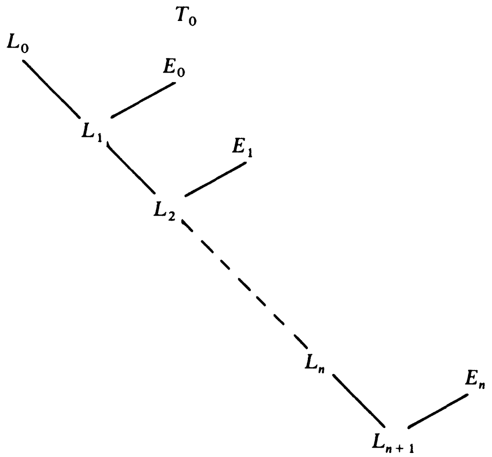
(1) there are $\neg Pc_1, Pc_2 \in C$ with an M_C computation from c_1 to c_2 with time $\leqq 2 \cdot l$ or

(2) there is $c_1 \neq c_2 \in C$ with an M_C computation from c_1 to c_2 with time $\leqq 2 \cdot l$.

Proof. Let T be the shortest input refutation of C and let $T =$



for $\bar{L}_{n+1} \in C$. In case $L_{n+1} \in C$ we are done. Otherwise $T_{n+1} =$

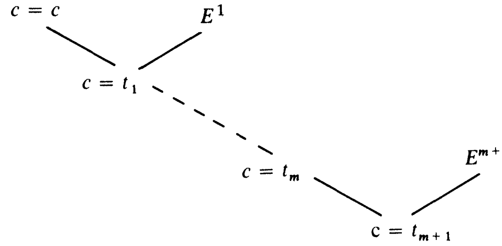


where T_0 is an input derivation of E_0 from C by paramodulation and

(1) $E_0 = L_{n+1}$ and the L_i for $0 \leqq i \leqq n$ and the E_j for $1 \leqq j \leqq n$ do not exist, or

(2) $L_0, E_1, \cdots, E_n \in C$.

In either case E_0 has the form $c = t$ or $t = c$. By Proposition 3.4 of Statman [4a] there is an input derivation by paramodulation of $c = t$ from C of the form



where $t_{m+1} = t$ and $m + 1 \leqq l$, and $E^1, \cdots, E^{m+1} \in C$.

In case 1, since C is simple t_{m+1} is a constant and the desired computation is $c, t_1, \cdots, t_{m+1}$. In case 2, L_{n+1} begins with $\pm P$. Let $L_i = \pm Ps_i$, so $s_0 = c$, $s_1 = t_{m+1}$ and s_{n+1} is a constant. The desired computation is $\gamma =_{df} c, t_1, \cdots, t_{m+1}, s_2, \cdots, s_{n+1}$ or γ^{-1}.

We have the following converse:

CONVERSE. *Suppose C is simple and there is an M_C computation from c_1 to c_2 with time l s.t. either*

(1) $\neg Pc_1, Pc_2 \in C$ *or*

(2) $c_1 \neq c_2 \in C$.

Then C has an input refutation of length $\leqq l + 1$.

Proof. The proof is easy.

THEOREM. *Input resolution with paramodulation requires in the worst case, proofs of exponential length.*

Proof. Let $C_n = C_{M_n} \cup \{\neg Px, Py\}$. There is no input refutation of C_n with length $< 2^{|C_n|/3} - 1$.

THEOREM. *The following problem is complete for polynomial time*:

Given a set C of unit clauses, is C satisfiable?

Proof. (1. Polynomial time). Given C construct C^* and $M =_{df} M_{C^*}$. For each pair $\neg Pc_1, Pc_2 \in C^*$ see if there is an M computation from c_1 to c_2, and for each $c_1 \neq c_2 \in C^*$ see if there is an M computation from c_1 to c_2. (2. Completeness). If L is a normal set of linear setences set $C_L = C_{M_L}$. Then, $L \models p \Leftrightarrow C_L \cup \{\neg Ps_p, Ps'_p\}$ is unsatisfiable.

Acknowledgment. The author would like to thank Professor S. A. Cook and the referees for their valuable comments and suggestions.

REFERENCES

[1] CHANG AND LEE, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[2] GENTZEN, *On the existence of independent axiom systems for infinite sentence systems*, The Collected Papers of Gerhard Gentzen, Szabó, ed., North-Holland, Amsterdam, 1969.

[3] JONES AND LAASSER, *Complete problems for deterministic polynomial time*, Theoret. Comput. Sci., 3 (1977).

[4a] STATMAN, *Herbrand's theorem and Gentzen's notion of a direct proof*, Handbook of Mathematical Logic, Barwise, ed., North-Holland, Amsterdam, 1977.

[4b] ———, *Intuitionistic propositional logic is P-space complete*, Theoret. Comput. Sci., to appear.

[5] COOK, *The complexity of theorem proving procedures*, Proceedings 3rd Annual A.C.M.S.T.O.C. (May 1971).

[6] COOK AND RECKHOW, *On the length of proofs in the propositional calculus*, Proceedings 6th Annual A.C.M.S.T.O.C. (April–May 1974).

[7] TSEITIN, *On the complexity of derivation in propositional calculus*, Studies in Constructive Math. and Math. Logic II, Slisenko, ed., 1968.

[8] KOZEN, *Lower bounds for natural proof systems*, Proc. 18th Annual F.O.C.S. Symposium (I.E.E.E.), October–November 1977.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0009 $01.00/0

AN EFFICIENT METHOD FOR WEIGHTED SAMPLING WITHOUT REPLACEMENT*

C. K. WONG† AND M. C. EASTON†

Abstract. In this note, an efficient method for weighted sampling of K objects without replacement from a population of n objects is proposed. The method requires $O(K \log n)$ additions and comparisons, and $O(K)$ multiplications and random number generations while the method proposed by Fagin and Price requires $O(K n)$ additions and comparisons, and $O(K)$ divisions and random number generations.

Key words. sampling without replacement, algorithm, computational complexity

1. Introduction. In [1], Fagin and Price consider the following experiment, which can be called weighted sampling without replacement. The experiment is described as the drawing of balls from an urn without replacement. Assume that an urn contains n balls numbered $1, \cdots, n$ and that the probability of drawing ball i is p_i ($i = 1, \cdots, n, \sum_i p_i = 1$). Suppóse that ball i_1 is selected first. Now renormalize the probabilities of the remaining $(n-1)$ balls so that they sum to 1. Thus, the probability of drawing ball j becomes $p_j/(1-p_{i_1})$, for $j \neq i_1$. Select a second ball from the urn, say, ball i_2. Again renormalize the probabilities of the remaining $(n-2)$ balls so that they sum to 1. Thus, the probability of drawing ball j becomes $p_j/(1-p_{i_1}-p_{i_2})$, for $j \neq i_1, i_2$. Continue the process until K balls have been selected. Fagin and Price use this experiment in the Monte Carlo evaluation of a combinatorial sum.

Another application of the experiment occurs in the design of sampling procedures. A sampling system called *multistage sampling with probability proportional to size* (PPS) is discussed in [2, p. 283]. When sampling human, animal, or plant populations, it is often important to first partition the populations into units within which the variation may be expected to be less than it is overall. In two-stage sampling, for example, first a unit is selected and then individuals within that unit are selected at random. In PPS sampling, the selection of the unit is done with the probability of selection proportional to the size of the population of the unit. The selection of K units is to be carried out. After a unit has been selected, it is no longer eligible for later selection. This can be done by the urn experiment considered here if we represent each unit by a ball which has associated probability equal to the fraction of the total population that is contained in the unit.

A special case of the urn experiment occurs when all p_i's are equal and is referred to as "sampling without replacement" in [3, p. 132]. A very efficient method for this case is described in [4, p. 125], which requires $O(K)$ multiplications, additions, random number generations and computations of the floor function $\lfloor \ \rfloor$. The method for the case of unequal p_i's described in [1] requires $O(Kn)$ additions and comparisons and $O(K)$ divisions and random number generations. Another method for the unequal p_i's case, due to D. B. Lahiri, is described in [6, p. 347]. The number of operations required by Lahiri's method is a function of the probabilities $(p_1, \cdots, p_n)$. If the distribution is "flat", performance approaches that of the unweighted algorithm in [4]. On the other hand, if the probabilities, say, follow Zipf's law [7]: $p_i = c/i$ where $c = 1/\sum_{i=1}^n i$, then $O(K n/\log n)$ operations are required. In the worst case, $O(K n)$ operations are required. All methods described require $O(n)$ operations of initialization before the drawing of balls can begin.

* Received by the editors July 7, 1978 and in revised form February 5, 1979.

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

In this note, we describe a method for performing weighted sampling that requires, after $O(n)$ initialization operations, $O(K \log n)$ additions and comparisons and $O(K)$ divisions and random number generations. After a sample of K balls has been drawn, the experiment can be restored to its original state by $O(K \log n)$ operations. Thus, this technique is well suited to the Monte Carlo application in [1], where the experiment is repeated many times.

2. Method. The contents of the urn at any time are described by values $\{p'_1, p'_2, \cdots, p'_n\}$. In general, $p'_i = p_i$ if ball i is still in the urn; $p'_i = 0$ if it has been removed. We maintain values $S_i = \sum_{k=1}^{i} p'_k$, $i = 0, \cdots, n$. Let $Q = S_n$ be the sum of the $\{p_i\}$ of the balls remaining in the urn.

The method of drawing a ball is as follows. Choose x with uniform probability from $[0, Q]$. (This step requires generation of a random number and multiplication.) Then find j such that $S_{j-1} \leqq x < S_j$. The ball to be drawn is ball j. It is easy to verify that the probability that ball j is drawn is p'_j/Q.

We now describe an efficient method for implementation. There are two aspects of the implementation‘that contribute to the efficiency. The first is the method for finding j such that $S_{j-1} \leqq x < S_j$. The second is the method for reflecting the change in value of p'_j to zero in the sums $\{s_j\}$.

3. Initialization. The search for the index of the selected ball is carried out by means of a binary search tree. As part of the initialization procedure, we construct a binary tree having n leaves (external nodes) and having maximum path length from root to leaf of $O(\log n)$. Methods for constructing such trees are described in [5]. The construction requires $O(n)$ operations.

The leaves of the tree are labeled, from left-to-right: $1, 2, \cdots, n$. The other nodes are given arbitrary labels that are distinct from each other and from the leaf labels.

As part of the initialization, we associate values G_i and H_i with internal node i, $i = 1, 2, \cdots, n-1$. Let $L(i)$ be the left-descendent of node i. Let $R(i)$ be the right-descendent of node i.

If node i has a leaf as left-descendent, then define $G_i = p_{L(i)}$. Otherwise, $G_i = G_{L(i)} + H_{L(i)}$. If node i has a leaf as right-descendent, then define $H_i = p_{R(i)}$. Otherwise $H_i = G_{R(i)} + H_{R(i)}$. The computation of the values of $\{G_i\}$ and $\{H_i\}$, carried out from "bottom to top" of the tree, requires $O(n)$ additions.

It is not hard to see that G_i is the sum of values of $\{p_j\}$ that have indices on leaves of the left subtree from node i. (See Fig. 1, where G_i values are shown at each node.) H_i is the sum of values of $\{p_j\}$ that have indices on leaves of the right subtree from node i. (The H_i values are used only in the initialization.)

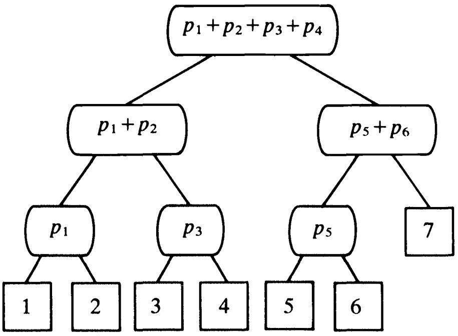


FIG. 1. *Example tree with $n = 7$ (sums are G_i values).*

At initialization time, G_i also is the sum of values of $\{p'_j\}$ that have indices on leaves of the left subtree from node i. It is this property that will be preserved as balls are drawn.

4. The algorithm. At each step of the algorithm a ball is selected as follows. The first node visited is the root node of the tree. Begin with $C=0$. At each node i:

```
**if** $x < G_i + C$ **then** move to node/leaf $L(i)$.
**else** set $C = C + G_i$; move to node/leaf $R(i)$.
```

The procedure ends when a leaf is reached. The label of this leaf gives the index of the chosen ball. $O(\log n)$ additions and comparisons are required for this procedure.

At each node, the value of $C+G_i$ gives the total of $\{p'_i\}$ values in leaves of parts of the tree that lie to the left of the current position.

By the structure of the tree, if the final leaf chosen is leaf j then the value of x satisfies:

$$S_{j-1}=p'_1+p'_2+\cdots+p'_{j-1}\leqq x<p'_1+p'_2+\cdots+p'_j=S_j.$$

In descending the tree, we keep track of which nodes are departed from *via a left branch*. Once j has been found, the value of G_i for each such node is decremented by the amount p_j in order to reflect the removal of the ball. (This is equivalent to setting the value of p'_j to zero.) The value of Q is also decremented by p_j.

A list is maintained of all changes (index, value) made in $\{G_i\}$. In the course of selecting K balls, at most $O(K \log n)$ entries are made on this list. At the conclusion of the experiment, this list is used to reinitialize the values of $\{G_i\}$. (If K is sufficiently large, of course, it is better to rerun the initialization procedure after each experiment.)

It follows that, once initialization has been completed, additional experiments of drawing K balls from a "full" urn can be performed at the expense of $O(K \log n)$ additions and comparisons and $O(K)$ multiplications and random number generations per experiment.

In summary, the algorithm works as follows to draw the next ball.

(a) Select x uniformly from $[0, Q]$ where Q is the sum of the probabilities of the remaining balls.

(b) Let $C=0$. Traverse the tree from the root down. At node m, branch right if $x \geqq G_m + C$, left if $x < G_m + C$. On left branches, record the label of the node. On right branches, increment C by G_m.

(c) When leaf j has been reached, decrement by p_j the values of $\{G_i\}$ for the nodes that were listed. The ball drawn is ball j.

Acknowledgment. Our use of sums of probabilities in the tree structure has been simplified by a suggestion of a referee.

REFERENCES

[1] R. FAGIN AND T. G. PRICE, *Efficient calculation of expected miss ratios in the independent reference model*, this Journal, 7 (1978) pp. 288–297.

[2] N. L. JOHNSON AND S. KOTZ, *Urn Models and Their Application*, John Wiley, New York, 1977.

[3] W. FELLER, *An Introduction to Probability Theory and Its Applications*, vol. 1, John Wiley, New York, 1970.

[4] D. KNUTH, *The Art of Computer Programming*, vol. 2, Addison-Wesley, Reading, MA, 1969.

[5] ———, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.

[6] F. YATES, *Sampling Methods for Censuses and Surveys*, Hafner, New York, 1960.

[7] G. K. ZIPF, *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Reading, MA, 1949.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

ON THE SUCCINCTNESS OF DIFFERENT REPRESENTATIONS OF LANGUAGES*

J. HARTMANIS†

Abstract. The purposes of this paper is to give simple new proofs of some interesting recent results about the relative succinctness of different representations of regular, deterministic and unambiguous context-free languages and to derive some new results about how the relative succinctness of representations change when the representations contain a formal proof that the languages generated are in the desired subclass of languages.

Key words. representation of languages, succinctness, context-free languages, deterministic languages, unambiguous languages, verified representations, Turing machines, valid computations

Introduction. It has been shown recently that there exist dramatic compressions of the length of representations of languages in subclasses of context-free languages as we go from restricted to unrestricted representations of these languages [3], [5], [6]. For example, when we consider the representation of deterministic context-free languages by deterministic versus nondeterministic pushdown automata, then there is no recursive function which can bound the size of the minimal deterministic pushdown automaton as a function of the size of the equivalent minimal nondeterministic pushdown automaton [6]. It is well known that we cannot recursively decide whether a given pushdown automaton has an equivalent deterministic pushdown automaton, but the above result makes a considerably stronger statement: even if we would know (or be given) which pushdown automata describe deterministic languages, we still could not effectively write down the corresponding deterministic pushdown automata because of their enormous size which grows nonrecursively in the size of the nondeterministic pushdown automata. Therefore we see that though nondeterminism is not needed in the description of deterministic context-free languages its use in the description permits nonrecursively bounded shortening of infinitely many representations.

Similar results hold for the relative succinctness of the description of unambiguous context-free languages by unambiguous and ambiguous context-free grammars [5], and the description of finite or regular sets by finite automata and pushdown automata [3].

Some of the original proofs of these results are quite hard and they require special results about context-free languages. In the first part of this paper we give a very simple, elementary proof that the relative succinctness of representing deterministic context-free languages by deterministic or nondeterministic pushdown automata is not recursively bounded, and using a result about inherently ambiguous context-free languages and Turing machine computations [4], derive an equally simple proof for the representation of unambiguous context-free languages by unambiguous or ambiguous context-free grammars. The results about the representation of finite and regular sets can be easily proven by the same methods.

We observe that in the representation of deterministic context-free languages by deterministic pushdown automata we can easily check whether a given pushdown automaton is deterministic, on the other hand, for a nondeterministic pushdown automaton we have no uniform way of verifying that it accepts a deterministic context-free language. Therefore the question arises whether the relative succinctness of the two representations is caused by the fact that in one representation we can prove what we are accepting but that no such proofs are possible in the other representation.

* Received by the editors July 10, 1978, and in final revised form March 9, 1979.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. This work was supported in part by National Science Foundation under Grant DCR75-09433 and MCS 78-00418.

Indeed a close inspection of the original proof [6] reveals that it does not hold when we represent deterministic context-free languages by deterministic pushdown automata and pushdown automata with attached proofs that they accept deterministic context-free languages.

In the second part of this paper we show that our proof techniques furthermore prove that, for example, the relative succinctness results hold for the representation of deterministic context-free languages by deterministic pushdown automata and nondeterministic pushdown automata with attached proofs that they accept deterministic context-free languages.

Finally, to gain further insight how the inclusion of formal proofs or correctness in representations of languages affects their succinctness, we consider the representation of finite sets. We show that there is no recursive bound in the relative succinctness of the representation of finite sets by finite automata or Turing machines (even if we attach proofs that the Tm accepts a finite set). On the other hand, we show that the relative succinctness is recursively bounded for the representation of finite sets by finite automata or Turing machines with proofs which explicitly give the cardinality of the finite set accepted.

It follows from the results that the relative succinctness is not recursively bounded for the representation of finite sets by finite automata (or tables) or Turing machines which accept them, but that there is a recursive bound for the representation of finite sets by finite automata (or tables) and Turing machines which list them and halt.

It is interesting to observe that the succinctness results discussed in this paper do not directly follow from Blum's well known size of machines theorem [1]. This theorem asserts that for any infinite, recursively enumerable set S of Turing machines one can effectively exhibit Turing machines in S which are arbitrarily (by any given recursive function) bigger than other equivalent Turing machines. This is actually not a succinctness result, in the sense used in this paper, since there is no guarantee that the shorter descriptions are not in S itself. One can derive a succinctness result between restricted and unrestricted Turing machine descriptions from Blum's theorem if the minimal machines in S can be recursively enumerated (for example, if the machines in S are total). Even then the results in this paper do not follow from this general theorem because they deal with succinctness between two restricted representations and furthermore, in several cases the class of machines (or grammars) considered in this paper is not recursively enumerable, for example the class of unambiguous context-free grammars.

Succinctness results about cfl's. We first establish notation and summarize some well known facts about context-free languages (cfl's).

We denote pushdown automata (pda) by A_i and deterministic pushdown automata (dpda) by D_j. Let $|A_i|$ denote the length of the description of the automaton A_i over some finite alphabet and $L(A_i)$ the language accepted by A_i. We consider only one-tape Turing machines, denoted by M_i, and for technical reasons we assume (without any loss of generality) that M_i can halt only after an even number of moves, M_i accepts by halting and that it makes at least two moves before halting, finally assume that M_i cannot print a blank. An instantaneous description of M_i depicts the symbols written on the tape, indicates the tape square scanned by M_i and its state; they are strings of the following form:

$$-\Sigma^*(a, q)\Sigma^*-, \qquad -(-, q)\Sigma^*- \quad \text{or} \quad -\Sigma^*(-, q)-,$$

where $-$ denotes a blank tape square, Σ is the finite alphabet of symbols M_i can print,

$a \in \Sigma$ and q is a state of M_i. For Tm M_i $\mathrm{ID}_0(x)$ denotes the instantaneous description of the starting configuration on input x and $\mathrm{ID}_1(x)$, $\mathrm{ID}_2(x), \cdots$ denote the successive instantaneous descriptions of M_i on input x. If $x = a_1 a_2, \cdots, a_n$ then $x^{\mathsf{T}} = a_n a_{n-1}, \cdots, a_2 a_1$. Let VALC $[M_i]$ denote the set of valid computations of M_i in which every second instantaneous description is reversed, i.e.,

$$\mathrm{VALC}\,[M_i] = \{\#\mathrm{ID}_0(x)\#[\mathrm{ID}_1\,(x)]^{\mathsf{T}}\#\mathrm{ID}_2(x)\cdots\#[\mathrm{ID}_{2k-1}(x)]^{\mathsf{T}}\#\mathrm{ID}_{2k}(x)\#\,|\,x \in \Sigma^* \text{ and } \mathrm{ID}_{2k}(x) \text{ is a halting configuration}\}.$$

Let

$$\mathrm{INVALC}\,[M_i] = \Gamma^* - \mathrm{VALC}\,[M_i].$$

It is well known that INVALC $[M_i]$ can be accepted by a nondeterministic pda and therefore it is a cfl [2]. On the other hand, VALC $[M_i]$ is a cfl iff $L(M_i)$ is a finite set, since otherwise for arbitrarily large inputs x the three first instantaneous descriptions must be related and the cfl pumping lemma does not hold. This yields the well known auxiliary result.

LEMMA 1. *INVALC $[M_i]$ is a deterministic cfl iff $L(M_i)$ is finite.*

Proof. If $L(M_i)$ is finite then INVALC $[M_i]$ is a regular set and therefore a dcfl. If $L(M_i)$ is infinite then VALC $[M_i]$ is not a cfl and therefore INVALC $[M_i]$ cannot be a dcfl. □

LEMMA 2. *The set $R = \{A_i \,|\, L(A_i)$ is not a dcfl$\}$ is not recursively enumerable.*

Proof. Since INVALC $[M_i]$ is a deterministic cfl iff $L(M_i)$ is finite, a recursive enumeration of R would yield a recursive enumeration of the set $\{M_i \,|\, L(M_i)$ is infinite$\}$, which is seen not to be possible by Rices's theorem. □

For two representations, such as the representation of deterministic cfl's by deterministic and nondeterministic pda's, we will say that their *relative succinctness is not recursively bounded*, if there does not exist a recursive function F such that for any pda, A, that accepts a deterministic cfl, there exists an equivalent deterministic pda, D, for which $|D| \leqq F(|A|)$.

THEOREM 3. *The relative succinctness of representing deterministic cfl's by deterministic and nondeterministic pda's is not recursively bounded.*

Proof. If such a recursive function F exists then for any pda A we can compute $F(|A|)$ and effectively list the dpda's whose length of description does not exceed $F(|A|)$, say $D_{i_1}, D_{i_2}, \cdots, D_{i_s}$. Then $L(A)$ is a nondeterministic cfl iff none of the D_{i_j}, $1 \leqq j \leqq s$, is equivalent to A, but if this is so then we can detect it by comparing the D_{i_j} and A on successive inputs from Σ^*. Therefore the existence of F implies that the set

$$\{A \,|\, L(A) \text{ is not a dcfl}\}$$

is recursively enumerable, which we know is not the case by Lemma 2. Therefore, F does not exist as was to be shown. □

Next we consider the relative succinctness between the representation of unambiguous cfl's by unambiguous and ambiguous cfg's.

We exploit a recent result, which is given in a somewhat different formulation in [4]. For any Tm, M_i, let

$$A_S(M_i) = \{\#\mathrm{ID}_0\,(x)\#([\mathrm{ID}_j]^T\#\mathrm{ID}_{j+1}\#)^*\,|\, \mathrm{ID}_{j+1} \text{ follows from } \mathrm{ID}_j \text{ by one operation of } M_i,\ x \in \Sigma^*\},$$

$$A_E(M_i) = \{\#(\mathrm{ID}_j\#[\mathrm{ID}_{j+1}]^T\#)^*\mathrm{ID}_{2k}\#\,|\,\mathrm{ID}_{j+1} \text{ follows from } \mathrm{ID}_j \text{ in one operation of } M_i \text{ and } \mathrm{ID}_{2k} \text{ is a halting configuration}\}$$

and define

$$A(M_i) = A_S(M_i) \cup A_E(M_i).$$

It is easily seen that $A(M_i)$ is a context-free language and it links the ambiguity question for $A(M_i)$ to finiteness of sets accepted by the Turing machine M_i.

THEOREM 4. *$A(M_i)$ is an inherently ambiguous cfl iff $L(M_i)$ is infinite.*

Proof. For the proof see [4]. □

THEOREM 5. *The relative succinctness of representing unambiguous cfl's by unambiguous and ambiguous cfg's is not recursively bounded.*

Proof. If a recursive bound F exists, then the set

$$AMB = \{G \mid G \text{ cfg and } L(G) \text{ is inherently ambiguous}\}$$

is recursively enumerable. To see this note that we can list for any cfg G all cfg's whose representations are shorter than $F(|G|)$ and then cross off those grammars which are found to be ambiguous or not equivalent to G as we test them on successive strings from Σ^*. $L(G)$ is inherently ambiguous iff eventually all grammars from the list are crossed off. Thus the set AMB is recursively enumerable and therefore, (by Theorem 4) so is the set

$$\{M_i \mid L(M_i) \text{ is infinite}\},$$

which leads to a contradiction. Therefore the recursive bound F does not exist. □

By the same method we can give an easy proof for the next result [3].

THEOREM 6. *The relative succinctness of the representation of cofinite sets by finite automata and pushdown automata is not recursively bounded. Therefore the relative succinctness of the representation of regular sets by finite automata and pushdown automata is also not recursively bounded.*

Proof. The proof is similar to the proof of Theorem 3, by using the set

$$R = \{A_i \mid L(A_i) \text{ is not cofinite}\}. \qquad \square$$

The same reasoning shows that there is no recursive bound between the size of context-free grammars (which generate cfl's whose complements are also cfl's) and the size of the cfg's generating the complements.

THEOREM 7. *There is no recursive function F such that for any cfg G for which $\Sigma^* - L(G)$ is a cfl, there exists a cfg, G', with $L(G') = \Sigma^* - L(G)$ and $|G'| \leqq F(|G|)$.*

Proof. The proof is similar to the proof of Theorem 3. □

Succinctness results about verified representations. In the representation of deterministic cfl's by deterministic and nondeterministic pda's we can easily verify that a given automaton is indeed deterministic, but for an equivalent nondeterministic pda we have no fixed way of verifying that it will accept a deterministic cfl. This lack of symmetry in our representations suggests that we should consider only representations by nondeterministic pda's with attached proofs that they accept a deterministic language and add the length of the proof to the length of the representation of the pda.

A close inspection of the original proofs [3], [5], [6] reveals that they do not extend to representations with added proofs. On the other hand our proof techniques show that the previous succinctness results can be extended to representations with attached verifications that they accept the desired type of language.

More precisely, let FS be an axiomatizable, sound formal mathematical system which is powerful enough to express and prove elementary facts about Turing machines, context-free languages and pushdown automata. Since FS is axiomatizable we know that we can recursively enumerate the set of provable theorems and soundness assures us that the provable theorems are true. Instead of specifying FS in detail we will describe what must be easily provable in FS.

(a) Let $M_{\sigma(r)}$ be a simply and uniformly constructed Tm which for each input x computes and saves the length of x, $|x| = n$; then enumerates all one-tape Tm's up to length r, i.e., $|M_i| \geqq r$, and simulates in a dove-tail manner the computations of this finite set of machines on blank tape. $M_{\sigma(r)}$ halts (and therefore accepts) iff M_i, $|M_i| \leqq r$, halts after performing n or more steps. From this construction we see that for all r, $r \geqq 1$, $M_{\sigma(r)}$ accepts a finite set. We assume that FS is sufficiently powerful that we can prove in FS that $L(M_{\sigma(r)})$ is finite and that the length of these proofs is recursively bounded in r.

(b) We furthermore assume that there is a simple and uniform construction ρ which yields for each Tm M_i a pda $A_{\rho(i)}$ such that

$$L(A_{\rho(i)}) = \mathrm{INVALC}\,[M_i]$$

and that it cań be proven in FS (by a proof whose length is recursively bounded in i) that:

if $L(M_i)$ if finite then $L(A_{\rho(i)}) = \mathrm{INVALC}\,[M_i]$ is a deterministic cfl.

From these assumptions it follows that we can prove (easily) in FS that:

$$A_{\rho(\sigma(r))} \text{ accepts a deterministic cfl.}$$

It should be observed that in any logic designed to reason about computations we should be able to formulate and prove easily the above result. Furthermore, to any given sound formal system we can add the above assertions as an axiom scheme to obtain the desired FS.

A nondeterministic pda with a proof in FS that it accepts a deterministic cfl is called a *verified* pda or vpda.

THEOREM 8. *The relative succinctness of representing dcfl's by dpda's and vpda's is not recursively bounded.*

Proof. For r, $r \geqq 1$, let $M_{\sigma(r)}$ be a Tm which accepts all inputs up to length N_r, where N_r is the maximal running time before halting achieved by a Tm of size r on blank tape. Let $A_{\rho(i)}$ be a nondeterministic pda which accepts INVALC $[M_i]$. It is assumed that $\sigma(r)$ and $\rho(r)$ are simple enough to compute and that FS is sufficiently rich that there exist short proofs (whose length is recursively bounded in r) that $L[M_{\sigma(r)}]$ is finite and therefore $L[A_{\rho(\sigma(r))}]$ is a deterministic cfl.

If there exists a recursive bound F between $|A_{\rho(\sigma(r))}|$ and the shortest equivalent dpda, then we can list all the dpda's

$$D_{i_1}, D_{i_2}, \cdots, D_{i_s}, \quad \text{such that } |D_{i_j}| \leqq F[|A_{\rho(\sigma(r))}|], \qquad 1 \leqq j \leqq s.$$

From this list of dpda's we can effectively construct a list of dpda's which accept the complements of these languages. From this new list we can effectively select the dpda's which accept finite sets and compute the longest string accepted by these dpda's. Clearly the length of this string is bigger than N_r and therefore N_r is recursively bounded in r, which is a contradiction. □

We get the next result by exploiting the fact that the length of the proof of "$L[A_{\rho(\sigma(r))}]$ is a dcfl" (in FS) is recursively bounded in r.

COROLLARY 9. *The relative succinctness of representing dcfl's by dpda's and verified pda's with attached proofs that they accept dcfl's is not recursively bounded.*

By assuming that we can easily prove in FS relations between $A(M_i)$ and ambiguous cfl's (i.e., Theorem 4) we obtain the next result.

COROLLARY 10. *There is no recursive succinctness bound between the representation of unambiguous cfl's by unambiguous cfg's and cfg's with proofs that they accept unambiguous cfl's.*

Representation of finite sets. The situation changes drastically if we consider representation of finite sets and finite sets of known size.

THEOREM 11. (a) *There is no recursive succinctness bound for the representation of finite sets by finite automata and by Tm's with proofs that they accept finite sets.*

(b) *There is a recursive succinctness bound for the representation of finite sets by finite automata* (*or tables*) *and Tm's with proofs which explicitly give the size of the finite set accepted.*

(c) *There is a recursive bound for the relative succinctness of representing finite sets by finite automata* (*or lists*) *and Tm's with proofs that they print a list and halt.*

Proof (a) Let $M_{\sigma(r)}$ be the Tm constructed for the proof of Theorem 8 and recall that we have assumed that our formal system FS is sufficiently rich to prove, by proofs whose length is recursively bounded in r, that $L(M_{\sigma(r)})$ is finite. Therefore the length of $M_{\sigma(r)}$ plus the length of the proof in FS that $L(M_{\sigma(r)})$ is finite is recursively bounded in r. On the other hand, since $L(M_{\sigma(r)})$ is finite the number of states of any finite automaton accepting $L(M_{\sigma(r)})$ must be no less than the length of the longest string in $L(M_{\sigma(r)})$, which by construction of $M_{\sigma(r)}$ is not recursively bounded in r. Therefore, the relative succinctness of these two representations cannot be recursively bounded.

(b) The relative succinctness bound F can be constructed as follows. For n construct all proofs of "M_i accepts a set of size k", $i, k = 1, 2, \cdots$, such that $|M_i|$ plus the length of the proof is less or equal to n. For the M_i with such proofs let k_n be the cardinality of the largest set accepted and a_n the length of the longest string accepted. Clearly k_n and a_n are effectively computable and

$$F(n) = a_n \cdot k_n + 2$$

is such a recursive bound.

(c) For any n we can effectively list the finite set of Tm's $M_{i_1}, M_{i_2}, \cdots, M_{i_k}$, such that

$$|M_{i_j}| + |\text{proof that } M_{i_j} \text{ prints a list and halts}| \leqq n.$$

Therefore we can run all the Tm's on this list, which are guaranteed to halt because FS is sound, and determine the length of the longest string printed, n_m. Clearly n_m is recursively computable from n, by the above procedure, and, furthermore, the size of the largest minimal finite automaton accepting the sets $L(M_{i_1}), L(M_{i_2}), \cdots, L(M_{i_k})$ is recursively bounded in n_m. Therefore the size of the finite automata representation of these sets is recursively bounded to n and therefore to the size of the Tm representation with proofs. □

REFERENCES

[1] M. BLUM, *On the size of machines*, Information and Control, 11 (1967), pp. 257–265.

[2] J. HARTMANIS, *Context-free languages and Turing machine computations*, Proceedings of Symposia in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society, Providence, RI, 1967, pp. 42–51.

[3] A. R. MEYER AND M. J. FISCHER, *Economy of description by automata, grammars and formal systems*, Conference Record, IEEE 12th Annual Symposium on Switching and Automata Theory (1971), pp. 188–190.

[4] H. Reedy and W. J. Savitch, *The Turing degree of the inherent ambiguity problem for context-free languages*, Theoret. Comput. Sci., 1 (1975), pp. 77–91.

[5] E. H. Schmidt and T. G. Szymanski, *Succinctness of descriptions of unambiguous context-free languages*, this Journal, 6 (1977), pp. 547–553.

[6] L. G. Valiant, *A note on the succinctness of descriptions of deterministic languages*, Information and Control, 32 (1976), pp. 139–145.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0011 $01.00/0

ADDITION CHAIN METHODS FOR THE EVALUATION OF SPECIFIC POLYNOMIALS*

DAVID DOBKIN† AND RICHARD J. LIPTON‡

Abstract. Addition chains are considered for specific polynomials. It is shown that for a wide class of polynomials the evaluation of their first n terms requires at least $n+O(n^{2/3})$ additions. Included in this class are the first n squares, the first n cubes, $\cdots$, the first n kth powers. The results are established by making contact with results in combinatorics.

Key words. polynomial evaluation, addition chains, Zaranckiewicz theorem

1. Introduction. Addition chains have been widely studied as a means of modelling problems of integer evaluation [1], [3], [5], [7], [9], [10]. In previous researches, questions regarding the complexity of evaluating arbitrary integers via addition chains have been considered. These results have been extended to upper and lower bounds on addition chains for arbitrary sequences of integers [1], [7], [10]. In all cases, algorithms have been produced which give upper bounds derived from counting arguments for worst case sequences. We turn in this paper to an extension of this problem posed by Knuth. That is, we consider the problem of finding addition chains for particular sequences of integers. In particular, we shall be interested in chains that are derived as the ranges of particular polynomials evaluated at sets of integer points. For example, we ask for the complexity of an addition chain for evaluating the first n squares or cubes or the first n values of a particular polynomial, rather than an addition chain for evaluating an arbitrary set of integers. In its present formulation, this problem is of significant practical interest, as it provides a means of modeling the problem of evaluating lacunary polynomials. Typical of such polynomials are the theta functions as defined by Jacobi [9] which can be used to describe certain constants associated with elliptic functions and integrals. Three of these functions (evaluated at $z=0$) are given by:

(i) $\theta_2(0,q)=2q^{1/4}[1+q^{1\cdot 2}+q^{2\cdot 3}+\cdots+q^{n(n+1)}+\cdots]$,

(ii) $\theta_3(0,q)=1+2[q+q^{2^2}+q^{3^2}+\cdots+q^{n^2}+\cdots]$,

(iii) $\theta_4(0,q)=1+2[-q+q^{2^2}-q^{3^2}+\cdots+(-1)^n q^{n^2}+\cdots]$

in the range $0<q<1$. It is clear from this example that fast methods of evaluating the addition chains (ii) and (iii) consisting of the first n squares and (i) consisting of the first n integers of the form $p(p+1)$ for p an integer would yield fast methods for evaluating these functions and hence fast methods for determining the necessary constants. This application is a sample of the type of problem which can be best approached by studying addition chains for computing particular sequences of integers. And indeed, the methods described here lead to algorithms which give a factor of 2 speedup over naive methods for evaluating these polynomials.

Our goal in what follows is to find for polynomially generated sequences addition chains which are optimal. This problem is different from previous approaches in that we are concerned with tight bounds for particular problems where previous studies have

* Received by the editors June 6, 1978, and in revised form January 12, 1979.

† Department of Computer Science, Yale University, New Haven, Connecticut. Now at Department of Computer Science, University of Arizona, Tucson, Arizona 85721. The work of this author was supported in part by the National Science Foundation under Grant MCS 76-11460.

‡ Department of Computer Science, Yale University, New Haven, Connecticut. Now at Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. The work of this author was supported in part by the National Science Foundation under Grant MCS 76-81486.

derived bounds by counting arguments and thus made statements of the form:

> There is a hard sequence and any sequence can be computed within the number of steps required by the hard sequence.

Our goal is to make statements of the form:

> Computing the sequence $p_1, \cdots, p_k$ generated as the first n values of the polynomial
>
> $$p(t)=\sum_{i=1}^{\delta} \alpha_i t^i$$
>
> requires at least a certain number of steps.

The impact of such a statement would be a lower bound on algorithms in this class for evaluating a particular polynomial.

Our main results deal with lower bounds and, in particular, we show a lower bound of $n+O(n^{2/3-\varepsilon})$ on the complexity of evaluating the first n terms of any polynomially generated sequence satisfying certain restrictions, which are known to be satisfied by an infinite set of polynomials.

2. Upper bounds. We begin with a presentation of our model. We shall use the standard notation as in [3]. That is, an addition chain is a sequence $a_0 = 1, a_1, a_2, \cdots, a_k$ such that for each i there exist $p, q < i$ with $a_i = a_p + a_q$. We shall represent such an addition chain as $\{a_i\}_{i=1}^{k}$ and define its length as k. An addition chain $\{a_i\}$ is said to realise the sequence $\{s_1, \cdots, s_l\}$ if there exist $i_1, i_2, \cdots, i_l$ with $1 \leqq i_1 < \cdots < i_l \leqq k$ such that $a_{i_j} = s_j$ for $j = 2, 3, \cdots, l$. Furthermore, we denote the length of the shortest addition chain for $\{s_1, \cdots, s_l\}$ as $C\{s_1, \cdots, s_l\}$.

We now turn to an application of these concepts to the problem of polynomial evaluation. Suppose that $f(t)$ is a quadratic polynomial with integer coefficients. We then say that (p, q, r) is a hypotenuse triple for f if $f(p) = f(q) + f(r)$ and denote the set of all such triples as $H(f(t))$. If we are evaluating a polynomial f at a set of integer points and (p, q, r) is a hypotenuse triple for $f(q < p, r < p)$, then $f(p)$ can easily be evaluated, given that $f(1), \cdots, f(p-1)$ have been found, in one step. Thus, we might expect the complexity of a sequence to be related to its density of hypotenuse triples. Indeed, this can be shown to be the case as follows:

DEFINITION. Let $A(f(t)) = \{p \mid q, r \text{ with } (p, q, r) \in H(f(t))\}$ be the set of hypotenuses for f.

We may now use this definition to describe our main results. We begin with a lemma relating densities of hypotenuse-sets to sequence complexity. For what follows, we shall assume that f is to be evaluated at the points $1, 2, \cdots, n$ and shall denote the complexity of this evaluation as $C_f(n)$. For notational convenience, we also define

$$A_n(f(t)) = \{p \in A(f(t)) \mid p < n\}.$$

When no confusion results, we will also use $A(f(t))$ and $A_n(f(t))$ to represent the cardinality of these sets. We may next establish an upper bound theorem.

THEOREM. *The complexity of evaluating t^2 at the first n integers grows as $n + o(n)$.*

Proof. The theorem is proved by first showing that

$$\lim_{n\to\infty} \frac{A_n(t^2)}{n} = 1.$$

This limit is derived by generating from each hypotenuse number an infinite set of hypotenuses. Every prime of the form $4k+1$ is a hypotenuse (see [9]) and further

$$A(t^2)=\{p\,|\,\text{a prime of the form } 4k+1 \text{ such that } (4k+1)\,|\,p\}.$$

Next we observe that the number of nonhypotenuses less than n is at most n times

$$\prod_{\substack{p\leqq n\\ p\equiv 1 \bmod 4\\ p \text{ prime}}}\left(1-\frac{1}{p}\right).$$

Uchiyama [10] shows that this product grows as $C(\log n)^{-1/2}+O((\log n)^{-3/2})$ so that the number of nonhypotenuse numbers less than n grows as $O(n\backslash\sqrt{\log n})$. We may now derive an algorithm for realizing the first n squares in $n+O(n\backslash\sqrt{\log n})$ additions. To begin, let $p(n)$ be the product of the first $a(n)$ primes of the form $4t+1$ and let $b_1,\cdots,b_{g(n)}$ be the nonhypotenuse numbers less than $p(n)$. We compute $e_{i0}=b_{i+1}^2-b_i^2$ and $\delta_i=2p(n)(b_{i+1}-b_i)$ for $i=0,\cdots,g(n)-1$ and $b_0=0$. This requires less than $O(p(n))$ additions. Next, we observe that each nonhypotenuse number is of the form $kp(n)+b_i$ and may be computed as $kp(n)+b_{i-1}+e_{ik}$ where e_{ik} is computed as $e_{i,k-1}+\delta_i$, resulting in $p(n)=O(n\backslash\sqrt{\log n})$. As we were preparing this paper, we learned of results due to Newman [8] who improved this result to $n+O(n \exp(-C \log n/\log\log n))$.

It is interesting to remark in passing that Fermat's last theorem suggests that no result of the type mentioned here will be possible for cubics or higher powers, since most likely $A_n(t^k)=0$ for all n and $k>2$.

3. Lower bounds. In the last section we demonstrated that there is an addition chain for the first n squares that uses at most $n+o(n/\sqrt{\log n})$ additions. A natural question is just how tight is this bound. Clearly a trivial argument yields a lower bound of n; a further reflection yields a lower bound of $n+t(n)$ where $t(n)$ tends to infinity. However, we can obtain an improved lower bound.

THEOREM 1. *Any addition chain for the first n squares requires at least* $n+n^a$ *steps where a is any constant less than* $2/3$ *and n is large enough.*

Thus at least n^a "extra" additions are needed by any chains for the first n squares.

Proof. Let $S_1,\cdots,S_m$ be an addition chain for the first n squares. Say S_i is an auxiliary number provided it is not a square, and let Ω be the set of such numbers. Now each step in this addition chain that computes a square, say p^2, is in one of the following forms:

(I) $p^2=q^2+r^2$;
(II) $p^2=q^2+a$ where $a\in\Omega$;
(III) $p^2=a_1+a_2$ where $a_1,a_2\in\Omega$.

By the fact that hypotenuse numbers are sufficiently sparse we can easily show that the number of steps of the forms (II) and (III) are at least $cn/\sqrt{\log n}$ for some $c>0$. We will now argue that this is possible only when $|\Omega|$ is large.

Let us first bound the number of steps of type (II). For each $a\in\Omega$, let Π_a be the set of (p,q) such that $p^2=q^2+a$. We will then show that $|\Pi_a|$ is $O(n^\varepsilon)$ and so it will follow that the number of (II) steps is at most $O(n^\varepsilon|\Omega|)$ which is what we wish to show. Consider therefore the equation $p^2-q^2=a$. This implies that $p+q$ and $p-q$ are both divisors of a. But, as is well known, a can have at most $O(n^{2\delta})$ divisors where δ is any number with $\delta>0$. Thus, $p+q=dl$ and $p-q=d2$ where $d1$ and $d2$ lie in a set of size at most $O(n^{2\delta})$. Clearly, $p=(d1+d2)/2$ and $q=(d1-d2)/2$ and so there are at most $O(n^{4\delta})$ choices for p and q; hence, it follows that $|\Pi_a|$ is less than $O(n^{4\delta})$ for each $a\in\Omega$.

We now turn to consider steps of the form (III). The key to our argument is the theorem of Zarankiewicz (Erdos and Spencer [2]). Suppose that there are t steps of form (III). As before, Ω is the set of auxiliary numbers. Now consider the 0-1 matrix $M=(m_{ij})$ such that

$$M_{ij}=1 \text{ iff } a_i+a_j \text{ is a perfect square less than or equal to } n^2.$$

Then M is a k by 0-1 matrix where $k=|\Omega|$, and further, M has at least t 1's. Thus by Zarankiewicz's theorem there is a 2 by l submatrix of all 1's with $l=f(k,t)$ where

$$f(k,t)=\max\left\{l \,|\, (t/k^2)^{2l}\binom{k}{l}\geqq 1\right\}.$$

Thus

$$p_1^2=a_{i_1}+a_{j_1}\cdots p_l^2=a_{i_1}+a_{j_l},$$
$$q_1^2=a_{i_2}+a_{j_1}\cdots q_l^2=a_{i_2}+a_{j_l},$$

say. Therefore, the difference $a_{i_1}-a_{i_2}$ occurs at least l times as a difference of two squares. But as in Lipton [4] we can show that this implies that $l<n^\delta$ for any δ and n large enough. Let us therefore assume that $l<n^\delta$ for some small $\delta>0$; hence, we can assume that $f(k,t)<n^\delta$. We now enter the "elementary calculation mode" i.e., we plan to use this bound to show that if t is large then so must k be. To see this, let us make the following assumptions:

$$l=f(k,t)<n^\delta,$$
$$k<n^\lambda \quad \text{where } \lambda<2/3,$$
$$t>n^\eta \quad \text{where } \eta>1-o(1).$$

The reasons for the first assumption have already been discussed; the second can be assumed or else the theorem is immediately true (recall the k measures the number of extra steps in the addition chain); the last assumption follows since otherwise there will not be enought steps of types (II) and (III). As in [2] we can show that M has a 2 by l submatrix of all 1's provided

$$(t/k^2)^{2l}\binom{k}{l}\geqq 1.$$

Now this must be the case and so the theorem is proved.

The proof given above can actually be generalized to prove lower bounds on addition chains for other sequences as well. Let p be a polynomial and consider the addition chains for generating the values of p at the first n integers. We observe that only two facts about the squares were used in the proof given above, first that there were suitably many squares which could not be given as sums of pairs of previous squares and second that no integer is the difference of arbitrarily many pairs of squares. We used these facts together with Zarankiewicz's theorem to prove the lower bound. This method can be generalized. We define two properties that a sequence of values of a polynomial need to have in order to give a lower bound as above. Throughout our discussion, we assume that p is monotone.

Property P1. There exists a constant $c>0$ such that for any $\varepsilon>0$ and n sufficiently large more than $cn^{1-\varepsilon}$ values of $p(i), i<n$, cannot be expressed as $p(i)=p(j)+p(k), j, k<i$.

Property P2. There exists a constant such that for n sufficiently large, for each $\varepsilon>0$, there exists at most cn^ε pairs (i,j) such that $p(i)-p(j)=n$.

Based on these facts, we can then prove the following

THEOREM 2. *If p satisfies Properties* P1 *and* P2, *then any addition chain for computing values of p at the first n integers requires at least $n+n^\alpha$ steps where $\alpha<2/3$.*

We observe that as corollaries of this theorem we have Theorem 1 as well as the following

COROLLARY. *Any addition chain for computing* $1^k, 2^k, \cdots, n^k$ *requires at least* $n+n^\alpha$ *steps where* α *is any constant less than* $2/3$, *and* k *is any integer* >2.

Proof. We observe that such a sequence satisfies Property P1 by a result of Mumford [6] and Property P2 follows as above by a factoring argument.

We leave open the problem of computing addition chains for values of arbitrary polynomials, for which we conjecture Theorem 2 holds.

Acknowledgment. We would like to thank Dr. Nicholas Pippenger for leading us to references [6], [8], and [10].

Note added in proof. Since this manuscript was written, we have been able to extend the results given here to addition/multiplication chains.

REFERENCES

[1] P. ERDOS, *Remarks on number theory III: On addition chains*, Acta Arith., 6 (1960), pp. 77–81.

[2] P. ERDOS AND J. SPENCER, *Probabilistic methods in combinatorics*, Academic Press, New York, 1974.

[3] D. E. KNUTH, *The art of computer programming, Vol. 2; Seminumerical algorithms*, Addison-Wesley, Reading, MA, 1969.

[4] R. J. LIPTON, *Specific hard* 0, 1 *polynomials over a monotone basis*, Yale Res. Rep., Yale University, New Haven, CT., 1977.

[5] R. J. LIPTON AND D. DOBKIN, *Complexity measures and hierarchies for the evaluation of integers, polynomials, and n-linear forms*, Seventh Proceeding of ACM Symposium on Theory of Computing, 1975, pp. 1–5.

[6] D. MUMFORD, *A remark on Mordell's conjecture*, Amer. J. Math., 87 (1965), pp. 1007–1016.

[7] N. PIPPENGER, *On the evaluation of powers and related problems*, Proc. IEEE 17th Annual Symposium on Foundations of Computer Science, pp. 258–263.

[8] ———, Private communication, April 1, 1977.

[9] T. H. SOUTHARD, *Addition chain for the first N squares*, Tech. Rep. CNA-84, Univ. of Texas at Austin, 1974.

[10] S. UCHIYAMA, *On some products in involving primes*, Proc. Amer. Math. Soc., 28 (1971), pp. 629–630.

[11] A. C. YAO, *On the evaluation of powers*, this Journal, 5 (1976), pp. 100–103.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0012 $01.00/0

ON THE COMPLEXITY OF SEARCHING A SET OF VECTORS*

D. S. HIRSCHBERG†

Abstract. The vector searching problem is, given k-vector A (a k-vector is a vector that has k components, over the integers) and given a set $\bar{B}$ of n distinct k-vectors, to determine whether or not A is a member of set $\bar{B}$. Comparisons between components yielding "greater than-equal-less than" results are permitted. If the vectors in $\bar{B}$ are unordered then nk comparisons are necessary and sufficient. In the case when the vectors in $\bar{B}$ are ordered, it is shown that $\lfloor \log n \rfloor + k$ comparisons are necessary and, for $n \geqq 4k$, $k\lceil \log (n/k)\rceil + 2k - 1$ comparisons are sufficient.

Key words. searching, vector, lower bounds, complexity

Searching an unordered set. We first consider the case in which the vectors in $\bar{B}$ are *not* ordered. In this case, an upper bound of nk comparisons can be easily demonstrated. A simple adversary can be constructed to show that nk comparisons are also necessary.

A nontrivial dynamic adversary can be used to construct an oracle to demonstrate that nk comparisons are necessary even if we allow comparisons between elements of the vectors in $\bar{B}$ as well as comparisons between elements of A and vectors in $\bar{B}$ [3].

Recently, Stockmeyer and Wong have demonstrated upper and lower bounds that are within a small factor from one another for the more general problem of determining the intersection of two sets of vectors [7].

For a review of the use of oracles to derive lower bounds, the reader is referred to [1], [4], [6].

Searching an ordered set. We now consider the case in which preprocessing of the set $\bar{B}$ is permitted. That is, we can assume that $\bar{B}$ is in some prearranged order, such as lexicographic order.

In the discussions that follow, all logarithms are assumed to be base 2.

A lower bound of $\lfloor \log n \rfloor + k$ comparisons can be seen by observing that $\lfloor \log n \rfloor + 1$ comparisons are required to determine if there is any vector having the correct value of one component, and $k-1$ comparisons are required to verify the agreement of the remaining components.

The oracle for distinguishing a path (which will be of length at least $\lfloor \log n \rfloor + k$) in each decision tree that solves this problem is as follows.

Initially, define *low* = 1 and *high* = n.

Let the next comparison presented to the oracle be $a_j : b_{ij}$.

```
    mid ← (low + high)/2
    If low ≠ high then:
        if $i <$ low then return>
        if low $\leqq i \leqq$ mid then [low ← $i+1$; return>]
        if mid $< i \leqq$ high then [high ← $i-1$; return<]
        if high $< i$ then return<
    Else (low = high) return=
```

low will not equal *high* until after at least $\lfloor \log n \rfloor$ comparisons. During these comparisons, vector A could equal any of the vectors $B_{\text{low}} \cdots B_{\text{high}}$.

When low = high, until *all* components of B_{low} have been compared with A, B_{low} may equal A but it is also possible that one component of B_{low} will be less than the

* Received by the editors January 12, 1978, and in revised form March 30, 1979. This research was supported by The National Science Foundation under Grant MCS-76-3933.

† Department of Electrical Engineering, Rice University, Houston, Texas 77001.

corresponding component of A and, in that case, it is possible that none of the vectors in $\bar{B}$ are equal to A.

Thus $\lfloor \log n \rfloor + k$ comparisons are necessary to solve this problem.

In the above analysis, we assumed that all comparisons are between a component of A and the corresponding component of a vector in $\bar{B}$. It is straightforward to generalize and allow comparisons between components of vectors both of which are elements of $\bar{B}$.

Having demonstrated a lower bound, we now consider upper bounds for this problem.

We present and analyze two algorithms that solve the ordered set problem and then combine them to obtain an algorithm that is faster than both.

The first algorithm is an example of *binary search.* Let $\bar{B} = \{B_1, B_2, \cdots, B_n\}$ and let $B_i = b_{i1} \cdots b_{ik}$. Proceed comparing the components of A with those of the central vector, i.e. compare a_h with b_{jh} for $h = 1, 2, \cdots$ where $j = \lfloor (n+1)/2 \rfloor$. If all comparisons result in "equal" then $A = B_j$. Otherwise, if at some point we get a "less than" result then $A \neq B_j$ and we can restrict our attention to $\bar{B}' = \{B_1, \cdots, B_{j-1}\}$. Similarly, if we get a "greater than" result, then we can restrict our attention to $\bar{B}' = \{B_{j+1}, \cdots, B_n\}$. In the worst case, we will require k comparisons in each of $1 + \lfloor \log n \rfloor$ iterations for a total of $k + k \lfloor \log n \rfloor$ comparisons. This is equivalent to the result in [2].

The second algorithm uses *linear search* and is as follows:

```
        $j \leftarrow 1$
        $h \leftarrow 1$
        **while** $j \leqq n$ AND $h \leqq k$
        **do compare** $a_h : b_{jh}$
            **if** = **then** $h \leftarrow h + 1$
            **else if** > **then** $j \leftarrow j + 1$
            **else [print** 'NO SOLUTION'; **stop]**
        **od**
        **if** $j > n$ **then [print** 'NO SOLUTION'; **stop]**
final:  **if** $a_i = b_{ji}$ for all $i \in \{1, 2, \cdots, k-1\}$
        **then print** $j$; **comment** $A = B_j$
        **else print** 'NO SOLUTION'
        **stop**
```

The algorithm finds the first (lowest indexed) vector, B_j, that matches A in the hth component. All lower indexed vectors are not considered further. All other vectors are *assumed* to match in this component. The algorithm then iterates on the $(h+1)$st component. This part of the algorithm will make at most $n + k - 1$ comparisons (each iteration increments either j with upper limit n, or h with upper limit k). If we succeed in matching all k components in this manner then the vector, B_j, that is found will be equal to A if all assumptions made earlier apply to B_j. However, if B_j disagrees with A in any component h' then A does not appear in $\bar{B}$ since all $j' < j$ have been eliminated, $B_j \neq A$ (assumed here) and for all $j'' > j$, $B_{j''}$ will disagree with A among the first h' components since $\bar{B}$ is in lexicographic order. The final phase of the algorithm, in which the components of B_j (which were assumed to agree with A) are compared with A, requires at most $k - 1$ comparisons for a total of at most $n + 2k - 2$ comparisons.

We note that the linear search algorithm's mirror image also works. That is, we can start with $j = n$ and decrement j, being careful to interchange the $<$'s and $>$'s. We can, as an initial improvement, compare A with the central vector in $\bar{B}$ rather than with B_1 or B_n and, at the first "less than" or "greater than" result, continue with the linear search algorithm applied to only half of the original set $\bar{B}$. This leads to an algorithm that requires, in the worst case, only $\lfloor n/2 \rfloor + 2k - 1$ comparisons. We call this improved algorithm the *modified linear search* algorithm.

We can make further improvements by deciding, at the time that a "less than" or "greater than" result is obtained, whether to continue in the style of the binary or the modified linear search algorithm depending upon which will lead to fewer comparisons in the worst case. If, after making h comparisons (resulting in "equal") along vector B_j within feasible set $\bar{B}$ of cardinality n, we make a comparison resulting in "less than" or "greater than" then continuing with the linear search algorithm requires, in the worst case, at most $\lfloor n/2 \rfloor + 2k - 1 - h$ additional comparisons. If, however, we decide to proceed with the comparisons in a new vector within $\bar{B}$ and thus follow the binary search algorithm or follow the modified linear search algorithm on a feasible set of half the size, then we will have upper bounds of $k \lfloor \log n \rfloor$ and $\lfloor n/4 \rfloor + 2k - 1$ additional comparisons respectively. We should continue with linear search only if

$$\lfloor n/2 \rfloor + 2k - 1 - h < \min \{ \lfloor n/4 \rfloor + 2k - 1, k \lfloor \log n \rfloor \}$$

which holds only if $n < 4h$. For particular values of k, we can solve this inequality to gain further restrictions. For example, if $k = 3$ then we should continue with linear search only if $n = 4$ and $h = 2$ or $n = 5$ and $h = 2$.

Let $T(n, k)$ be the minimum number of comparisons required for the ordered vector search problem when $\bar{B}$ consists of n k-vectors. Then, for $n \leqq 4k$, $T(n, k) \leqq \lfloor n/2 \rfloor + 2k - 1$.

For $n = k2^r$, $T(n, k) \leqq k + T(n/2, k) \leqq (r - 2)k + T(4k, k) \leqq (r + 2)k - 1$.

For $k2^{r-1} < n < k2^r$, $T(n, k) \leqq (r - 2)k + T(4k - 1, k) \leqq (r + 2)k - 2$. Note that in both cases, $r = \lceil \log (n/k) \rceil$.

Algorithm VECTOR_SEARCH incorporates the modifications mentioned above.

```
         VECTOR_SEARCH ($A$, $\bar{B}$, $n$, $k$)
         low ← 1
         high ← $n$
binary:  binsearch ← TRUE
         while binsearch AND low ≦ high
         do $j$ ← (low + high)/2
             compare $a_1 : b_{j1}$
             if > then high ← $j - 1$
             else if < then low ← $j + 1$
             else binsearch ← FALSE
         od
         if low > high then [print 'NO SOLUTION'; stop]
         $n$ ← high − low + 1
modlin:  $h$ ← 2
         while $h \leqq k$
         do compare $a_h : b_{jh}$
             if = then $h$ ← $h + 1$
             else if > then if $n \geqq 4*h$
                                then [high ← $j - 1$; goto binary]
                                else goto linear
             else if < then if $n \geqq 4*h$
                                then [low ← $j + 1$; goto binary]
                                else goto linear2
```

```
          od
          print $j$; comment $A = B_j$
          stop
linear:   while $j \leqq$ high AND $h \leqq k$
          do compare $a_h : b_{jh}$
               if = then $h \leftarrow h + 1$
               else if > then $j \leftarrow j + 1$
               else [print 'NO SOLUTION'; stop]
          od
          if $j >$ high then [print 'NO SOLUTION'; stop]
final:    if $a_i = b_{ji}$ for all $i \in \{1, 2, \cdots, k-1\}$
          then print $j$; comment $A = B_j$
          else print 'NO SOLUTION'
          stop
linear 2: while $j \geqq$ low AND $h \leqq k$
          do compare $a_h : b_{jh}$
               if = then $h \leftarrow h + 1$
               else if < then $j \leftarrow j - 1$
               else [print 'NO SOLUTION'; stop]
          od
          if $j <$ low then [print 'NO SOLUTION'; stop]
          goto final
```

REFERENCES

[1] A. V. AHO, D. S. HIRSCHBERG AND J. D. ULLMAN, *Bounds on the complexity of the longest common subsequence problem*, J. Assoc. Computer Mach., 23 (1976), pp. 1–12.

[2] D. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, this Journal, 5 (1976), pp. 181–186.

[3] D. S. HIRSCHBERG, *On the complexity of vector searching*, Rice Univ. Tech. Rept. # 7807, Houston, TX, June 1978.

[4] D. E. KNUTH, *The Art of Computer Programming*, vol. 3. Addison-Wesley, New York, 1973.

[5] V. V. RAGHAVAN AND C. T. YU, *A note on a multidimensional searching problem*, IPL 6 (1977), pp. 133–135.

[6] E. M. REINGOLD, *On the optimality of some set algorithms*, J. Assoc. Comput. Mach., 19 (1972), pp. 649–659.

[7] L. J. STOCKMEYER AND C. K. WONG, *On the number of comparisons to find the intersection of two relations*, IBM Watson Research Center Tech. Rept., 1978.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-013 $01.00/0

COMBINATORIAL GRAY CODES*

J. T. JOICHI,† DENNIS E. WHITE‡ AND S. G. WILLIAMSON¶

Abstract. We consider families $\{\mathbf{C}(n, k) : O \leqq k \leqq n\}$ where each $\mathbf{C}(n, k)$ is a set of combinatorial objects, $C(n, k) = |\mathbf{C}(n, k)|$ satisfies a recursion $C(n, k) = a_{n,k}C(n-1, k-1) + b_{n,k}C(n-1, k)$, and each object in $\mathbf{C}(n, k)$ is represented by an n-vector. We study "loop-free" or "uniformly bounded transition" algorithms, i.e., algorithms which yield linear orders on the sets $\mathbf{C}(n, k)$ so that the vectors representing consecutive objects are "close to each other" (combinatorial Gray codes).

Key words. listing algorithms, uniformly bounded operations, uniformly bounded transition algorithms, loop-free algorithms, binary reflected Gray codes, combinatorial Gray codes, binomial grids

Introduction. We consider families $\{C_\lambda : \lambda \in \Lambda\}$ where each C_λ is a set of combinatorial objects and Λ is some countable index set. In all of our examples, Λ is either $\mathbf{N}_0 = \{0, 1, 2, \cdots\}$ or $\mathbf{T} = \{(n, k) \in \mathbf{N}_0 \times \mathbf{N}_0 : k \leqq n\}$. For example, the combinatorial objects may be permutations of a set of n objects, subsets of an n-set, partitions of a set of n objects into k blocks, etc.

Suppose $\Lambda = \mathbf{N}_0$ or $\mathbf{T}$ and $\lambda = n$ or (n, k). We then specify certain basic representations of the objects in C_λ as strings of symbols $\alpha_1\alpha_2 \cdots \alpha_m$ where $m = O(n)$. For example, a partition of $\{1, 2, \cdots 5\}$ might be written $\{1, 3, 5\}, \{2, 4\}$ or 12121. The latter representation, of the form $\alpha_1\alpha_2 \cdots \alpha_5$, specifies that symbol i be placed in block α_i. A permutation might be written 21354, or in cycle form (12)(3)(45), or as 21121. The latter string, of the form $\alpha_1\alpha_2 \cdots \alpha_5$, specifies that the permutation sends i to p_i where p_i is the α_ith symbol in the list 1, 2, 3, 4, 5 with $p_1, p_2, \cdots, p_{i-1}$ removed. Systematic methods for constructing such representations of the basic combinatorial objects are discussed in Wilf [5] and Williamson [6].

In many theoretical situations in classical enumerative combinatorics one considers linear orders on these sets C_λ. In any computational problem involving the objects in C_λ, a listing algorithm for the strings representing the objects is ipso facto a linear order on the sets. In this paper we study algorithms which yield linear orders with the property that the strings representing consecutive objects in C_λ are, roughly speaking, close to each other. As for the representation itself, we make no further judgments on its efficacy. However, in § 5 we make some remarks concerning representations.

An operation requiring a sequence of steps which manipulate strings of symbols parameterized by λ is said to be *uniformly bounded* (UB) if the number of steps required is bounded by M, where M is a constant independent of λ. For example, if $\lambda = n$, the statements "$p \leftarrow x_{n+1}$, $v_p \leftarrow 0$, $x_{n+1} \leftarrow y_{n+1}$" form a UB operation while the statement "find $p \in \{1, 2, \cdots, n\}$ such that $v_p = 1$", in general, does not.

Suppose $\mathscr{A}$ is an algorithm which generates a list of strings of length $O(n)$ representing the combinatorial objects in C_λ where $\lambda = (n, k)$ or $\lambda = n$. Suppose further that $\mathscr{A}$ requires certain auxiliary data. We say that $\mathscr{A}$ is a *uniformly bounded transition*

* Received by the editors February 16, 1977, and in revised from July 15, 1978.

† School of Mathematics, University of Minnesota, Minneapolis, Minnesota 55455.

‡ School of Mathematics, University of Minnesota, Minneapolis, Minnesota 55455. The work of this author was supported in part by the National Science Foundation under Contracts GP-43010 and MCS 76-04839.

¶ Department of Mathematics, University of California, San Diego, La Jolla, California 92093. The work of this author was supported in part by the National Science Foundation under Contract MCS 74-02714-A02.

(UBT) *algorithm* iff (1) the auxiliary data uses no more than $O(n^2)$ space, (2) the hth string in the list and its corresponding auxiliary data is obtained by $\mathscr{A}$ from the $(h-1)$th string and its auxiliary data by a UB operation, and (3) the completeness of the list is determined by $\mathscr{A}$ in a UB manner. Ehrlich [2] calls such algorithms "loop-free". If the auxiliary data uses no more than $O(n)$ space, we say that $\mathscr{A}$ is a *linear* UBT algorithm.

Standard binary reflected Gray codes can be thought of as linear orders on the set of subsets of a set such that adjacent subsets in the list are "close". See [1], [2] for more detailed discussions of Gray codes. We call a linear order on the strings of symbols representing the combinatorial objects in C_λ a *(linear) combinatorial Gray code* iff there is a (linear) UBT algorithm which generates the list.

In many cases where $\Lambda = \mathbf{T}$, the cardinalities of the sets C_λ satisfy simple two-term recursions. These recursions are usually based upon some natural combinatorial argument. In §§ 3 and 4 we present two algorithms which yield combinatorial Gray codes for many such sets. In § 1 we describe the general recursion and give a number of examples. In § 2 we develop two basic algorithms which are used in the algorithms appearing in §§ 3 and 4. We omit the technical details of the UBT versions of all these algorithms. Finally, we make some concluding remarks in § 5.

In what follows we shall use $[n]$ to represent $\{1, 2, \cdots, n\}$ and (n) to mean $\{0, 1, 2, \cdots, n-1\}$.

1. Binomial recursions and grids. Let $\mathbf{C}(n, k)$ be a doubly-indexed set of combinatorial objects with $n \geqq 0$ and $0 \leqq k \leqq n$. Let $C(n, k) = |\mathbf{C}(n, k)|$ and suppose $C(n, k)$ satisfies the recursion

$$C(n, k) = a_{n,k}C(n-1, k-1) + b_{n,k}C(n-1, k)$$

where $a_{n,k}$ and $b_{n,k}$ are nonnegative integers. Following Wilf [5], we may construct an infinite multigraph on the set of points

$$V = \{(i, j) : i, j \text{ integers}, 0 \leqq j \leqq i\}$$

as follows: construct $b_{i,j}$ edges between the points $(i-l, j)$ and (i, j) and construct $a_{i,j}$ edges between the points $(i-1, j-1)$ and (i, j) (Fig. 1.1).

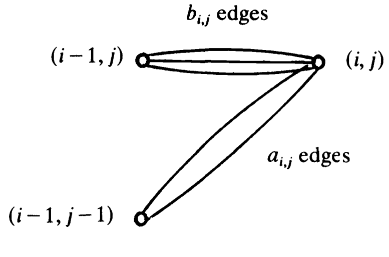


FIG. 1.1

Let H_{C} be the graph thus constructed. Let $H_{\mathrm{C}}(n, k)$ be the directed subgraph induced by the set of points $\{(i, j) \in V : 0 \leqq j \leqq k, j \leqq i \leqq n-k+j\}$, where the horizontal edges are directed to the right and the diagonal edges are directed to the upper-right. A path in $H_{\mathrm{C}}(n, k)$ will mean a directed path. Adopting terminology similar to Wilf [5], we call such a directed graph a *binomial grid*. Further discussion of binomial grids and their relationship to basic linear orders on sets of combinatorial objects may be found in Williamson [6].

By an *(n, k)-path* in $H_C(n, k)$, we shall mean a path from $(0, 0)$ to (n, k). There is an obvious one-to-one correspondence between the set $\mathbf{C}(n, k)$ and the set of all (n, k)-paths in $H_C(n, k)$.

Suppose the edges of $H_C(n, k)$ are labeled. Then corresponding to each (n, k)-path is a unique n-tuple of labels, say $(v_1, \cdots, v_n)$, where v_1 is the outgoing edge from $(0, 0)$ and the remaining edges follow in order. If the labels are carefully chosen, the n-tuple will uniquely determine the path (and hence, the combinatorial object). Such a labeling is described in § 4.

In §§ 3 and 4 we shall also have occasion to consider *transposed* graphs, that is, graphs which are "flipped over" so that the horizontal and diagonal edges are interchanged. More precisely, by the *transpose* of the infinite graph H_C, we mean the graph H_C^t defined on the set of points V with $a_{i,i-j}$ edges between the points $(i-1, j)$ and (i, j) and $b_{i,i-j}$ edges between the points $(i-1, j-1)$ and (i, j). Then $H_C^t(n, n-k)$, the directed subgraph of H_C^t induced by the set of points $\{(i, j) \in V : j \leqq n-k, i \leqq k+j\}$ is merely $H_C(n, k)$ "flipped over" and we call it the *transpose* of $H_C(n, k)$.

Let $G(n, k)$ denote the directed graph corresponding to the set of k-subsets of $[n]$ and the binomial recursion

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

If we label each diagonal edge 1 and each horizontal edge 0, then we have a one-to-one correspondence between the set of (n, k)-paths in $G(n, k)$ and the set of n-tuples of 0's and 1's containing exactly k 1's. In § 3, this labeling of the edges of $G(n, k)$ in conjunction with a labeling of the edges of $H_C(n, k)$ will provide us with a description of the (n, k)-paths in $H_C(n, k)$.

Finally, when we speak of the $a_{i,j}$ and $b_{i,j}$ in a grid $H_C(n, k)$, we mean the $a_{i,j}$ for $1 \leqq j \leqq k$ and $j \leqq i \leqq n-k+j$, and the $b_{i,j}$ for $0 \leqq j \leqq k$ and $j+1 \leqq i \leqq n-k+j$.

We conclude this section with some examples of binomial grids.

Example 1.1 (See Fig. 1.2)

$\mathbf{C}(n, k) =$ set of k-subsets of $[n]$,

$C(n, k) = \binom{n}{k}$,

$C(n, k) = C(n-1, k-1) + C(n-1, k) \qquad (a_{i,j} = b_{i,j} = 1).$

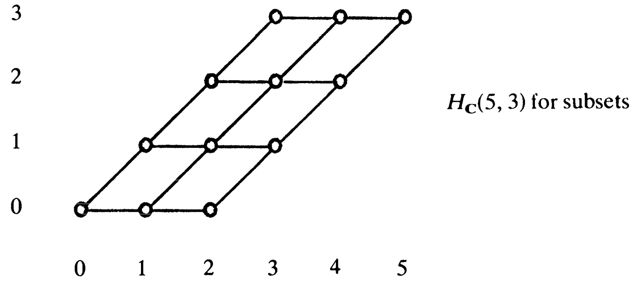


FIG. 1.2

Example 1.2 (see Fig. 1.3).

$\mathbf{C}(n, k) =$ set of partitions of $[n]$ into k blocks,

$C(n, k) = S(n, k)$ (Stirling number of the second kind),

$C(n, k) = C(n-1, k-1) + kC(n-1, k) \qquad (a_{i,j} = 1, b_{i,j} = j).$

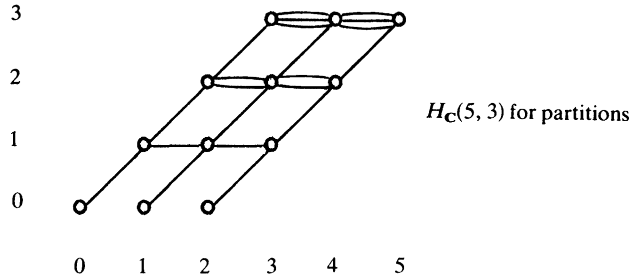


FIG. 1.3

Example 1.3.

$\mathbf{C}(n, k) =$ set of permutations of $[n]$ with k cycles,

$C(n, k) = |s(n, k)|$ ($s(n, k) =$ Stirling number of first kind),

$C(n, k) = C(n-1, k-1) + (n-1)C(n-1, k) \qquad (a_{i,j} = 1, b_{i,j} = i-1).$

Example 1.4 (see Fig. 1.4).

$\mathbf{C}(n, k) =$ set of permutations of $[n]$ with k runs (subsequences $p_i < p_{i+1} < p_{i+2} < \cdots < p_j$ with $p_i < p_{i-1}$ and $p_j > p_{j+1}$),

$C(n, k) =$ Eulerian number,

$C(n, k) = (n-k+1)C(n-1, k-1) + kC(n-1, k) \qquad (a_{i,j} = i-j+1, b_{i,j} = j).$

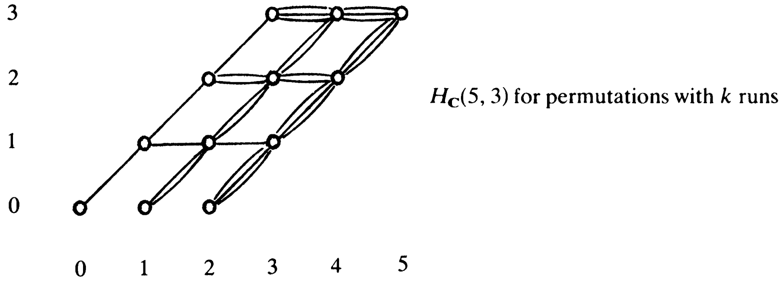


FIG. 1.4

Example 1.5.

$\mathbf{C}(n, k) =$ set of permutations of $[n]$, k at a time,

$C(n, k) = (n)_k = n(n-1) \cdots (n-k+1),$

$C(n, k) = kC(n-1, k-1) + C(n-1, k) \qquad (a_{i,j} = j, b_{i,j} = 1).$

Example 1.6.

$\mathbf{C}(n, k) =$ set of k-dimensional subspaces of an n-dimensional vector space over a field with q elements,

$$C(n, k) = \begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{(q^n - 1)(q^{n-1} - 1) \cdots (q^{n-k+1} - 1)}{(q^k - 1)(q^{k-1} - 1) \cdots (q - 1)},$$

$C(n, k) = C(n-1, k-1) + q^k C(n-1, k) \qquad (a_{i,j} = 1, b_{i,j} = q^j).$

2. Two basic algorithms. In this section we present two algorithms which are not only of interest in themselves, but are also basic to the construction of the general algorithm in § 3. The first is an algorithm for generating the set of k-subsets of $[n]$, and the second is an algorithm for generating all vectors in a general Cartesian product space. The algorithm of § 4 may be thought of as a generalization of the second algorithm.

2.1. The subset algorithm. The subset algorithm we shall use is a special case of a general recursive algorithm based upon the combinatorial interpretation of Pascal's formula. In order to be more specific and, eventually, to construct a UBT version of this algorithm, we introduce some terminology and notation. First we associate with a k-subset of $[n]$ an n-vector $\bar{v} = (v_1, \cdots v_n)$ with k 1's and $n-k$ 0's as entries in the natural way. A set of entries $v_i, \cdots v_j$ constitute a *block of* 1's in $\bar{v}$ if the indices $i, \cdots, j$ are consecutive, $v_i = \cdots = v_j = 1$, $v_{i-1} = 0$ if $i \neq 1$, and $v_{j+1} = 0$ if $j \neq n$. For a vector $\bar{v}$ and $p = 0,1, \cdots, n-1$, let $\bar{v}(p) = (v_{p+1}, \cdots, v_n)$. We say that $\bar{v}(p)$ is *extreme* if it has at most one block of 1's; $\bar{v}$ *is extreme* if $\bar{v}(0)$ is extreme. The following algorithm is that found in Ehrlich [2]. It utilizes two parameters p and q and labels on the entries of $\bar{v}$.

ALGORITHM 2.1.

```
1. Initialize $\bar{v}$ : $\bar{v}$ must be extreme.
   Initialize labels: if there exists $j > i$ such that $v_j \neq v_i$,
   then label $v_i$ active; otherwise, label $v_i$ passive.
2. Print $\bar{v}$.
3. Update $p : p \leftarrow \max \{0\} \cup \{i : v_i \text{ is active}\}$.
4. If $p = 0$, then stop.
5. Update $q$: if $v_p = 0$ then $q \leftarrow \min\{j : j > p \text{ and } v_j = 1\}$; otherwise, $q \leftarrow -1 + \min\{j : j > p \text{ and } v_j = 1\} \cup \{n+1\}$.
6. Update $\bar{v}$: switch the entries $v_p$ and $v_q$.
7. Update labels: labels $v_p$ passive; for $i > p$ if there exists $j > i$ such that $v_j \neq v_i$,
   then label $v_i$ active.
8. Go to 2.
```

Some properties of this algorithm and a UBT version may be found in Ehrlich [2]. An example of the list generated when $n = 6$ and $k = 3$ is given below. The bars indicate which positions are active.

$\bar{0}\bar{0}\bar{0}111$
$\bar{0}\bar{0}1\bar{0}11$
$\bar{0}\bar{0}11\bar{0}1$
$\bar{0}\bar{0}1110$
$\bar{0}1\bar{0}\bar{1}\bar{1}0$
$\bar{0}1\bar{0}\bar{1}01$
$\bar{0}1\bar{0}011$
$\bar{0}11\bar{0}\bar{0}1$
$\bar{0}11\bar{0}10$
$\bar{0}11100$
$1\bar{0}\bar{1}\bar{1}00$
$1\bar{0}\bar{1}0\bar{0}1$
$1\bar{0}\bar{1}010$
$1\bar{0}0\bar{1}\bar{1}0$
$1\bar{0}0\bar{1}01$
$1\bar{0}0011$
$11\bar{0}\bar{0}\bar{0}1$
$11\bar{0}\bar{0}10$
$11\bar{0}100$
111000

2.2. The Cartesian product space algorithm. Suppose $\bar{u} = (u_1, \cdots, u_n)$ is a vector with all positive integer entries. We wish to generate all vectors $\bar{v} = (v_1, \cdots, v_n)$ in the product space $(u_1) \times \cdots \times (u_n)$. To do this, each of the sets (u_i) is linearly ordered in some way. The first and last values in each set will be called *extreme* values. Initially we shall give an algorithm utilizing a parameter p, labels on the entries of $\bar{v}$, and an auxiliary vector $\bar{w} = (w_1, \cdots, w_n)$ where $w_i = +1$ or -1 is to indicate the direction of the next change which will take place in v_i in the linearly ordered set (u_i). We note that this algorithm is a generalization of a Gray Code.

ALGORITHM 2.2

```
1. Initialize $\bar{v}$: each $v_i$ must be extreme.
   Initialize $\bar{w}$: if $v_i$ is the first value in $(u_i)$, then $w_i \leftarrow +1$;
                if $v_i$ is the last value in $(u_i)$, then $w_i \leftarrow -1$.
   Initialize labels: if $u_i \neq 1$, label $v_i$ active;
                      if $u_i = 1$, label $v_i$ passive.
2. Print $\bar{v}$.
3. Update $p$: $p \leftarrow \max\{0\} \cup \{i: v_i \text{ is active}\}$.
4. If $p = 0$, then stop.
5. Update $\bar{v}$: if $w_p = +1$, then replace $v_p$ by its successor in $(u_p)$;
                 if $w_p = -1$, then replace $v_p$ by its predecessor in $(u_p)$.
6. Update labels: if $i > p$ and $u_i \neq 1$, then label $v_i$ active;
                  if $v_p$ is extreme, then label $v_p$ passive.
7. Update $\bar{w}$: if $v_p$ is extreme, then $w_p \leftarrow -w_p$.
8. Go to 2.
```

This algorithm clearly generates all vectors in the product space as desired. Clearly, different linear orders on the (u_i) yield different lists. For example, if for each i, we introduce the order $0, 1, \cdots, u_{i-1}$, then step 5 becomes: $v_p \leftarrow v_p + w_p$. We note that in this case, with $u_i = i$ and a natural identification between $(1) \times (2) \times \cdots \times (n)$ and the set S_n of permutations of $[n]$, Algorithm 2.2 becomes the Johnson–Trotter adjacent mark algorithm for S_n [3], [4].

In §§ 3 and 4 we shall use the order $0, u_i - 1, u_i - 2 \cdots, 1$; here, 0 and 1 are extreme values. The list generated for $(2) \times (3) \times (2)$ with starting vector $(1, 1, 0)$ follows. The bar above v_i indicates v_i is active.

$$\begin{array}{ll}
(\bar{1}, \bar{1}, \bar{0}) & (0, \bar{0}, \bar{1}) \\
(\bar{1}, \bar{1}, 1) & (0, \bar{0}, 0) \\
(\bar{1}, \bar{2}, \bar{1}) & (0, \bar{2}, \bar{0}) \\
(\bar{1}, \bar{2}, 0) & (0, \bar{2}, \bar{1}) \\
(\bar{1}, 0, \bar{0}) & (0, 1, \bar{1}) \\
(\bar{1}, 0, 1) & (0, 1, 0)
\end{array}$$

Algorithm 2.2 can be modified to give a UBT version.

3. The multi-path algorithm. Let $\mathbf{C}(n, k)$ be a set of combinatorial objects for which $C(n, k)$ satisfies a binomial recursion $C(n, k) = a_{n,k}C(n-1, k-1) + b_{n,k}C(n-1, k)$ with $a_{n,k}$ and $b_{n,k}$ nonnegative integers, and let $H_{\mathrm{C}}(n, k)$ be the corresponding binomial grid. It has already been noted in § 1 that there is a natural one-to-one correspondence between the set $\mathbf{C}(n, k)$ and the set of (n, k)-paths in $H_{\mathrm{C}}(n, k)$. Thus, any algorithm which generates this set of paths will, in effect, be an algorithm for generating the set $\mathbf{C}(n, k)$. We will give one such algorithm in this section and one in the following. We will also give conditions under which the algorithms will be UBT. Many of the standard examples of sets of combinatorial objects of the type described above will satisfy these conditions; thus, we will have combinatorial Gray codes for each of these sets.

Suppose the $a_{i,j}$ edges between the points $(i-1, j-1)$ and (i, j) in $H_{\mathrm{C}}(n, k)$ are labeled $0, 1, \cdots, a_{i,j} - 1$ and the $b_{i,j}$ edges between $(i-1, j)$ and (i, j) are labeled $0, 1, \cdots, b_{i,j} - 1$. Also suppose the edges of the binomial grid $G(n, k)$ corresponding to the k-subsets of $[n]$ are labeled with 0's and 1's in the way discussed in § 1. Then we have a one-to-one correspondence between n-vectors $\bar{s}$ of 0's and 1's with exactly k 1's and (n, k)-paths in $G(n, k)$. In turn, if each $a_{i,j}$ and $b_{i,j}$ in $H_{\mathrm{C}}(n, k)$ is nonzero, then to each (n, k)-path in $G(n, k)$ there corresponds a unique *multi-path* (a path of multi-edges) from $(0, 0)$ to (n, k) in $H_{\mathrm{C}}(n, k)$, and conversely. Suppose there are u_i edges between the $(i-1)$th and ith vertices of this multi-path ($u_i = a_{i,j}$ or $b_{i,j}$, some j). Then there is a one-to-one correspondence between the set of (n, k)-paths in $H_C(n, k)$ within the multi-path and vectors $\bar{v}$ in the product space $(u_1) \times \cdots \times (u_n)$. It follows that there is a one-to-one correspondence between the set of (n, k)-paths in $H_C(n, k)$ and ordered pairs of n-vectors $(\bar{s}, \bar{v})$ where $\bar{v} \in (u_1) \times \cdots \times (u_n)$ and $\bar{u} = (u_1, \cdots, u_n)$ is uniquely determined by $\bar{s}$.

Although we assumed in the preceding that each $a_{i,j}$ and $b_{i,j}$ in $H_C(n, k)$ is nonzero, in many examples we have $b_{1,0} = 0$; thus, there is no edge in $H_C(n, k)$ between $(0, 0)$ and $(1, 0)$ and we do not have a one-to-one correspondence between the set of (n, k)-paths in $G(n, k)$ and the set of multi-paths from $(0, 0)$ to (n, k) in $H_C(n, k)$. Where this occurs among our examples in § 1, we always have $a_{1,1} = 1$ and each $a_{i,j}$ and $b_{i,j}$ nonzero for $j \geqq 1$. It is easily seen that in this case a simple translation in the plane will reduce the situation to the previous case. Thus, we assume henceforth that each $a_{i,j}$ and $b_{i,j}$ in $H_C(n, k)$ is nonzero. Of course, if $b_{1,0} = 0$ and $a_{1,1} > 1$, then the algorithm which we shall present must be modified to accommodate the multi-edge between $(0, 0)$ and $(1, 1)$. This can be done in a straightforward manner by considering vectors where the last entry corresponds to the edge between $(0, 0)$ and $(1, 1)$; we omit any further discussion of this case.

The preceding discussion indicates how we can construct an algorithm which will generate the set of (n, k)-paths in $H_C(n, k)$ by use of Algorithms 2.1 and 2.2. In this connection, we shall henceforth use $\bar{s}$ for the code vector in the subset algorithm and retain $\bar{v}$ for the code vector in the product space algorithm. There should be no confusion between the other overlapping symbols as we shall always refer to them as corresponding to $\bar{s}$ or to $\bar{v}$.

ALGORITHM 3.1.

1. Initialize $\bar{s}$ and auxiliary data for Algorithm 2.1.
2. Initialize $\bar{v}$ and auxiliary data for Algorithm 2.2 for the $\bar{u}$ determined by $\bar{s}$.
3. Use Algorithm 2.2 to generate the product space $(u_i) \times \cdots \times (u_n)$; for each vector $\bar{v}$ generated, print $(\bar{s}, \bar{v})$. When list is complete, go to 4.
4. If $\bar{s}$ was last vector, then stop; otherwise, update $\bar{s}$ by use of Algorithm 2.1.
5. Reinitialize $\bar{v}$ and the auxiliary data for Algorithm 2.2 for the $\bar{u}$ determined by the updated $\bar{s}$.
6. Go to 3.

An appropriate modification of Algorithm 3.1 can be made UBT under certain conditions. These conditions are given below.

Conditions 3.2.

1. If $j \geqq 1$ and $i \geqq j+2$, then $b_{i,j} \geqq 2$.
2. If $b_{i,0} = 1$ and $1 \leqq h \leqq i$, then $b_{h,0} = 1$.
3. If $b_{j+1,j} = 1$ and $1 \leqq h \leqq j$, then $b_{h+1,h} = a_{h,h} = 1$.

Conditions 3.3.

1. If $j \geqq 2$ and $i \geqq j+1$, then $a_{i,j} \geqq 2$.
2. If $a_{i,i} = 1$ and $1 \leqq h \leqq i$, then $a_{h,h} = 1$.
3. If $a_{i,1} = 1$ and $1 \leqq h \leqq i-1$, then $a_{n,1} = b_{h,0} = 1$.

We note that a grid $H_C(n, k)$ will satisfy Conditions 3.3 if and only if the transposed grid $H_C^t(n, n-k)$ satisfies Conditions 3.2. Thus, for each example in § 1, either Conditions 3.2 or 3.3 will be satisfied by the grid $H_C(n, k)$.

We say that the coefficients $a_{i,j}$ in $H_C(n, k)$ are *monotone* if $a_{i',j'} = 1$ whenever $a_{i,j} = 1$, $j' \leqq j$ and $i' < i - j + j'$.

Conditions 3.4.

1. The coefficients $a_{i,j}$ are monotone.
2. $b_{i,j} = 1$.

Conditions 3.5.

1. The coefficients $b_{i,j}$ are monotone.
2. $a_{i,j}=1$.

We note that among our examples of § 1, Conditions 3.4 are satisfied by Examples 2, 3, and 6 and Conditions 3.5 are satisfied by Example 5.

If any grid $H_C(n, k)$ satisfies one of these four sets of conditions, Algorithm 3.1 can be made UBT. We conclude this section with two examples of output.

The following (Table 1) is the list of partitions of $[n]$ into k blocks (Example 1.2) when $n=5$ and $k=3$. The first vector is $\bar{s}$, the second is $\bar{v}$. These vectors are followed by the corresponding set partition, which is obtained from $\bar{s}$ and $\bar{v}$ as follows: $\bar{s}$ describes the set of smallest elements in the blocks, i.e., $s_i=1$ iff i is smallest in its block. Order the blocks in increasing order of the smallest elements, say $B_0, B_1, \cdots, B_{k-1}$. Let $j_i, \cdots, j_{n-k}$ be the remaining indices, i.e., $s_{j_i}=0$. Then $v_{j_i}=h$ iff $j_i \in B_h$ describes the placement of $j_i, \cdots, j_{n-k}$ in the blocks. (This correspondence is the same as the correspondence between placements of nontaking rooks in a triangular Ferrers board and set partitions.)

TABLE 1

10011	00000	123-4-5
10101	00000	124-3-5
10101	00010	12-34-5
10110	00000	125-3-4
10110	00002	12-3-45
10110	00001	12-35-4
11010	00001	13-25-4
11010	00002	13-2-45
11010	00000	135-2-4
11010	00100	15-23-4
11010	00102	1-23-45
11010	00101	1-235-4
11001	00100	14-23-5
11001	00110	1-234-5
11001	00010	13-24-5
11001	00000	134-2-5
11100	00000	145-2-3
11100	00002	14-2-35
11100	00001	14-25-3
11100	00021	1-25-34
11100	00022	1-2-345
11100	00020	15-2-34
11100	00010	15-24-3
11100	00012	1-24-35
11100	00011	1-245-3

Next (Table 2) we give the list of permutations of n objects with k runs (Example 1.4) when $n=5$ and $k=3$. The vectors $\bar{s}$ and $\bar{v}$ are followed by a permutation on $[n]$ with k runs constructed from $\bar{s}$ and $\bar{v}$ as follows: The numbers $\{1, 2, \cdots, n\}$ are inserted into the permutation in order. The number i is inserted in the v_ith position which creates a new run (resp. does not create a new run) if $s_i=1$ (resp. $s_i=0$).

4. Another algorithm. In this section we shall consider an alternate method of labeling the edges of the grid $H_C(n, k)$ so that not only will each (n, k)-path determine a unique n-vector $\bar{v}$, but, conversely, the vector $\bar{v}$ will uniquely determine the path. In contrast to Algorithm 3.1, the algorithm presented in this section will not list all of the paths corresponding to a particular multi-path together. Furthermore, Algorithm 3.1

TABLE 2

10011	00000	54123	10110	00111	14352	11001	00111	21534
10011	00002	41253	10110	00112	14325	11001	00112	21354
10011	00001	41523	10110	00110	14532	11001	00110	52134
10011	00021	15243	10110	00100	45132	11001	00100	52413
10011	00022	12543	10110	00102	41325	11001	00102	24153
10011	00020	51243	10110	00101	41352	11001	00101	25413
10011	00010	51423	10110	00001	43512	11001	00001	25341
10011	00012	14253	10110	00002	43125	11001	00002	23541
10011	00011	15423	10110	00000	45312	11001	00000	52341
10101	00011	31524	10110	00010	35142	11001	00010	52314
10101	00012	31254	10110	00012	31425	11001	00012	23154
10101	00010	53124	10110	00011	31452	11001	00011	25314
10101	00000	53412	11010	00011	24351	11100	00011	32451
10101	00002	34152	11010	00012	24315	11100	00012	32415
10101	00001	35412	11010	00010	24531	11100	00010	35241
10101	00101	15342	11010	00000	45231	11100	00020	35214
10101	00102	13542	11010	00002	42315	11100	00022	32145
10101	00100	51342	11010	00001	42351	11100	00021	32514
10101	00110	51324	11010	00101	42513	11100	00001	34251
10101	00112	13254	11010	00102	42135	11100	00002	34215
10101	00111	15324	11010	00100	45213	11100	00000	34521
			11010	00110	25143			
			11010	00112	21435			
			11010	00111	21453			

requires a pair of vectors, $(\bar{s}, \bar{v})$, to describe the object while the algorithm in this section uses a single vector. The algorithm which we will construct to generate this set of vectors will in many ways be similar to Algorithm 2.2 and, loosely speaking, is a generalization of it.

For the sake of simplicity in notation, we shall assume that for a given n and k, if (i, j) is not a vertex of the grid $H_C(n, k)$, then $a_{i,j} = b_{i,j} = 0$. However, as in § 3, we assume that each $a_{i,j}$ and $b_{i,j}$ in $H_C(n, k)$ is nonzero. For any vertex V of the grid, let deg V = out-degree of V; in particular, $\deg(i, j) = b_{i+1,j} + a_{i+1,j+1}$. Now the $b = b_{i+1,j}$ horizontal edges directed out from (i, j) are to be labeled $0, 1, \cdots, b-1$ and the $a = a_{i+1,j+1}$ diagonal edges directed out from (i, j) are to be labeled $b, b+1, \cdots, b+a-1$. Thus, the $d = \deg(i, j)$ edges directed out from (i, j) are labeled $0, 1, \cdots, d-1$.

Consider a multi-path Q in $H_C(n, k)$. Let $V_0, \cdots, V_n$ where $V_i = (i, j_i)$ be the vertices of Q; let $u_i = \deg V_{i-1}$. Then for each (n, k)-path P within Q, that is, through the vertices $V_0, \cdots, V_n$, if $\bar{v}$ corresponds to P, then $\bar{v} \in (u_1) \times \cdots \times (u_n)$. Thus, to each multi-path Q there corresponds a unique vector $\bar{u} = (u_1, \cdots, u_n)$ such that for each path P within Q, the corresponding vector $\bar{v}$ belongs to the product space determined by $\bar{u}$.

However, it should be noted that not all vectors $\bar{v}$ belonging to this product space correspond to a path within Q; in fact, there may be vectors $\bar{v}$ for which there is no corresponding path in $H_C(n, k)$. In what follows, if path P corresponds to vector $\bar{v}$, P is within multi-path Q, and $\bar{u}$ corresponds to Q, then we shall say that $\bar{u}$ corresponds to $\bar{v}$.

We next consider what modifications are necessary in Algorithm 2.2 if we are to generate only those vectors $\bar{v}$ which correspond to paths in $H_C(n, k)$. Suppose we have $\bar{u}$ corresponding to $\bar{v}$ and we are proceeding as though we were generating the product space $(u_1) \times \cdots \times (u_n)$. Suppose $\bar{v}$ has right-most active entry v_p, the path P corresponding to $\bar{v}$ passes through the vertices $V_0, \cdots, V_n$ and v_p and $\bar{v}$ are to be updated to v_p' and $\bar{v}'$, respectively. There are two cases to consider initially.

Case 1. The edge labeled v'_p directed out of V_{p-1} is directed into V_p (as is the edge labeled v_p).

Case 2. The edge labeled v'_p directed out of V_{p-1} is not directed into V_p.

In Case 1, there is a path P' within Q corresponding to $\bar{v}'$; no modification is necessary here, just generate $\bar{v}'$ and then proceed to the next vector in the product space determined by $\bar{u}$. In Case 2, in updating v_p to v'_p, we have switched from a horizontal edge directed out of V_{p-1} to a diagonal one, or the reverse. In the former case we say that the vector $\bar{v}$, or the path P is to undergo an *up-switch* and, in the reverse case, a *down-switch*; in either case, we call V_{p-1} the *active vertex of the path P*. Whether $\bar{v}$ is to undergo an up-switch or a down-switch, there are two subcases to consider.

Case 2a. There exists a path P' in $H_C(n, k)$ corresponding to $\bar{v}'$.

Case 2b. There does not exist a path in $H_C(n, k)$ corresponding to $\bar{v}'$.

In Case 2a the path P' belongs to a multi-path Q' where $Q' \neq Q$. Thus, after generating $\bar{v}'$, we must update the vector $\bar{u}$, say to $\bar{u}'$, to correspond to Q'. We then proceed as though we were generating the product space $(u'_1) \times \cdots \times (u'_n)$ where $\bar{u}' = (u'_1, \cdots, u'_n)$.

To analyze Case 2b, we first note that each set (u_i) is to be ordered $0, u_1 - 1, \cdots, 2, 1$. Also, if v_p is the right-most active entry of $\bar{v}$, then for each $i > p$, v_i is extreme (0 or 1). Finally, for any vertex V not on the upper or right-hand boundaries of the grid, we have $\deg V \geqq 2$ and there must be edges directed out of V labeled 0 and 1. Thus, in Case 2b, there must be an index $q \geqq p$ and a partial path through the vertices $V_0, \cdots, V_{p-1}, V'_p, \cdots V'_q$ determined by the q-vector $(v_1, \cdots, v_{p-1}, v'_p, v_{p+1}, \cdots, v_q)$ such that V'_q is either on the upper or right-hand boundary of the grid and $v_{q+1} = 1$ while $\deg V'_q = 1$ (hence, the only edge directed out of V'_q is labeled 0). Here we must modify $\bar{v}'$ by setting $v_{q+1} = 0$ and similarly for any other such v_i for $i > q + 1$. The vector $\bar{v}'$ so modified satisfies the condition of Case 2a and we proceed as in that case.

We are now in a position to give our algorithm. It is based on Algorithm 2.2 and we utilize the same notation.

ALGORITHM 4.1.

```
1. Initialize $\bar{v}$: any vector with each $v_i$ extreme for which there exists a cor-
                  responding path in the grid.
   Initialize $\bar{u}$: determine $\bar{u}$ corresponding to $\bar{v}$.
   Initialize $\bar{w}$: as in Algorithm 2.2.
   Initialize labels: as in Algorithm 2.2.
2. Print $\bar{v}$.
3. Update $p$: as in Algorithm 2.2.
4. If $p = 0$, then stop.
5. Update $\bar{v}$: as in Algorithm 2.2.
6. Update labels: as in Algorithm 2.2.
7. Update $\bar{w}$: as in Algorithm 2.2.
8. If Case 2a above prevails, then update $\bar{u}$ as indicated.
9. If Case 2b above prevails, then update $\bar{v}$ and $\bar{u}$ as indicated.
10. Go to 2.
```

It is easily shown by induction on $n + k$ that Algorithm 4.1 will generate the set of (n, k)-paths in the grid as desired for any "proper" initial vector.

It is clear that Algorithm 4.1 will be UBT if steps 8 and 9 can be carried out in a uniformly bounded manner and a UBT version of Algorithm 2.2 is used. However, in examples where many of the coefficients $a_{i,j}$ and $b_{i,j}$ are equal to one, the algorithm occasionally produces rather unpredictable changes in the multi-path Q (and hence, in the vector $\bar{u}$) when $\bar{v}$ is updated. To avoid such situations we shall only consider grids which satisfy Conditions 3.2 or 3.3. A modification of Algorithm 4.1 will be UBT for such grids.

Remark 4.2. In Algorithm 4.1 the representations of the combinatorial objects in the examples may seem "closer" to the objects if the edges on the right hand boundary of the grid were labeled in a more "natural" manner. For instance, in Example 2, $a_{n-i,k-1} = 1$ and this edge would be labeled $k-i$. Of course, some modifications to Algorithm 4.1 would be necessary. The algorithm thus constructed for the example is that found in Ehrlich [2].

We now give two examples of the output from Algorithm 4.1. First (Table 3) we list the partitions of $[n]$ into k blocks (Example 1.2) where $n=5$ and $k=3$. The vector given is $\bar{v}$ modified by Remark 4.2. This is followed by the corresponding set partition which is obtained as follows: v_i indicates that i is in block v_i. These vectors are called "restricted growth functions".

TABLE 3

00012	123-4-5	01201	14-25-3
00102	124-3-5	01221	1-25-34
00120	125-3-4	01222	1-2-345
00122	12-3-45	01220	15-2-34
00121	12-35-4	01210	15-24-3
00112	12-34-5	01212	1-24-35
01012	13-24-5	01211	1-245-3
01020	135-2-4	01112	1-234-5
01022	13-2-45	01120	15-23-4
01021	13-25-4	01122	1-23-45
01002	134-2-5	01121	1-235-4
01200	145-2-3	01102	14-23-5
01202	14-2-35		

TABLE 4

00012	54123	00114	31254	01013	25314
00014	41253	00112	53124	01201	34251
00013	41523	00120	45312	01202	34215
00033	15243	00122	43125	01200	34521
00034	12543	00121	43512	01220	35214
00032	51243	00131	31452	01222	32145
00022	51423	00132	31425	01221	32514
00024	14253	00130	35142	01211	32451
00023	15423	00102	53412	01212	32415
00203	15342	00104	34152	01210	35241
00204	13542	00103	35412	01112	52134
00202	51342	01003	25341	01114	21354
00230	14532	01004	23541	01113	21534
00232	14325	01002	52341	01120	45213
00231	14352	01030	24531	01122	42135
00221	41352	01032	24315	01121	42513
00222	41325	01031	24351	01131	21453
00220	45132	01021	42351	01132	21435
00212	51324	01022	42315	01130	25143
00214	13254	01020	45231	01102	52413
00213	15324	01012	52314	01104	24153
00113	31524	01014	23154	01103	25413

Next (Table 4) we give the list of permutations of n objects with k runs (Example 1.4) when $n = 5$ and $k = 3$. The vector $\bar{v}$ (modified by Remark 4.2) will be followed by the corresponding permutation constructed from $\bar{v}$ as follows: assume $1, 2, \cdots, i-1$ have been inserted into the permutation. Let l be the number of runs in this permutation. If $v_i \geqq l$, insert i in the $(v_i - l)$th position which creates a new run. If $v_i < l$, insert i at the end of the v_ith run.

5. Remarks.

Remark 5.1. In all of the examples described in § 1, the values $a_{i,j}$ and $b_{i,j}$ may be computed as needed (in a UB manner) instead of stored. Thus, in these cases, Algorithms 3.1 and 4.1 may be made into linear UBT algorithms.

Remark 5.2. As was described in the Introduction, combinatorial objects may have a number of "natural" representations or codings. The ones presented in §§ 3 and 4 were chosen, of course, to obtain the UBT property. However, since they are based upon recursions for which there are natural combinatorial proofs, they themselves turn out to be quite natural.

For example, the representation of a partition of a set obtained in Example 1.2 from Algorithm 4.1 (as modified by Remark 4.2) yields the second representation described in the Introduction. The representation of a partition of a set obtained from Algorithm 3.1 may be interpreted as placements of nontaking rooks on a triangular chessboard.

Remark 5.3. In this paper we have not addressed the question of rank/unrank algorithms. Such algorithms are straightforward for Algorithms 2.1 and 2.3. However, in Algorithm 3.1, because of the problem of predicting the first $\bar{v}$ associated with a given $\bar{s}$, such an algorithm may be difficult to construct. A rank/unrank algorithm associated with Algorithm 4.1 seems more likely.

Acknowledgment. We would like to thank the referees for their suggestions and comments which have made this paper more readable.

REFERENCES

[1] J. R. BITNER, G. EHRLICH AND E. M. REINGOLD, *Efficient generation of the binary reflected Gray code and its applications*, Comm. ACM, 19 (1976), pp. 517–521

[2] G. EHRLICH, *Loopless algorithms for generating permutations, combinations, and other combinatorial configurations*, J. Assoc. Comput. Mach., 20 (1973), pp. 500–513.

[3] S. M. JOHNSON, *Generation of permutations by adjacent transpositions*, Math. Comput., 17 (1963), pp. 282–285.

[4] H. F. TROTTER, *Algorithm 115 Perm.*, Comm. ACM, 5 (1962), pp. 434–435.

[5] H. S. WILF, *A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects*, Advances in Math., 24 (1977), pp. 281–291.

[6] S. G. WILLIAMSON, *On the ordering, ranking, and random generation of basic combinatorial sets*, Table Ronde, Combinatoire et Représentation du Groupe Symétrique, D. Foata, Ed., Strasbourg, 26–30 April 1976. (Available through D. Foata, Département de Mathématique, Université Louis-Pasteur de Strasbourg, 7, rue René-Descartes, 67084 Strasbourg, France.)

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0014 $01.00/0

A NOTE ON GRAY CODE AND ODD-EVEN MERGE*

P. FLAJOLET† AND LYLE RAMSHAW‡

Abstract. Delange has demonstrated an elegant method for computing the sum of all of the digits used when the first n nonnegative integers are expressed in base $q \geqq 2$. We show that his method can be adapted to unusual number systems such as Gray code and balanced ternary and can also be adapted to count the occurrences of each digit separately. As an application, we consider Sedgewick's analysis of Batcher's odd-even merge, and use our results about Gray code to provide an alternative, and perhaps more direct, derivation of the asymptotics of the average case.

Key words. analysis of algorithms, digital sums, Gray code, odd-even merge, merge exchange sort, gamma function, zeta function

1. Introduction. The performance of interesting algorithms sometimes depends upon the properties of a number system. The basic operations on Vuillemin's binomial queues [13], for example, are closely related to the arithmetic of the binary number system. Hence, the analysis of binomial queues demands information about the function $\beta(n)$ that maps the nonnegative integer n into the number of 1-bits in its binary expansion. A more subtle example is provided by the question of determining how many registers are needed to evaluate an arithmetic expression optimally. Let R_n represent the expected number of registers needed to evaluate an expression tree with n binary operations, where the $(1/(n+1))\binom{2n}{n}$ such trees are considered equally likely. Here, the relation to the binary number system is not nearly so direct. Yet R_n can be expressed in terms of a convolution of $\beta(n)$ with binomial coefficients [4]; it turns out that

$$R_n = 1+(n+1)\sum_{i>0}\beta(i)\left[\frac{\binom{2n}{n+i+2}-3\binom{2n}{n+i+1}+3\binom{2n}{n+i}-\binom{2n}{n+i-1}}{\binom{2n}{n}}\right].$$

The determination of the asymptotic behavior of such a convolution can present something of a challenge. One method of attack uses integral transforms in the complex plane, often beginning with a Mellin transform of e^{-x} of the form

$$e^{-x}=\frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty}\Gamma(z)x^{-z}\,dz.$$

Knuth calls this approach the *gamma function method* [8], and attributes it to N. G. de Bruijn. R. Kemp uses complex integral transforms to handle R_n [6], and there are several other examples of this method in the literature [2], [11].

Flajolet, Raoult, and Vuillemin established the asymptotic behavior of R_n independently, using a different technique [4]. After applying summation by parts to replace $\beta(i)$ with its sum, they invoked the following result of Delange [3]. (We will write "lg" for "$\log_2$" and "ln" for "$\log_e$" throughout.)

* Received by the editors December 7, 1977, and in revised form March 23, 1979.

† IRIA LABORIA, Rocquencourt, Le Chesnay, France.

‡ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by the National Science Foundation under Grant MCS-77-23738 and by the Office of Naval Research under Contract N00014-76-C-0330.

THEOREM B (H. Delange) (The number of 1-bits in binary). *Let $\beta(n)$ denote the number of 1-bits in the binary representation of n. There exists a continuous, nowhere differentiable function $B:\mathbf{R}\rightarrow\mathbf{R}$, periodic with period* 1, *such that*

$$\sum_{0\leqq i<n}\beta(i)=\frac{n\lg n}{2}+nB(\lg n)\quad \textit{for } n\geqq 1.$$

Furthermore, the Fourier series $B(x)=\sum_k b_k e^{2k\pi i x}$ of B converges absolutely, and its coefficients b_k are given by

$$b_0=\frac{\lg \pi}{2}-\frac{1}{2\ln 2}-\frac{1}{4}=-0.145599^+,$$

$$b_k=\frac{-\zeta(\chi_k)}{(\ln 2)\chi_k(1+\chi_k)}\quad \textit{for } \chi_k=\frac{2k\pi i}{\ln 2},\ k\neq 0.$$

The example of R_n thus justifies an interest in digital sums like the one in Theorem B. Our purpose in this note is to show that the method behind Delange's proof of Theorem B can be used to handle other digital sums in standard and exotic number systems. For example, let $\gamma(n)$ for $n\geqq 0$ denote the number of 1-bits in the Gray code representation of n. Section 2 derives a formula for $\sum_i \gamma(i)$ that constitutes the Gray code analogue of Theorem B. The proof is a straightforward modification of Delange's method, but it is presented here for completeness. In § 3, we define a fairly broad class of positional number systems in which Delange's method can be used to count the number of occurrences of each nonzero digit.

In § 4, we consider an application to the analysis of odd-even merging. Sedgewick has expressed the average case exchange performance of Batcher's odd-even merge in terms of a convolution of the function $\gamma(n)$ with binomial coefficients [11]. He went on to determine the asymptotic behavior of this average case by means of the gamma function method mentioned above. By invoking the Gray code analogue of Theorem B, we will be able to provide an alternative derivation of the asymptotic behavior.

2. The Gray code case. A *Gray code* [10, pp. 173–179, 198] is an encoding of the integers as sequences of 0's and 1's with the property that the representations of adjacent integers differ in exactly one position. We will restrict our consideration to the *standard Gray* (or *binary reflected*) code, which encodes n as the binary representation of $g(n)$ where g: N→N is defined by

$$g(0)=0,\qquad g(2^p+j)=2^p+g(2^p-1-j)\quad \text{for } 0\leqq j<2^p.$$

The Gray code representations of the first sixteen integers are presented in Table 1.

Let $\gamma_k(n)$ denote the kth bit in the Gray code representation of n, and let $\gamma(n)=\sum_k \gamma_k(n)$ denote the total number of 1-bits in that sequence. In the notation of Theorem B, we have $\gamma(n)=\beta(g(n))$. We wish to count the 1-bits in the first n rows of Table 1, that is, to evaluate the sum

$$F(n)=\sum_{0\leqq i<n}\gamma(i).$$

Again, our argument closely parallels Delange's proof [3] of Theorem B.

We begin by considering the columns of the table. Note that the kth column consists of an infinite repetition of the block of bits $0^{2^k}1^{2^k}1^{2^k}0^{2^k}$. Eventually, therefore, each column contains about half 0's and half 1's. Let $t_k(n)$ denote the difference between the number of 1-bits actually present in the first n positions of column k and the number of 1-bits that we would expect to find there, namely $n/2$; in symbols, we

TABLE 1
The standard Gray code representation.

3	2	1	0	k / n
			0	0
			1	1
		1	1	2
		1	0	3
	1	1	0	4
	1	1	1	5
	1	0	1	6
	1	0	0	7
1	1	0	0	8
1	1	0	1	9
1	1	1	1	10
1	1	1	0	11
1	0	1	0	12
1	0	1	1	13
1	0	0	1	14
1	0	0	0	15

have

$$t_k(n)=\Bigl(\sum_{0\leqq i<n}\gamma_k(i)\Bigr)-\frac{n}{2}.$$

Whenever n is a multiple of 2^k, the relevant portion of the kth column will consist of an integral number of blocks of 2^k bits, each entirely 0's or entirely 1's, occurring in the order mentioned above. This data allows us to compute special values of $t_k(n)$; in particular, we deduce that

$$t_k(p2^k)=\begin{cases} 0 & \text{if } p\equiv 0 \pmod 4\\ -2^{k-1} & \text{if } p\equiv 1 \pmod 4\\ 0 & \text{if } p\equiv 2 \pmod 4\\ 2^{k-1} & \text{if } p\equiv 3 \pmod 4).\end{cases}$$

Furthermore, between these special values of n, the graph of $t_k(n)$ will consist of linear segments. Hence, we can express all of the t_k's as rescalings of a single periodic function $t\colon \mathbf{R}\to\mathbf{R}$, shown in Figure 1 and formally defined by

$$(2.1)\qquad t(x)=\begin{cases} -x/2 & \text{if } 0\leqq x\leqq \frac14\\ x/2-\frac14 & \text{if } \frac14\leqq x\leqq\frac34\\ \frac12-x/2 & \text{if } \frac34\leqq x\leqq 1\end{cases}\quad \text{and}\quad t(x+1)=t(x)\quad \text{for all } x.$$

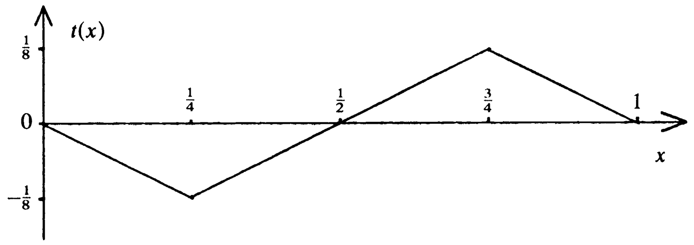


FIG. 1. *The graph of $t(x)$.*

In particular, for all k and n, we have

$$t_k(n) = 2^{k+2}t(n/2^{k+2}).$$

Let $l = \lfloor \lg n \rfloor$, and note that all of the columns in Table 1 of index greater than l begin with at least n zeros. Therefore, we have

$$F(n) = \sum_{0 \leqq k \leqq l} \left(\frac{n}{2} + t_k(n)\right) = \frac{(l+1)n}{2} + \sum_{0 \leqq k \leqq l} 2^{k+2}t(n/2^{k+2}).$$

The lower bound on the index of the remaining summation can be dropped, since $t(x)$ equals zero whenever x is integral or half-integral; replacing that index of summation k by $(l-k)$ then gives

$$F(n) = \frac{(l+1)n}{2} + \sum_{k \geqq 0} 2^{l-k+2}t(n/2^{l-k+2}). \tag{2.2}$$

The latter term suggests that we consider the superposition function

$$h(x) = \sum_{k \geqq 0} \frac{t(2^k x)}{2^k}. \tag{2.3}$$

This function is of some mathematical interest in its own right; it is a close analogue of Van der Waerden's example of a function that is continuous but nowhere differentiable [12, pp. 351–354]. In fact, we can show the following.

LEMMA H. *The function* $h\colon \mathbf{R} \to \mathbf{R}$ *defined by* (2.3) *is periodic with period* 1, *continuous, and nowhere differentiable.*

Proof. The periodicity of h follows immediately from the periodicity of t. Also, since t is bounded, we can apply the Weierstrass M-test to deduce that the series defining h converges uniformly, and hence that h is continuous. It only remains to verify nondifferentiability.

Given a real number y, let $I_j = [a_j, b_j]$ for $j \geqq 1$ denote the closed interval of the form $[p/2^j, (p+1)/2^j]$ that contains y. The I_j form a nested sequence of intervals whose intersection contains only y. If $2^k y$ is an integer, there will be two possibilities for I_j when $j \geqq k$, and we can choose either one as long as we choose consistently.

If $h(x)$ were differentiable at $x = y$, the difference quotient

$$\frac{h(b_j) - h(a_j)}{b_j - a_j} \tag{2.4}$$

would have to converge to $h'(y)$ as j went to infinity. From (2.3), we have

$$\frac{h(b_j) - h(a_j)}{b_j - a_j} = \sum_{k \geqq 0} \frac{t((p+1)2^{k-j}) - t(p2^{k-j})}{2^{k-j}}.$$

Since $t(x) = 0$ for integral or half-integral x, the kth term of this sum is zero for $k \geqq j-1$. On the other hand, the kth term for $0 \leqq k < j-1$ will contribute $\pm\frac{1}{2}$, since $t(x)$ is linear over the interval $[p2^{k-j}, (p+1)2^{k-j}]$. Thus, twice the difference quotient (2.4) will be an even or odd integer depending as j is odd or even; in particular, the difference quotient cannot converge, and thus h is not differentiable at y. □

In terms of this new function h, our formula (2.2) for $F(n)$ takes the form

$$F(n) = \frac{(l+1)n}{2} + 2^{l+2}h(n/2^{l+2}).$$

Recalling that $l=\lfloor \lg n\rfloor$, we can rewrite this

$$F(n)=\frac{n\lg n}{2}+\frac{n(\lfloor \lg n\rfloor-\lg n+1)}{2}+n2^{\lfloor \lg n\rfloor-\lg n+2}h(2^{\lg n-\lfloor \lg n\rfloor-2}).$$

Using $\{x\}=x-\lfloor x\rfloor$ to denote the fractional part of x, define the function $G\colon \mathbf{R}\to\mathbf{R}$ by

$$G(x)=\frac{1-\{x\}}{2}+2^{2-\{x\}}h(2^{\{x\}-2});\tag{2.5}$$

then, we have

$$F(n)=\frac{n\lg n}{2}+nG(\lg n).$$

Now, the definition of $G(x)$ in Equation (2.5) immediately implies that G is periodic with period 1; it also shows that the identity

$$G(x)=\frac{1-x}{2}+2^{2-x}h(2^{x-2})\tag{2.6}$$

holds for x in the range $0\leqq x<1$. In fact, this identity holds over the entire range $x\leqq 1$; to see this, it is enough to note that, for $x\leqq 0$, we have

$$\begin{aligned}\frac{1-x}{2}+2^{2-x}h(2^{x-2})&=\frac{1-x}{2}+2^{2-x}(t(2^{x-2})+\tfrac{1}{2}h(2^{x-1}))\\&=\frac{1-x}{2}+2^{2-x}((-\tfrac{1}{2})2^{x-2}+\tfrac{1}{2}h(2^{x-1}))\\&=\frac{1-(x+1)}{2}+2^{2-(x+1)}h(2^{(x+1)-2}).\end{aligned}$$

In particular, we may conclude that the behavior of $G(x)$ over the interval $(-1,1)$ is a smoothly distorted version of the behavior of $h(x)$ over the interval $(\frac{1}{8},\frac{1}{2})$. This means that $G(x)$ must be continuous and nowhere differentiable on $(-1,1)$, and hence on the whole real line by periodicity.

We now turn to the task of computing the Fourier series of G. We want to express G in the form

$$G(x)=\sum_k g_k e^{2k\pi ix},$$

where the coefficients g_k are determined by Fourier inversion,

$$g_k=\int_0^1 G(u)e^{-2k\pi iu}\,du.$$

But is behooves us to be a trifle cautious, since we are dealing with fairly ill-behaved functions. Since G is continuous, we know from Fejér's Theorem [12, p. 414] that the Fourier series of G will be summable (C, 1) everywhere. But, since G is also nowhere differentiable, it cannot possibly be of bounded variation, and hence there is no particular reason to assume that the Fourier series of G will converge. Therefore, after we compute the coefficients g_k, we will have to explicitly explore the convergence question.

First, we must compute the coefficients. Substituting for $G(u)$ its definition from (2.5), we have $g_k = c_k + d_k$ where

$$c_k = \int_0^1 \left(\frac{1-u}{2}\right) e^{-2k\pi i u}\,du,$$

$$d_k = \int_0^1 2^{2-u} h(2^{u-2}) e^{-2k\pi i u}\,du.$$

We can calculate the c_k at once, getting $c_0 = \frac{1}{4}$ and $c_k = 1/(4k\pi i)$ for $k \neq 0$. Turning to d_k, recall that (2.3) defined $h(x)$ as the sum of a uniformly convergent series: therefore, after substituting for $h(2^{u-2})$, we may interchange summation and integration to get

$$d_k = \sum_{j \geqq 0} \int_0^1 2^{2-u-j} t(2^{u+j-2}) e^{-2k\pi i u}\,du.$$

Performing the change of variables $u = 2 - j + \lg v$ then gives us

$$d_k = \frac{1}{\ln 2} \sum_{j \geqq 0} \int_{2^{j-2}}^{2^{j-1}} \frac{t(v)\,dv}{v^{2+2k\pi i/\ln 2}}.$$

Adopting the abbreviation $\chi_k = 2k\pi i/\ln 2$, we have

$$d_k = \frac{1}{\ln 2} \int_{1/4}^{\infty} \frac{t(v)\,dv}{v^{2+\chi_k}} = \frac{1}{\ln 2} H(1 + \chi_k) \tag{2.7}$$

where $H(z)$ represents the integral

$$H(z) = \int_{1/4}^{\infty} \frac{t(v)}{v^{z+1}}\,dv.$$

Now, although this integral is absolutely convergent for $\Re(z) > 0$, it is easy to evaluate only when $\Re(z) > 2$. Taking advantage of the fact that $H(z)$ is analytic for $\Re(z) > 0$, we will first evaluate it under the more restrictive assumption that $\Re(z) > 2$, and then use analytic continuation to extend our result.

Equation (2.1) is the original definition of the function $t(x)$; working from this, it is easy to verify the alternative form

$$t(v) = \int_0^v \left(\lfloor x + \tfrac{3}{4} \rfloor - \lfloor x + \tfrac{1}{4} \rfloor - \tfrac{1}{2}\right) dx.$$

Integrating $H(z)$ by parts then yields

$$H(z) = \frac{-4^{z-1}}{2z} + \frac{1}{z} \int_{1/4}^{\infty} \left(\lfloor v + \tfrac{3}{4} \rfloor - \lfloor v + \tfrac{1}{4} \rfloor - \tfrac{1}{2}\right) \frac{dv}{v^z}. \tag{2.8}$$

Now, using our assumption that $\Re(z) > 2$, we can split this integral into three parts; the third is straightforward, since

$$\int_{1/4}^{\infty} \frac{dv}{v^z} = \frac{4^{z-1}}{z-1}.$$

The first two parts turn out to involve the generalized Riemann zeta function defined for $\Re(z) > 1$ by

$$\zeta(z, \alpha) = \sum_{j \geqq 0} \frac{1}{(j + \alpha)^z}; \tag{2.9}$$

the standard Riemann zeta function $\zeta(z)$ corresponds to the case $\alpha = 1$. We calculate as follows, for $0 < \alpha \leqq 1$:

$$\begin{aligned}(z-1)\int_{\alpha}^{J+\alpha} \lfloor v+1-\alpha \rfloor \frac{dv}{v^z} &= \sum_{1 \leqq j \leqq J} j \int_{\alpha+j-1}^{\alpha+j} \frac{(z-1)\,dv}{v^z} \\ &= \sum_{1 \leqq j \leqq J} j\left(\frac{1}{(\alpha+j-1)^{z-1}} - \frac{1}{(\alpha+j)^{z-1}}\right) \\ &= \left(\sum_{0 \leqq j < J} \frac{1}{(j+\alpha)^{z-1}}\right) - \frac{J}{(J+\alpha)^{z-1}}.\end{aligned}$$

Recalling that $\Re(z) > 2$, we let J go to infinity and conclude that

$$(z-1)\int_{\alpha}^{\infty} \lfloor v+1-\alpha \rfloor \frac{dv}{v^z} = \zeta(z-1, \alpha).$$

Plugging into our formula for $H(z)$ in (2.8), we deduce that, at least for $\Re(z) > 2$, we have

$$H(z) = \frac{-4^{z-1}}{2(z-1)} + \frac{\zeta(z-1, \frac{1}{4}) - \zeta(z-1, \frac{3}{4})}{z(z-1)}.$$

By analytic continuation, this formula also holds for $\Re(z) > 0$ if $z \neq 1$.

Some simplification is now possible. For any integer $l \geqq 2$, we can group the terms of the series in (2.9) according to the residue class of j modulo l, and verify that

$$\zeta(z, 1/l) + \zeta(z, 2/l) + \cdots + \zeta(z, 1) = l^z \zeta(z).$$

Invoking this relation for $l = 2$ and for $l = 4$, we see that

$$\zeta(z, \tfrac{3}{4}) = (4^z - 2^z)\zeta(z) - \zeta(z, \tfrac{1}{4});$$

hence, we have

$$H(z) = \frac{-4^{z-1}}{2(z-1)} + \frac{2\zeta(z-1, \frac{1}{4})}{z(z-1)} - \frac{2(4^{z-1} - 2^{z-1})\zeta(z-1)}{z(z-1)}.$$

We can now compute the d_k for $k \neq 0$ as specified in (2.7) by substituting $(1 + \chi_k)$ for z; the third term conveniently drops out, and we have

$$d_k = \left(\frac{-1}{4k\pi i} + \frac{2\zeta(\chi_k, \frac{1}{4})}{(\ln 2)\chi_k(1 + \chi_k)}\right) \quad \text{for } k \neq 0.$$

To compute d_0, we need the value $H(1)$, which we can find using the expansions

$$\begin{aligned}&4^{z-1} = 1 + 2\ln 2(z-1) + O((z-1)^2) \\ &2^{z-1} = 1 + \ln 2(z-1) + O((z-1)^2) \\ &\frac{1}{z} = 1 - (z-1) + O((z-1)^2) \\ &\zeta(2-1, \tfrac{1}{4}) = \tfrac{1}{4} + \left(\ln \Gamma(\tfrac{1}{4}) - \frac{\ln(2\pi)}{2}\right)(z-1) + O((z-1)^2);\end{aligned}$$

the first three are trivial, and the last is classical [14, p. 271]. Plugging in and grinding away, we arrive at

$$H(z) = 2\ln \Gamma(\tfrac{1}{4}) - \frac{3\ln 2}{2} - \ln \pi - \tfrac{1}{2} + O(z-1)$$

which implies that

$$d_0 = 2\lg\Gamma(\tfrac{1}{4}) - \lg\pi - \frac{1}{2\ln 2} - \frac{3}{2}.$$

Recalling that $g_k = c_k + d_k$, we have found that the Fourier coefficients of G are

$$g_0 = 2\lg\Gamma(\tfrac{1}{4}) - \lg\pi - \frac{1}{2\ln 2} - \frac{5}{4},$$

$$g_k = \frac{2\zeta(\chi_k, \frac{1}{4})}{(\ln 2)\chi_k(1+\chi_k)} \quad \text{for } \chi_k = \frac{2k\pi i}{\ln 2}, \quad k \neq 0.$$

As mentioned earlier, we have no reason yet for thinking that the Fourier series of G will converge; but we can now check convergence explicitly. Since $\zeta(it, \alpha) = O(|t|^{1/2}\log|t|)$ [14, p. 276], the coefficients satisfy

$$g_k = O(|k|^{-3/2}\log|k|),$$

and hence the Fourier series of G converges absolutely on the entire real line. This completes the proof of the 1-bit counting theorem for Gray code, which we now state formally.

THEOREM G. (The number of 1-bits in Gray code). *Let $\gamma(n)$ denote the number of 1-bits in the standard Gray code representation of n. There exists a continuous, nowhere differentiable function G: $\mathbf{R} \to \mathbf{R}$, periodic with period* 1, *such that*

$$\sum_{0\leqq i<n} \gamma(i) = \frac{n\lg n}{2} + nG(\lg n) \quad \textit{for } n \geqq 1.$$

Furthermore, the Fourier series $G(x) = \sum_k g_k e^{2k\pi i x}$ of G converges absolutely, and its coefficients g_k are given by

$$g_0 = 2\lg\Gamma(\tfrac{1}{4}) - \lg\pi - \frac{1}{2\ln 2} - \tfrac{5}{4} = 0.093604^+,$$

$$g_k = \frac{2\zeta(\chi_k, \frac{1}{4})}{(\ln 2)\chi_k(1+\chi_k)} \quad \textit{for } \chi_k = \frac{2k\pi i}{\ln 2}, \ k \neq 0.$$

3. Balanced ternary and other systems. The same technology that handled Theorems B and G can be used to count the occurrences of each nonzero digit d in a fairly broad class of positional number systems. We will begin this subject by considering the case of balanced ternary.

Balanced ternary [7] is the base 3 positional number system obtained by using the ternary digits or *trits* -1, 0, and 1 instead of 0, 1, and 2. Every integer can be written uniquely in balanced ternary without the use of an explicit sign. Table 2 depicts the balanced ternary representations of the first sixteen nonnegative integers, where we use $\bar{1}$ to stand for -1.

Paralleling our Gray code development, let d represent either 1 or -1, and let $\tau_k(n)$ be 1 if d is the trit in the kth position when n is expressed in balanced ternary, and 0 otherwise. Also, let $\tau(n)$ denote the total number of d-trits in the balanced ternary representation of n, so that $\tau(n) = \sum_k \tau_k(n)$. We want to study the function $\sum_i \tau(i)$.

Now, the zeroth column in Table 2 consists of an infinite repetition of the block of trits $01\bar{1}$. In order for the exact analogue of our Gray code argument to work, it would be essential that the kth column consist of repetitions of the block of trits $0^{3^k}1^{3^k}\bar{1}^{3^k}$. Unfortunately, this is *not* the case. But, there is a solution to this difficulty. Imagine

TABLE 2
The balanced ternary system.

3	2	1	0	k / n
			0	0
			1	1
		1	$\bar{1}$	2
		1	0	3
		1	1	4
	1	$\bar{1}$	$\bar{1}$	5
	1	$\bar{1}$	0	6
	1	$\bar{1}$	1	7
	1	0	$\bar{1}$	8
	1	0	0	9
	1	0	1	10
	1	1	$\bar{1}$	11
	1	1	0	12
	1	1	1	13
1	$\bar{1}$	$\bar{1}$	$\bar{1}$	14
1	$\bar{1}$	$\bar{1}$	0	15

removing half of the zeroth row from Table 2, leaving only half of a 0-trit in each column. After this truncation, the zeroth column of the table consists of repetitions of the block $0^{\frac{1}{2}}1\bar{1}0^{\frac{1}{2}}$; and, in general, the kth column consists of repetitions of the block $0^{3^k/2}1^{3^k}\bar{1}^{3^k}0^{3^k/2}$. This insight suggests that, instead of the obvious sum

$$\sum_{0\leqq i<n} \tau(i),$$

we should instead study $F(n)$ where

$$F(n)=\frac{\tau(0)}{2}+\tau(1)+\tau(2)+\cdots+\tau(n-1)+\frac{\tau(n)}{2}.$$

Of course, since $\tau(0)$ is actually zero, the first term here could be omitted, but this would obscure the structure of the truncation insight.

Happily enough, the function $F(n)$ succumbs straightforwardly to Delange's techniques. Each position in Table 2 has, in some sense, a one in three chance of containing the trit d. We define the function $t_k(n)$ to be the difference between the number of d-trits actually present in the first n positions of column k and the number of d-trits that we would expect to find there, namely $n/3$. Of course, by "the first n positions of column k," we are now referring to the entire entries in rows 1 through $n-1$ along with half of the entry in each of rows 0 and n. In symbols, we have

$$t_k(n)=\left(\frac{\tau_k(0)}{2}+\tau_k(1)+\tau_k(2)+\cdots+\tau_k(n-1)+\frac{\tau_k(n)}{2}\right)-\frac{n}{3}.$$

By our insight above into the structure of the truncated version of Table 2, we can express all of the functions $t_k(n)$ as rescalings of a single function $t(x)$, and we are in good shape.

In fact, this truncation technique works in a more general environment. Among all possible positional number systems with base q for $q\geqq 2$, the simplest are those whose digits form a sequence of q consecutive integers including 0. We will call such a system the $\langle q, r\rangle$ *number system*, where q denotes the base and r, in the range $0\leqq r\leqq q-1$, denotes the number of negative digits; that is, the digits in the $\langle q, r\rangle$ system are precisely the integers $-r, 1-r, 2-r, \ldots, q-1-r$. Note that standard base q representation is the $\langle q, 0\rangle$ system, and that balanced ternary is the $\langle 3, 1\rangle$ system.

These number systems come in symmetric pairs. The legal digits in the $\langle q, r\rangle$ system are exactly the negatives of the legal digits in the $\langle q, q-1-r\rangle$ system, and so representing n in one of these systems is equivalent to representing $-n$ in the other. This symmetry shows that we lose no essential generality by only considering the representations of nonnegative integers. Now, all of the digits of the $\langle q, q-1\rangle$ system are nonpositive, and hence, without the use of an explicit sign, this system can only represent the nonpositive integers. Therefore, we will exclude the case $r = q-1$ in what follows. On the other hand, every nonnegative integer has a unique representation in the $\langle q, r\rangle$ system for $0 \leqq r \leqq q-2$.

The truncation trick that we discussed in the balanced ternary case works in any $\langle q, r\rangle$ number system, and leads to the following Theorem. Since the proof of this Theorem is so similar to the proof of Theorem G, we hereby relegate any consideration of the details to Appendix P.

THEOREM P (The number of d-digits in the $\langle q, r\rangle$ positional number system). *Let q and r be integers satisfying $q \geqq 2$ and $0 \leqq r \leqq q-2$. Let the $\langle q, r\rangle$ number system be the positional number system with base q and digits $-r, 1-r, \ldots, q-1-r$, and let d be a nonzero digit in this system. Let $\rho(n)$ denote the number of times that the digit d is used when n is expressed in the $\langle q, r\rangle$ number system, and let $F(d, n)$ denote the appropriately truncated summation of ρ, in particular,*

$$F(d, n) = \left(1 - \frac{r}{q-1}\right)\rho(0) + \rho(1) + \rho(2) + \cdots + \rho(n-1) + \left(\frac{r}{q-1}\right)\rho(n).$$

Then, there exists a continuous, nowhere differentiable function P: $\mathbf{R} \to \mathbf{R}$, periodic with period 1, such that

$$F(d, n) = \frac{n \log_q n}{q} + nP(\log_q n) \quad \textit{for } n \geqq 1.$$

The Fourier series $P(x) = \sum_k p_k e^{2k\pi i x}$ of P converges absolutely. Finally, if we determine m by the relations $1 \leqq m \leqq q-1$ and $m \equiv d \pmod q$, and define the ξ and η by the formulas

$$\xi = \frac{m}{q} - \frac{r}{q(q-1)},$$

$$\eta = \frac{m+1}{q} - \frac{r}{q(q-1)},$$

the coefficients p_k are given by

$$p_0 = \log_q \Gamma(\xi) - \log_q \Gamma(\eta) - \frac{1}{q \ln q} - \frac{1}{2q}$$

$$p_k = \frac{\zeta(\chi_k, \xi) - \zeta(\chi_k, \eta)}{(\ln q)\chi_k(1+\chi_k)} \quad \textit{for } \chi_k = \frac{2k\pi i}{\ln q},\ k \neq 0.$$

This theorem has several corollaries. First, note that any linear combination

$$\sum_{\substack{-r \leqq d < q-r \\ d \neq 0}} \lambda_d F(d, n)$$

of the digit counts $F(d, n)$ can be evaluated by taking the corresponding linear combination of the results above. In particular, we can compute the sum of all of the digits used when the first n nonnegative integers are expressed in the standard base q system by computing the linear combination

$$\sum_{0<d<q} dF(d, n)$$

in the $\langle q, 0\rangle$ number system. The rather complex expressions that result for the relevant Fourier coefficients simplify pleasantly, and we arrive at the result that Delange actually demonstrated [3], of which our Theorem B was a special case. However, note that linear combinations of nowhere differentiable functions are not necessarily nowhere differentiable; hence the claim of nowhere differentiability in the following Corollary demands a separate proof, which Delange supplied.

COROLLARY S (H. Delange) (The sum of the digits in base q). *Let $q \geqq 2$ be an integer, and let $s(n)$ denote the sum of the digits used when n is expressed in the standard base q number system. Then, there exists a continuous, nowhere differentiable function S: $\mathbf{R} \to \mathbf{R}$, periodic with period* 1, *such that*

$$\sum_{0\leqq i<n} s(i) = \frac{q-1}{2} n \log_q n + nS(\log_q n) \quad \text{for } n \geqq 1.$$

Furthermore, the Fourier series $S(x) = \sum_k s_k e^{2k\pi ix}$ of S converges absolutely, and its coefficients s_k are given by

$$s_0 = \frac{(q-1)(\ln 2\pi - 1)}{2 \ln q} - \frac{q+1}{4}$$

$$s_k = \frac{(1-q)\zeta(\chi_k)}{(\ln q)\chi_k(1+\chi_k)} \quad \text{for } \chi_k = \frac{2k\pi i}{\ln q},\ k \neq 0.$$

It is fun to use Theorem P to compute the sum of the digits in balanced ternary, as well as in the standard base q systems discussed by Corollary S. Because of the symmetry of balanced ternary, the $(n \log n)$ term drops out completely, and the $k = 0$ coefficient of the Fourier series simplifies to $\log_3(2)$.

Theorem P can also be applied in some less obvious ways. The *negabinary* number system is the positional system with base -2 and digits 0 and 1; it is known that every integer has a unique representation in this system [7]. Theorem P does not apply immediately. But, if one groups adjacent pairs of digits in the negabinary system, and views the pairs as "super-digits," it is not hard to show that the resulting scheme is isomorphic to the $\langle 4, -2\rangle$ number system. The results of Theorem P can be carried over via this isomorphism.

In fact, Theorem G can almost be viewed as a corollary of Theorem P. Gray code corresponds in some sense to a $\langle 2, \frac{1}{2}\rangle$ number system, which a slight generalization of Theorem P can handle. We omit the details of this reduction, however.

4. The analysis of odd-even merging. The odd-even merge is the basic step of a sorting procedure due to Batcher [1]. The difficult part of the analysis of this algorithm is the study of the number of exchanges. Let B_n denote the average number of exchanges needed when two random files of size n are combined with Batcher's odd-even merge. As mentioned in the Introduction, Sedgewick managed to express B_n as a convolution of $\gamma(n)$ with binomial coefficients [11]. Although he did not explicitly mention Gray code, he showed that

$$B_n=\frac{n}{2}+2\sum_{k\geqq 1}F(k)\frac{\binom{2n}{n-k}}{\binom{2n}{n}}\quad \text{where } F(k)=\sum_{0\leqq i<k}\gamma(i).$$

Sedgewick then used the gamma function method to determine the asymptotic behavior of B_n. In this section, we will sketch an alternative derivation that uses Theorem G.

We begin our attack on B_n, as does Sedgewick, by limiting k to the range $|k|<\sqrt{n}\ln n$. In that range, Stirling's formula leads to the Gaussian approximation

$$\frac{\binom{2n}{n-k}}{\binom{2n}{n}}=e^{-k^2/n}\left(1+O\left(\frac{\log^4 n}{n}\right)\right).$$

Futhermore, when k exceeds $\sqrt{n}\ln n$, the left hand side will be exponentially small, in particular $O(\exp(-\log^2 n))$, so that we have

$$B_n=\frac{n}{2}+2\sum_{1\leqq k<\sqrt{n}\ln n}F(k)e^{-k^2/n}(1+O(n^{-1}\log^4 n)).$$

We now take advantage of Theorem G's information on the structure of $F(k)$, getting

$$B_n=\frac{n}{2}+2\sum_{1\leqq k<\sqrt{n}\ln n}\left(\frac{k\lg k}{2}+kG(\lg k)\right)e^{-k^2/n}(1+O(n^{-1}\log^4 n)),$$

where G is periodic with a known Fourier series. Splitting this sum into two parts, we have

$$\begin{aligned} B_n&=\frac{n}{2}+(C_n+2D_n)(1+O(n^{-1}\log^4 n)),\\ C_n&=\sum_{1\leqq k<\sqrt{n}\ln n}k\lg k\, e^{-k^2/n},\\ D_n&=\sum_{1\leqq k<\sqrt{n}\ln n}kG(\lg k)e^{-k^2/n}. \end{aligned}\tag{4.1}$$

For C_n, we can use the Euler–McLaurin summation formula in the form

$$\sum_{1\leqq k<m}f(k)=\int_1^m f(x)\,dx-\frac{f(m)-f(1)}{2}+\frac{f'(m)-f'(1)}{12}+O\left(\int_1^m |f''(x)|\,dx\right).$$

Putting $m=\sqrt{n}\ln n$ and $f(x)=x\lg x\, e^{-x^2/n}$, we can verify that $f''(x)=O(1)$ and thus that

$$C_n=\int_1^{\sqrt{n}\ln n}x\lg x\, e^{-x^2/n}\,dx+O(\sqrt{n}\log n).$$

Readjusting the limits of integration to 0 and ∞ only introduces an additional $O(1)$ error; performing the change of variables $y=x/\sqrt{n}$ and integrating, we have

$$\begin{aligned} C_n&=\frac{n\ln n}{2\ln 2}\int_0^\infty ye^{-y^2}\,dy+\frac{n}{\ln 2}\int_0^\infty y\ln y\, e^{-y^2}\,dy+O(\sqrt{n}\log n)\\ &=\frac{n\ln n}{2\ln 2}\left(\frac{1}{2}\right)+\frac{n}{\ln 2}\left(\frac{-\gamma}{4}\right)+O(\sqrt{n}\log n). \end{aligned}\tag{4.2}$$

We now turn to D_n. Unfortunately, the function $G(x)$ is not differentiable, so the Euler–McLaurin formula does not apply; we must resort to another method for determining the error incurred when replacing the sum by an integral. At least we can say on a term by term basis that

$$\left| f(k) - \int_k^{k+1} f(x)\,dx \right| \leqq \operatorname{osc}(f, [k, k+1])$$

where the oscillation of the function f on the interval I is defined by

$$\operatorname{osc}(f, I) = \sup_{x \in I} f(x) - \inf_{x \in I} f(x).$$

In the case of D_n, we have $f(x) = xG(\lg x)e^{-x^2/n}$ where we recall from § 2 that

$$G(x) = \frac{1-\{x\}}{2} + 2^{2-\{x\}} h(2^{\{x\}-2}),$$

$$h(x) = \sum_{j \geqq 0} 2^{-j} t(2^j x).$$

Beginning at the bottom and working up, note that $t(x)$ has maximum slope $\frac{1}{2}$, and hence satisfies the Lipschitz inequality

$$(4.3) \qquad |t(b) - t(a)| \leqq \tfrac{1}{2}|b-a|.$$

Now consider the oscillation of $h(x)$ on an interval of length $1/k$. If we split up the sum at the $\lfloor \lg k \rfloor$ term, we find that

$$h(x) = \sum_{0 \leqq j < \lfloor \lg k \rfloor} 2^{-j} t(2^j x) + 2^{-\lfloor \lg k \rfloor} h(2^{\lfloor \lg k \rfloor} x);$$

the summation here involves $\lfloor \lg k \rfloor$ terms, each of which satisfies the Lipschitz condition (4.3), while the final term is itself $O(1/k)$. Hence, we deduce that

$$\operatorname{osc}(h(x), I) = O\left(\frac{\log k}{k}\right) \quad \text{if } |I| = \frac{1}{k}.$$

Going from $h(x)$ to $G(x)$ only changes the constant factors involved; since the interval $[\lg k, \lg(k+1)]$ has length essentially $1/k$, we may then conclude that

$$\operatorname{osc}(G(x), [1\mathrm{g}\, k, \lg(k+1)]) = 0 = O\left(\frac{\log k}{k}\right).$$

Finally, returning to $f(x)$, we see that

$$\operatorname{osc}(f(x), [k, k+1]) = O(\log k),$$

and hence that

$$\left| D_n - \int_1^{\sqrt{n} \ln n} xG(\lg x)e^{-x^2/n}\,dx \right| \leqq \sum_{1 \leqq k < \sqrt{n} \ln n} O(\log k) = O(\sqrt{n} \log^2 n).$$

Again, we can change the limits of integration to 0 and ∞ with only a $O(1)$ error, and we deduce that

$$D_n = \int_0^\infty xG(\lg x)e^{-x^2/n}\,dx + O(\sqrt{n} \log^2 n).$$

To evaluate the integral, we substitute for G its Fourier series $G(x) = \sum_k g_k e^{2k\pi i x}$. Since this series converges absolutely, we may interchange summation and integration, getting

$$D_n = \sum_k g_k \int_0^\infty x^{1+2k\pi i/\ln 2} e^{-x^2/n}\, dx + O(\sqrt{n} \log^2 n).$$

Changing variables to $y = x^2/n$, we find that

$$\begin{aligned} D_n &= \frac{n}{2} \sum_k g_k e^{k\pi i \lg n} \int_0^\infty y^{k\pi i/\ln 2} e^{-y}\, dy + O(\sqrt{n} \log^2 n) \\ &= \frac{n}{2} \sum_k g_k e^{k\pi i \lg n} \Gamma(1 + k\pi i/\ln 2) + O(\sqrt{n} \log^2 n). \end{aligned} \tag{4.4}$$

Putting together (4.1), (4.2), and (4.4), we find that B_n satisfies

$$B_n = \frac{n \lg n}{4} + \left(\frac{1}{2} - \frac{\gamma}{4 \ln 2}\right) n + n \sum_k g_k \Gamma\left(1 + \frac{k\pi i}{\ln 2}\right) e^{2k\pi i(\log_4 n)} + O(\sqrt{n} \log^2 n). \tag{4.5}$$

If we define $M_1(x)$ to be the sum of the Fourier series

$$M_1(x) = \sum_k g_k \Gamma(1 + k\pi i/\ln 2) e^{2k\pi i x},$$

then the third term in (4.5) is just $M_1(\log_4 n)$. Furthermore, since the expression $\Gamma(1 + k\pi i/\ln 2)$ is exponentially small in k, this Fourier series converges extremely rapidly; the sum of such a series is analytic in a neighborhood of the real axis [14, pp. 161–162]. This gives us the following result, where $M(x)$ is $M_1(x)$ with its zeroth Fourier coefficient altered to absorb the linear second term in (4.5).

THEROEM M [11] (The average case exchange performance of Batcher's odd-even merge). *Let B_n denote the average number of exchanges that occur when two random files of size n are combined with Batcher's odd-even merge. There exists a function M: $\mathbf{R} \to \mathbf{R}$, analytic in a neighborhood of the real axis and periodic with period* 1, *such that*

$$B_n = \frac{n \lg n}{4} + nM(\log_4 n) + O(\sqrt{n} \log^2 n).$$

Furthermore, the coefficients m_k of the Fourier series $M(x) = \sum_k m_k e^{2k\pi i x}$ of M are given by

$$m_0 = 2 \lg \Gamma(\tfrac{1}{4}) - \lg \pi - \frac{\gamma + 2}{4 \ln 2} - \frac{3}{4} = 0.385417^+,$$

$$m_k = \frac{\Gamma(\chi_k/2)\zeta(\chi_k, \frac{1}{4})}{(\ln 2)(1 + \chi_k)} \quad \textit{for}\ \chi_k = \frac{2k\pi i}{\ln 2},\ k \neq 0.$$

Sedgewick's proof of this theorem shows that the error term can be reduced to $O(\sqrt{n} \log n)$; our rather heavy-handed oscillation argument accounts for the extra factor of $\log n$ in our result. On the other hand, the method that we have just employed seems somewhat more straightforward and direct than the gamma function method. Indeed, before turning to the gamma function method, Sedgewick first sketches out exactly our line of attack; he abandons it only for lack of information about the structure of the linear term in the expansion of $\sum_i \gamma(i)$.

Our method of proof also helps to give an insight into the source of the periodic term in the asymptotic expansion of B_n: it arises in the inherently periodic structure of the number of 1-bits in Gray code. Be warned however that this is just an insight, not a complete and intuitively satisfying explanation. In particular, the coefficient of the linear term has period 1 as a function of $\log_4 n$, not as a function of $\log_2 n$ as one might expect from this Gray code insight. This doubling of the period also occurs in the problem of the average number of registers R_n, discussed in the Introduction.

Recent results indicate that there is a fairly general correspondence between derivations like Sedgewick's, which use the gamma function method to compute a periodic term, and derivations like those in this note, which compute the same term by Fourier series techniques [5]. When viewed in an appropriate framework, these two kinds of derivations are revealed as more nearly equivalent than they appear at first glance.

Appendix. In this Appendix, we will skim over the highlights of the proof of Theorem P. The proof is very similar to the proof of Theorem G in § 2; the intent here is to concentrate on the differences. Thus, the reader should refer to § 2 for a fuller description of most of the following arguments. We will use the same notations for analogous concepts.

Recall the hypotheses of Theorem P. We begin the proof by considering the behavior of $\rho(n)$ and $F(d, n)$ on each digit position separately. Let $\rho_k(n)$ be 1 if the digit d appears in the kth position when n is written in the $\langle q, r\rangle$ number system, and 0 otherwise; also, define $F_k(d, n)$ by the truncated summation

$$F_k(d, n)=\left(1-\frac{r}{q-1}\right)\rho_k(0)+\rho_k(1)+\rho_k(2)+\cdots+\rho_k(n-1)+\left(\frac{r}{q-1}\right)\rho_k(n).$$

Since each position will contain the digit d with probability $1/q$ in some sense, we then define $t_k(n)$ by

$$t_k(n)=F_k(d, n)-\frac{n}{q}.$$

The functions $t_k(n)$ can all be expressed as rescalings of a single function $t\colon \mathbf{R}\to\mathbf{R}$ defined by

$$\text{(A.1)}\qquad t(x)=\begin{cases}-\dfrac{x}{q} & \text{if } 0\leqq x\leqq \xi\\[2mm] \left(\dfrac{q-1}{q}\right)x-\xi & \text{if } \xi\leqq x\leqq \eta\\[2mm] \dfrac{1}{q}-\dfrac{x}{q} & \text{if } \eta\leqq x\leqq 1\end{cases}\quad \text{and } t(x+1)=t(x)\quad \text{for all } x,$$

where ξ and η are as defined in the statement of the Theorem. In particular, we have

$$t_k(n)=q^{k+1}t(n/q^{k+1}).$$

We now let $l=\lfloor\log_q n\rfloor+1$, and deduce that

$$F(d, n)=\sum_{0\leqq k\leqq l}\left(\frac{n}{q}+t_k(n)\right)=\frac{(l+1)n}{q}+\sum_{0\leqq k\leqq l}q^{l-k+1}t(n/q^{l-k+1}).$$

Dropping the lower bound and replacing k by $(l-1-k)$, we get

$$\text{(A.2)}\qquad F(d, n)=\frac{(l+1)n}{q}+q^{l+1}h(n/q^{l+1}),$$

where h is the superposition function defined by

$$\text{(A.3)}\qquad h(x)=\sum_{k\geqq 0}\frac{t(q^kx)}{q^k}.$$

The function h is clearly periodic with period 1, and is continuous by the Weierstrass M-test. Furthermore, we can show that h is not differentiable at a point y by considering the behavior of its difference quotient over the intervals I_j for $j \geqq 1$, where I_j is the interval of the form

$$I_j = \left[\frac{p}{q^j} - \frac{r}{q^j(q-1)}, \frac{p+1}{q^j} - \frac{r}{q^j(q-1)}\right]$$

which contains y. Since

$$\frac{p}{q^j} - \frac{r}{q^j(q-1)} = \frac{pq-r}{q^{j+1}} - \frac{r}{q^{j+1}(q-1)},$$

the intervals I_j will be nested. The kth term in (A.3) will contribute nothing to the difference quotient if $k \geqq j$, since $t(q^k x)$ is periodic with period q^{-k}. For $0 \leqq k < j$, each term will contribute either $-1/q$ or $(q-1)/q$, since $t(q^k x)$ is linear over every interval I_j for $j > k$. Therefore, the difference quotient cannot converge, and h is nowhere differentiable.

Next, we define the function P: $\mathbf{R} \to \mathbf{R}$ by specifying that P is periodic with period 1 and satisfies the following identity for $0 \leqq x < 1$:

$$P(x) = \frac{2-x}{q} + q^{2-x} h(q^{x-2}).$$

Using the fact that $\xi > q^{-2}$, we can then check that this identity holds for all $x \leqq 1$. This means that P must also be continuous and nowhere differentiable.

The Fourier coefficients of $P(x)$ are then $p_k = c_k + d_k$, where

$$c_k = \int_0^1 \left(\frac{2-u}{q}\right) e^{-2k\pi i u}\, du = \begin{cases} 3/2q & \text{for } k = 0, \\ 1/2qk\pi i & \text{for } k \neq 0 \end{cases}$$

$$d_k = \int_0^1 2^{2-u} h(2^{u-2}) e^{-2k\pi i u}\, du.$$

Letting $\chi_k = 2k\pi i/\ln q$, we then have $d_k = H(1+\chi_k)/\ln q$ where

$$H(z) = \int_{q^{-2}}^{\infty} \frac{t(v)}{v^{z+1}}\, dv.$$

Noting that $t(v)$ can be written

$$t(v) = \int_0^v \left(\lfloor x+1-\xi \rfloor - \lfloor x+1-\eta \rfloor - \frac{1}{q}\right) dx,$$

we can integrate H by parts, and we have

$$H(z) = \frac{-q^{2(z-1)}}{qz} + \frac{1}{z} \int_{q^{-2}}^{\infty} \left(\lfloor v+1-\xi \rfloor - \lfloor v+1-\eta \rfloor - \frac{1}{q}\right) \frac{dv}{v^z}.$$

When $\Re(z) > 2$, we can split the integral into three terms, and we find that

$$H(z) = \frac{-q^{2(z-1})}{q(z-1)} + \frac{\zeta(z-1,\xi) - \zeta(z-1,\eta)}{z(z-1)}.$$

By analytic continuation, this formula also holds for $\Re(z) > 0$ if $z \neq 1$.

For $k \neq 0$, this is enough to compute d_k; in particular, we have

$$d_k = \left(\frac{-1}{2qk\pi i} + \frac{\zeta(\chi_k, \xi) - \zeta(\chi_k, \eta)}{(\ln q)\chi_k(1+\chi_k)}\right) \quad \text{for } k \neq 0.$$

With the aid of the classical expansion [14, p. 271]

$$\zeta(z-1, \alpha) = (\tfrac{1}{2} - \alpha) + \left(\ln \Gamma(\alpha) - \frac{\ln(2\pi)}{2}\right)(z-1) + O((z-1)^2),$$

we can calculate $H(1)$, and we deduce that

$$d_0 = \log_q \Gamma(\xi) - \log_q \Gamma(\eta) - \frac{1}{q \ln q} - \frac{2}{q}.$$

Combining the c_k with the d_k, we arrive at the values of p_k stated in the Theorem.

Finally, we observe that the p_k satisfy

$$p_k = O(|k|^{-3/2} \log |k|),$$

and hence the Fourier series of P converges absolutely. This completes the proof. □

Acknowledgment. The authors would like to thank Jean Vuillemin for assistance with an early version of this paper.

REFERENCES

[1] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Comp. Conf. (Montvale, NJ), 1968, pp. 307–314.

[2] N. G. DE BRUIJN, D. E. KNUTH AND S. O. RICE, *The average height of planted plane trees*, Graph Theory and Computing, R. C. Reed, ed., Academic Press, New York, 1972, pp. 15–22.

[3] H. DELANGE, *Sur la fonction sommatoire de la fonction somme des chiffres*, Enseignement Math., 21 (1975), pp. 31–47.

[4] P. FLAJOLET, J. C. RAOULT AND J. VUILLEMIN, *On the average number of registers required for evaluating arithmetic expressions*, Proc. 18th Symp. on Foundations of Computer Science (Providence, RI), 1977, pp. 196–205.

[5] L. GUIBAS, L. RAMSHAW AND R. SEDGEWICK, *Transform methods for evaluating certain combinatorial sums*, in preparation.

[6] R. KEMP, *The average number of registers needed to evaluate a binary tree optimally*, Saarbrücken University Report A 77104, Saarbrücken, 1977.

[7] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[8] ———, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[9] M. D. MCILROY, *The number of 1's in binary integers: bounds and extremal properties*, this Journal, 3 (1974), pp. 255–261.

[10] E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[11] R. SEDGEWICK, *Data movement in odd-even merging*, this Journal, 7 (1978), pp. 239–272.

[12] E. C. TITCHMARSH, *The Theory of Functions*, second edition, Oxford University Press, London, 1939.

[13] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–315.

[14] E. T. WHITTAKER AND G. N. WATSON, *A Course of Modern Analysis*, fourth edition, Cambridge University Press, London, 1927.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0015 $01.00/0

MONOIDS FOR RAPID DATA FLOW ANALYSIS*

BARRY K. ROSEN†

Abstract. Ambitious optimizing compilers have alertness and selectivity properties that lead to a serious discrepancy between the total cost of data flow analysis and the partial cost covered by the usual kind of theoretical estimate. This discrepancy motivates a new high-level analysis method for data flow problems expressible in terms of a semilattice L and monoid M of isotone maps from L to L, under algebraic constraints somewhat weaker than those imposed by Graham and Wegman in solving data flow problems on reducible graphs. The cost of the new method is roughly similar to that of the method of Graham and Wegman when estimates are made in the usual way, while the cost of updating in alert and selective compilers tends to be lower. The new method copes with arbitrary escapes and jumps, can find sharper information than fixpoint methods when M is not distributive, and can be tuned to trade time for sharpness of information.

Key words. data flow analysis, optimizing compilers, high-level languages, monoid, semilattice, structured programming

0. Introduction. In speaking of *ambitous compilers* we mean to include interactive program manipulation systems as well as noninteractive compilers that try to produce very efficient code. See [AS78], [Ca77], [CK76], [GRW77], [Har77a], [Har77b], [Kn74], [Lo77] for examples. This paper is primarily motivated by the serious discrepancy between the total cost of data flow analysis in ambitious compilers and the partial cost that is usually estimated in the theoretical literature. The issue has already been raised informally [Ro77b, pp. 713, 723]. A more precise discussion will be possible after relating syntactic structure to control flow in a precise and transparent way (despite possible deviations from "structured" programming), so we postpone this application to § 8. For introductory purposes we consider an important secondary motivation that can be understood with reference to the theoretical literature alone.

The earliest data flow analysis research dealt with concrete problems (such as detection of available expressions) and with low-level representations of control flow (with one large graph, each of whose nodes represents a basic block). Several recent papers have introduced an abstract approach, dealing with any problem expressible in terms of a semilattice L and a monoid M of isotone maps from L to L, under various algebraic constraints. Examples include [CC77], [GW76], [KU76], [Ki73], [Ta75], [Ta76], [We75]. Several other recent papers have introduced a high-level representation with many small graphs, each of which represents a small portion of the control flow information in a program. The hierarchy of small graphs is explicit in [Ro77a], [Ro77b] and implicit in papers that deal with syntax-directed analysis of programs written within the confines of classical structured programming [DDH, § I.7]. Examples include [TK76], [ZB74]. The abstract papers have retained the low-level representations while the high-level papers have retained the concrete problems of the earliest work. This paper studies abstract conditions on L and M that lead to rapid data flow analysis, with emphasis on high-level representations.

The "rapid" monoids introduced here are intuitively similar to the "fast" monoids introduced by Graham and Wegman [GW76]. Some important data flow monoids are rapid but not fast. Problems with rapid monoids can be solved by high-level methods. Under easily detected conditions that occur frequently in structured programming,

* Received by the editors March 15, 1978, and in revised form February 22, 1979. A condensation of an earlier version of this paper was presented at the 5th Annual ACM Symposium on Principles of Programming Languages, Tucson, January 1978.

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

high-level analysis can avoid computing with graphs altogether. Unlike some analysis methods oriented toward structured programming [TK76], [Wu75], [ZB74], our method retains the ability to cope with arbitrary escape and jump statements while it exploits the control flow information implicit in the parse tree. Another method with this ability is that of [BJ78a], [BJ78b], which is formulated for essentially the same concrete problems considered in [Ro77b].

The general algebraic framework for data flow analysis with semilattices is presented in § 1. Apart from the new concept of rapidity, this framework is more or less standard. The intuition is much as in cited parts of [GW76], but the formulation is significantly more general. We also introduce briefer and more mnemonic wording than in [GW76]. Section 2 introduces *flow covers,* which associate a family of formal expressions with a graph. Given a flow cover for a graph, we can solve many different flow problems by simply evaluating the expressions. (The values of the variables in the expressions are read off from the local and entry information specified by the problem.) Section 3 relates data flow problems to the hierarchies of small graphs introduced in [Ro77a], [Ro77b]. High-level analysis begins with local information expressed by mapping the arcs of a large graph into the monoid M, much as in low-level analysis. But each arc in our small graphs represents a set (often an infinite set) of paths in the underlying large graph. Appropriate members of M are associated with these arcs. This "globalized" local information is used to solve global flow problems in § 4. The fundamental theorem of § 4 is applied to programs with the control structures of classical structured programming in § 5. For a given rapid monoid M, the time required to solve any global data flow problem is linear in the number of statements in the program. (For varying M, the time is linear in the product of this number by $t_@$, where $t_@$ is a parameter of M introduced in the definition of rapidity.) For reasons sketched as the beginning of § 6, we feel obliged to cope with source level escape and jump statements as well as with classical structured programming. Section 6 shows how to apply the fundamental theorem of § 4 to programs with arbitrary escapes and jumps. The explicit time bound for programs with escapes but without jumps is roughly similar to the bound from [GW76, Thms. 4.2 and 5.4]. Examples of rapid monoids and a comparison between the results obtained by our method and those obtained by [GW76] are in § 7. Finally, § 8 lists conclusions and open problems, with emphasis on data flow analysis in ambitious compilers. The order of presentation here is based on the logical structure of the mathematical machinery. We proceed from the general to the particular, except in some places where bending the rule a little makes a significant improvement in the expository flow. Lemmas tend to appear as soon as their presuppositions have been established. One advantage of this organization is that it helps us avoid some unnatural and occasionally inadmissible assumptions that are common in the literature for historical reasons. One disadvantage is that a concept or a lemma without obvious motivation sometimes appears well before there is any useful work for it to do. We use appropriate literature citations or forward pointers in the text to mitigate this motivational disadvantage.

Apart from motivation, this paper is largely self-contained. Being self-contained often entails being long. (For various technical reasons, heavier reliance on the literature would have a high cost in readability for a slight decrease in length.) This work can be read as one long paper, but it can also be read as a paper of medium length sharing a bibliography with several short notes. In particular, § 1 is a note on the algebraic presuppositions of data flow analysis. It is not biased toward any one algorithm or family of algorithms. One could read § 1 and then pause to formulate one's favorite algorithm at § 1's level of generality before proceeding. Other sections whose ends are good places to pause are §§ 2, 4, and 6. The following glossary of symbols and concepts may be helpful in coping with the necessarily large amount of notation. Each long line in the glossary shows an important symbol, followed by the corresponding word or phrase and an indication of where to find a detailed explanation. Each long line is followed by a short line with a brief hint.

(L, M) algebraic context Sec. 1

L is a semilattice and M is a set of isotone maps $U: L \to L$.

G graph Sec. 1

G has node set $\mathbf{N}_G$, arc set $\mathbf{A}_G$, source and target maps $\mathbf{s}_G$, $\mathbf{t}_G$: $\mathbf{A}_G \to \mathbf{N}_G$.

$\mathcal{P} = (G, f, \mathbf{E}, E)$ global flow problem Def. 1.2

$\mathcal{P}$ has local information $f: \mathbf{A}_G \to M$, entry nodes $\mathbf{E} \subseteq \mathbf{N}_G$, and entry information $E: \mathbf{E} \to L$.

$I: \mathbf{N}_G \to L$ good solution to $\mathcal{P}$ Defs. 1.2, 1.3

Among the solutions to $\mathcal{P}$, I is good enough to dominate each fixpoint.

$@: M \times M \to M$ rapidity Def. 1.6

"Spiral" is a good way to pronounce @, which is written as an infix operator.

X, Y, Z flow expressions Def. 2.1

Operators $\wedge$, $\circ$, @ are formally applied to paths in graphs.

X, Y flow covers Def. 2.3

Flow expressions $X(m, n)$ are associated with certain pairs (m, n) of nodes.

$\pi \in \Sigma$ maximum π in poset Σ Secs. 3, 5

Especially, π may be a program and Σ may be partially ordered by nesting of statements.

$N\alpha$ and $A\alpha$ nodes and arcs contributed to G by α in Σ (3.1)

Each program statement contributes a set of nodes and a set of arcs to the control flow.

$\Pi(\alpha, n, p)$ path bit for statement α and nodes n, p (3.3)

$\Pi(\alpha, n, p)$ tells whether there is a path from'n to p within α.

$N_0\alpha$ new nodes in $N\alpha$ (3.5.2)

These nodes in $N\alpha$ are not in $N\beta$ for any $\beta < \alpha$.

$G\alpha$ induced graph for α (3.5)

The node set $NG\alpha$ has $N_0\alpha \subseteq NG\alpha \subseteq N\alpha$ and includes entrances and exits for parts of α.

$RAG\alpha$ real arcs in $G\alpha$ (3.5.3)

Arcs in α with sources and targets in $NG\alpha$ are also in $RAG\alpha$.

$IAG\alpha[\beta]$ imaginary arcs in $G\alpha$ due to β (3.5.4)

Arcs (n, p) with $\Pi(\beta, n, p) = 1$ are in $IAG\alpha[\beta]$.

$(G\alpha, \text{ENTR}\alpha)$ induced flow scheme Def. 3.6

High-level analysis uses a flow cover $X\alpha$ for each induced flow scheme.

$\mathcal{P}_0, \mathcal{P}[\beta]$ auxiliary problems (3.8)

A problem $\mathcal{P}$ can be solved by solving problems with smaller graphs.

$EG\alpha$ expected induced graph Sec. 5

The syntactic production that generates a statement α determines $EG\alpha$.

H effective height of L in (L, M) Def. 7.3

In the context (L, M), the only relevant strictly descending chains in L are at most H long.

$@_1, @_2$ loop products Lemma 7.4

These are possible choices for @ in Definition 1.6 when L has finite effective height.

1. Algebraic framework. Common mathematical notation is used, as when "is a subset of" is abbreviated by $\subseteq$ and "is in" is abbreviated by $\in$. The empty set is denoted $\varnothing$. To avoid excessive parentheses, the value of a function f at an argument x is fx rather than $f(x)$. If fx is itself a function then $(fx)y$ is the result of applying fx to y. The usual $\leqq$ and $\geqq$ symbols are used for arbitrary partial orders as well as for the usual order among integers. A function from a partially ordered set (*poset*) to a poset is *isotone* iff $x \leqq y$ implies $fx \leqq fy$. (Isotone maps are sometimes called "monotonic" in the literature.) A *meet semilattice* is a poset with a binary operation $\wedge$ such that $x \wedge y$ is the greatest lower bound of the set $\{x, y\}$. A meet semilattice wherein every subset has a greatest lower bound is *complete*. (Such a semilattice is also a complete lattice, a fact that is interesting but does not happen to be used in this paper.) The greatest lower bound of a set X will be denoted $\bigwedge X$. In particular, the empty subset has $\bigwedge\varnothing = \top_L$ in L, and $\top_L$ is then the maximum element in L. When L is clear from context we will write just $\top$ here. Note the distinction between our meet semilattices and the *join* semilattices of [Ro77b], [Ta75], [We75], where least upper bounds are considered instead of greatest lower bounds. To speak of meets is more natural in applications that are intuitively stated in terms of "what must happen on all paths" in some class of paths in a program, while to speak of joins is more natural in applications that are intuitively stated in terms of "what can happen on some paths." By checking whether there are any paths in the relevant class and by using the logical rule and $\exists$ is equivalent to $\neg\forall\neg$, join-oriented applications can be reduced to meet-oriented ones (and vice versa). A general theory should speak in one way or the other, and we have chosen meets. For us, strong assertions about a program's data flow are high in the semilattice. This is somewhat more common that the use of join semilattices in the literature.

A *monoid* is a set together with an associative binary operation $\circ$ that has a *unit element* $\mathbf{1}$ with $\mathbf{1} \circ m = m \circ \mathbf{1} = m$ for all m. In all our examples the monoid M will be a *monoid of functions*: every member of M is a function (from a set into itself), the operation $\circ$ is the usual composition $(U \circ V)x = U(Vx)$, and the unit $\mathbf{1}$ is the identity function with $\mathbf{1}x = x$ for all x. In addition to a complete semilattice L there may be given a monoid M of isotone maps $U: L \to L$, such that M is closed under pointwise meets: $U \wedge V$ is the map such that $(U \wedge V)x = Ux \wedge Vx$ for all x in L. This framework is a natural generalization of the "information propagation space" [GW76, p. 175], wherein L is the set of all subsets of a given finite set and $\leqq$ is set inclusion. With $U \leqq V$ iff $Ux \leqq Vx$ for all x in L, M is a meet semilattice with the pointwise meet operation. We will further assume that M has a maximum $\top_M$ which agrees with the maximum in the complete semilattice of *all* isotone maps on L: $\top_M x = \top_L$ for all x in L. We say that M is a *closed monoid of isotone maps* on L, and that the pair (L, M) is a *closed algebraic context* for data flow analysis. If we are given a complete meet semilattice L and a set M of isotone maps on L, with no guarantee of any other properties, then we say that (L, M) is an *algebraic context* for data flow analysis. The smallest monoid of functions containing M will be denoted M^*, with $M^* = M$ when the context is closed. Certain algebraic contexts that have $M^* = M$ but are not necessarily closed (because $U \wedge V$ need not be in M when U, V are in M) have been called "monotone data flow analysis frameworks" [KU77, § 3]. As will emerge shortly, the known data flow analysis algorithms fall into two classes. The algorithms that work in any monotone data flow analysis framework do not use the assumption $M^* = M$. The algorithms that use the assumption $M^* = M$ require a closed context as well but do not need some of the other assumptions imposed in [KU77, § 3].

Data flow problems arise when members of M are associated with the arcs of a directed graph. Only finite graphs will be considered here. Our view of graphs is more general than is usual in data flow research. A graph G consists of finite sets $\mathbf{N}_G$ (the *nodes*) and $\mathbf{A}_G$ (the *arcs*) together with maps $\mathbf{s}_G$ and $\mathbf{t}_G$ from $\mathbf{A}_G$ to $\mathbf{N}_G$. These are the *source* and *target* maps. In pictures, arcs are drawn as arrows from sources to targets. Of course the G subscripts are sometimes omitted if there is no doubt as to which graph is intended. A *path* is a finite sequence $\mathbf{c} = (c_1, \cdots, c_K)$ of arcs such that $\mathbf{t}c_k = \mathbf{s}c_{k+1}$ whenever $1 \leqq k < K$. The null sequence with $K = 0$ is allowed and is denoted λ. A nonnull path is from the node $\mathbf{s}c_1$ to the node $\mathbf{t}c_K$. We borrow map notation with $\mathbf{c}$: $\mathbf{s}c_1 \to \mathbf{t}c_K$ and speak of sources and targets for nonull paths as well as for arcs.

DEFINITION 1.1. Given a graph G, an algebraic context (L, M), and a map f: $\mathbf{A}_G \to M$, the *extension of* f *to map paths into* M^* is also denoted f and is defined by

$$f\mathbf{c} = \mathbf{1} \text{ if } \mathbf{c} = \lambda \quad \text{and} \quad f(c_1, \cdots, c_K) = fc_K \circ \cdots \circ fc_1 \quad \text{if } \mathbf{c} \neq \lambda.$$

When a program is represented by a graph G, we have a map f: $\mathbf{A}_G \to M$ with fc: $L \to L$ for each arc c. This local information tells how an "assertion" x in L associated with $\mathbf{s}c$ is propagated to $\mathbf{t}c$ as a transformed "assertion" $y = (fc)x$ when control flows along c. There is also a set $\mathbf{E} \subseteq \mathbf{N}_G$ of designated *entry* nodes and an initial assignment E: $\mathbf{E} \to L$ of "entry assertions," with En being "true" whenever control enters the program at n. Note that En need not be true whenever control reaches n from within the program itself. Knowing what is true at entry nodes at the time of entrance, we want to determine what is true at all nodes at all times.

DEFINITION 1.2. A *global flow problem* $\mathcal{P}$ is a quadruple $(G, f, \mathbf{E}, E)$, where G is a graph, f: $\mathbf{A}_G \to M$, $\mathbf{E} \subseteq \mathbf{N}_G$, and E: $\mathbf{E} \to L$. A *solution* for $\mathcal{P}$ is any I: $\mathbf{N}_G \to L$ such that, whenever m is in $\mathbf{E}$ and $\mathbf{c}$: $m \to n$ is a path in G, $In \leqq (f\mathbf{c})Em$.

Thus In asserts no more at n than can be propagated from m along $\mathbf{c}$. Because there may be other paths from entries to n, In may well assert much less than can be propagated along this one path. Definition 1.2 generalizes the definition of a "safe assignment" [GW76, p. 177] in several ways. We do not assume that $\mathbf{E}$ consists of a single node m and that all nodes in G are reachable from m. We do not assume that $Em = \bot$ for m in $\mathbf{E}$, where $\bot = \bigwedge L$ is the minimum element of L. These assumptions are only minor technical conveniences in [GW76], [KU76], [U173] and other places where they occur. Here they would be quite inconvenient. Of course any problem can be solved by letting $In = \perp$ for all n. This solution is uninteresting. A maximum solution would be ideal, but for some choices of M there can be no algorithm to find one [KU77, Thm. 7]. As in "acceptable assignments" [GW76, p. 177], we therefore consider solutions that are large enough to be interesting but that may be computable with a reasonable amount of effort.

DEFINITION 1.3. A *fixpoint* for a global flow problem $\mathcal{P} = (G, f, \mathbf{E}, E)$ is any $J\colon \mathbf{N}_G \to L$ such that, for all m in $\mathbf{E}$ and c in $\mathbf{A}_G$,

$$(1) \qquad Jm \leqq Em \text{ and } Jtc \leqq (fc)Jsc.$$

A *good* solution for $\mathcal{P}$ is any solution I such that, for each fixpoint J and each n in $\mathbf{N}_G$, $In \geqq Jn$.

Comparing Definition 1.3(1) with Definition 1.2, it is clear that any fixpoint is a solution. In many examples L is *well-founded*: there are no infinite descending chains. In this case, the maximum fixpoint can be found by beginning with the guess $In =$ (**if** n is in $\mathbf{E}$ **then** En **else** $\top_L$) and correcting I repeatedly, in light of Definition 1.3(1), until it stabilizes. Certain optimizations of this basic idea are pratical as data flow algorithms [KU76], [Ke75], [Ki73], [Ta76], [We75]. The only operation on M used by these *iterative* algorithms is **eval**: $M \times L \to L$ defined by **eval**$(U, x) = Ux$. Iterative algorithms always find a good solution when L is well-founded. Except in unusual cases where some nodes are not reachable from entry nodes, the good solution I_{iter} found by any iterative algorithm is the smallest one: the maximum fixpoint. It is well-known [Ki73, Thm. 2] that I_{iter} is the maximum solution whenever L is well-founded and the context is *distributive* [KU76, p. 160]:

$$(1.4) \qquad (\forall U \in M)(\forall x, y \in L)[U(x \wedge y) = Ux \wedge Uy].$$

Good solutions that are larger than I_{iter} can sometimes be found in nondistributive contexts. The algorithms that accomplish this have other advantages to justify their being more complicated than iteration.

Instead of assuming that L is well-founded, *elimination* algorithms [AC76], [GW76], [Ta75], [U173] assume that M is a closed monoid and use the composition and pointwise meet operations on M as well as **eval**: $M \times L \to L$. Our algorithm is in this family. These algorithms summarize the net effects on data flow information of certain sets of paths between certain pairs of nodes. The pairs of nodes and the set of paths considered for each pair are not the same for all elimination algorithms, but they all do presuppose a closed algebraic context. Composition of maps reveals the net effect of going from m to p by going from m to n and then from n to p. A meet of maps reveals the net effect of going from m to p in either of two ways, both of which have already been summarized. To deal with loops we do need to assume more than just a closed context. Somehow the effects of infinitely many paths must be summarized in finitely many steps by operations in M. To see how elimination algorithms accomplish this, it is helpful to consider a very simple example. Suppose an arc c runs from a node m to itself while an arc d runs from m to another node p. There is an infinite set $\Pi = \{(d), (c, d), (c, c, d), \cdots\}$ of paths from m to p. Let f maps arcs into M with $fc = U$ and $fd = V$. Then $f(\Pi) = \{f\mathbf{c} | \mathbf{c} \in \Pi\} = \{V, V \circ U, V \circ U^2, \cdots\}$ will in general be infinite. The following paragraph reviews the algebraic conditions that previous elimination algorithms have imposed upon M. By studying the consequences of successively weaker conditions, as they affect the task of summarizing $f(\Pi)$, we will be led to the new concept of rapidity.

The algebraic context (L, M) is *indempotent* iff each U in M is idempotent in the usual sense:

$$U \circ U = U. \tag{1.5.1}$$

For the illustrative set Π, if U is idempotent then $f(\Pi) = \{V, V \circ U\}$ and can be summarized by $V \wedge (V \circ U)$. Interval analysis [AC76], [U173] exploits idempotence. Because some important problems do not have idempotent contexts, weaker conditions must also be considered. The algebraic context (L, M) is *fast* iff each U in M is fast in the sense of [GW76, p. 175]:

$$U \circ U \geqq U \wedge \mathbf{1}. \tag{1.5.2}$$

For the illustrative set Π, $f(\Pi)$ does have a greatest lower bound $\bigwedge f(\Pi)$ in the complete semilattice $[L \rightarrow L]$ of *all* isotone maps from L to L. Computing in $[L \rightarrow L]$ under the assumption that U is fast, we find that $V \circ (U \wedge \mathbf{1}) \leqq \bigwedge f(\Pi)$ because $(U \wedge \mathbf{1}) \leqq U^r$ for all r. Sometimes the inequality is strict (as will be seen in § 7), but $V \circ (U \wedge \mathbf{1})$ is still adequate as a summary. Why? Consider any fixpoint J for a global data flow problem. Because $Jp \leqq VJm$ and $Jm \leqq UJm$, it follows that $Jp \leqq (V \circ (U \wedge \mathbf{1}))Jm$. Therefore J is also a fixpoint for a simpler problem with c and d replaced by an arc from m to p that has the local information $V \circ (U \wedge \mathbf{1})$ in M. Using $V \circ (U \wedge \mathbf{1})$ as a summary amounts to solving this simpler problem instead of the original one. If we can somehow find a good solution I for the simpler problem, then I will be above any fixpoint for the simpler problem and therefore will be above J. Even though $V \circ (U \wedge \mathbf{1})$ may be strictly less informative than the ideal $\bigwedge f(\Pi)$, it is good enough for getting above any fixpoint J. At least for the illustrative infinite set Π of paths, any W in M with $V \circ (U \wedge \mathbf{1}) \leqq W \leqq \bigwedge f(\Pi)$ will be an adequate summary. Logically, fastness is a weaker condition than idempotence. On the other hand, every fast context that arises in [GW76] is actually idempotent as well. There *are* important algebraic contexts that are not idempotent, and they are also not fast. For example, neither form of the algebraic context *CP* for constant propagation in [KU76, p. 167] is fast. To deal with such contexts we begin by recalling the trick used to deal with them in [GW76]. The *fastness closure* U^* of any U in M is the map

$$U^* = \bigwedge\{(U \wedge \mathbf{1})^r \mid r \in \mathbf{N}\}, \tag{1.5.3}$$

where $\mathbf{N}$ is the set of all natural numbers and $\bigwedge$ is taken in $[L \rightarrow L]$. (The original definition [GW76, p. 182] is equivalent to (1.5.3) for the contexts considered in [GW76].) The assumption that U^* is actually in M and can be found in finitely many steps, independent of the choice of u, is all that [GW76] really needs in order to cope with contexts that are not fast. For the illustrative set Π, we can reason just as above but with U^* instead of $(U \wedge \mathbf{1})$. We find that $V \circ U^* \leqq \bigwedge f(\Pi)$, and sometimes the inequality is strict, but $V \circ U^*$ is still adequate as a summary. At least for the illustrative infinite set Π of paths, any W in M with $V \circ U^* \leqq W \leqq \bigwedge f(\Pi)$ will be an adequate summary. More generally, the following definition considers contexts wherein certain expressions involving greatest lower bounds like (1.5.3) can be approximated within M.

DEFINITION 1.6. The algebraic context (L, M) is *rapid* iff there is given a binary operation @ on M and a positive integer $t_@$ such that, for all U, V in M,

$$V \circ U^* \leqq V \mathbin{@} U \leqq \bigwedge\{V \circ U^r \mid r \in \mathbf{N}\},$$

and $V \mathbin{@} U$ can be computed from V and U within at most $t_@$ steps, where any $\wedge$ or $\circ$ operation is counted as a single step. (When L is understood, M alone may also be called "rapid.")

In particular, we can define $V \mathbin{@} U$ to be $V \circ U^*$ whenever U^* is in M and we know how to find it in $t_@ - 1$ steps. Thus the contexts considered in [GW76] are all rapid.

(The converse fails, but the method of [GW76] is easily extended so as to work with arbitrary rapid closed contexts.) In rapid contexts where $V \circ U^* < V @ U$ we will sometimes be able to get sharper data flow information than in [GW76], as will be seen in § 7. A simple but important example of rapidity is provided by any indempotent context: let $V @ U = V \wedge (V \circ U)$ with $t_@ = 2$. The contexts for traditional global flow problems like available expressions [Ul73] are rapid for this reason. The definition of rapidity can be simplified in the important special case where L is well-founded and (1.4) holds. In this case (L, M) is also *strongly* distributive:

$$(1.7)\qquad (\forall U \in M)(\forall X \text{nonempty subset of } L)[U(\bigwedge X) = \bigwedge\{Ux | x \in X\}].$$

Under (1.7) it follows that $V \circ U^*$ is the only possible choice for $V @ U$ in Definition 1.6 because $V \circ U^* = \bigwedge\{V \circ U^r | r \in \mathbf{N}\}$. Therefore, a strongly distributive closed context is rapid iff each U^* is in M and can be found in a number of steps independent of the choice of U in M. Like [GW76], [Ta75], this paper does not assume (strong) distributivity. For us, the $\leqq$ signs in Definition 1.6 may be strict. Our algorithm is like other elimination algorithms in that it can only be applied to problems with rapid closed contexts. (For correctness alone, without concern for time bounds, elimination algorithms do not need $t_@$ in Definition 1.6.)

2. Flow schemes and flow covers. Instead of attacking a global flow problem directly, high-level analysis considers a hierarchy of problems posed with smaller graphs. The results in the rest of this section may seem wildly impractical to those used to thinking about graphs for entire programs, but we will only apply these results to the small auxiliary graphs in the hierarchy. Because G and $\mathbf{E}$ will vary in much smaller ranges than f and E as we move through the heirarchy of auxiliary problems, we will find it helpful to consider *global flow schemes*: pairs $(G, \mathbf{E})$ such that G is a graph and $\mathbf{E}$ is a set of nodes in G. Thus a scheme can be fleshed out to a problem by choosing an algebraic context (L, M) and then adding $f: \mathbf{A}_G \rightarrow M$ and $E: \mathbf{E} \rightarrow L$. We can simultaneously solve all the problems derived from a given scheme by working with formal expressions. Intuitively, an expression $X(m, n)$ represents all possible effects of propagating data flow information from m to n. The @ operation in rapid contexts lets us use finite expressions even when there are infinitely many paths from m to n.

DEFINITION 2.1. A *flow expression* for a given scheme $(G, \mathbf{E})$ is any formal expression X built with the set of operators $\{\wedge, \circ, @\}$, using paths in G as variables and a symbol $\top$ as a constant. Given a rapid closed algebraic context (L, M) and $f: \mathbf{A}_G \rightarrow M$, the *value* $[X:f]$ in M has

$$(1)\qquad [X:f] = f\mathbf{c} \text{ if } X \text{ is a path } \mathbf{c} \text{ and } [X:f] = \top_M \text{ if } X \text{ is } \top;$$

$$(2)\qquad [X:f] = [Y:f] \boxtimes [Z:f] \text{ if } X \text{ is } Y \boxtimes Z \text{ with } \boxtimes \text{ in } \{\wedge, \circ, @\}.$$

Now we want to assign a family of flow expressions to each scheme, so that any problem derived by fleshing out the scheme (in a rapid closed context) can be solved by evaluating expressions.

DEFINITION 2.2. A *solution* for a scheme $(G, \mathbf{E})$ is any map X, assigning a flow expression $X(m, n)$ to each (m, n) in $\mathbf{E} \times \mathbf{N}_G$, such that, whenever m is in $\mathbf{E}$ and $\mathbf{c}$: $m \rightarrow n$ is a path in G, each rapid closed context (L, M) has $[X(m, n):f] \leqq f\mathbf{c}$ for all $f: \mathbf{A}_G \rightarrow M$.

If only the members of some special class S of rapid closed contexts are of interest, we can consider the analogue of Definition 2.2 with (L, M) restricted to members of S.

With such a restriction, Definition 2.2 defines solutions *relative* to the special class S. The next definition and everything else in this paper can be similarly relativized, if desired.

DEFINITION 2.3. A *flow cover* for a scheme $(G, \mathbf{E})$ is any solution X such that, whenever $\mathcal{P}$ is a global flow problem $(G, f, \mathbf{E}, E)$ derived from $(G, \mathbf{E})$ in a rapid closed context and J is a fixpoint for $\mathcal{P}$, $Jn \leqq [X(m, n):f]Jm$ for all (m, n) in $\mathbf{E} \times \mathbf{N}_G$.

In the terminology of [Ta75, p. 9], the value $[X(m, n):f]$ is a "tag" for the triple (m, n, P), where P is the set of all paths from m to n in G. The net effect of using tags and "propagation sequences" [Ta75, p. 9] is like the net effect of using flow covers and the hierarchy of auxiliary problems. The technical realization of the similar intuitions behind [Ta75] is quite unlike what happens here.

LEMMA 2.4. *Let $\mathcal{P} = (G, f, \mathbf{E}, E)$ be a global flow problem and suppose $(G, \mathbf{E})$ has a flow cover X. Then a good solution I for $\mathcal{P}$ is obtained by setting, for each node n in G,*

$$In = \bigwedge\{[X(m, n):f]Em \mid m \text{ is in } \mathbf{E}\}.$$

Proof. Since $In \leqq (f\mathbf{c})Em$ whenever m is in $\mathbf{E}$ and $\mathbf{c}$: $m \to n$, I is a solution. Now consider any fixpoint J. Then $Jn \leqq [X(m, n):f]Jm \leqq [X(m, n):f]Em$ for all m in $\mathbf{E}$, so $Jn \leqq In$. □

To apply the above lemma efficiently in data flow analysis it will be helpful to know that good solutions to slightly perturbed problems are also good solutions to the original problems. The following lemma will be used in § 4.

LEMMA 2.5. *Let $\mathcal{P} = (G, f, \mathbf{E}, E)$ be a global flow problem with a good solution I. Let S be a set of nodes in G and let $\mathcal{P}_S$ be a problem exactly like $\mathcal{P}$, except that E_s: $\mathbf{E} \to L$ has $E_S m = Im$ for all m in $\mathbf{E} \cap S$. Any good solution for $\mathcal{P}_S$ is also a good solution for $\mathcal{P}$.*

Proof. Let I_S be a good solution for $\mathcal{P}_S$. In Definition 1.2 we find that I_S solves $\mathcal{P}$ because $E_S \leqq E$. In Definition 1.3(1) we find that any fixpoint J for $\mathcal{P}$ is also a fixpoint for $\mathcal{P}_S$ because $I \geqq J$ in Definition 1.3 for I and $\mathcal{P}$. By Definition 1.3 for I_S and $\mathcal{P}_S$, $I_S \geqq J$ and I_S is a good solution for $\mathcal{P}_S$. □

Before considering sufficient conditions for the existence and effective computability of flow covers, we must review some facts about cycles in graphs. A path $(c_1, \cdots, c_K)$ is *simple* iff $\mathbf{s}c_i \neq \mathbf{s}c_j$ whenever $i \neq j$. A nonnull path is a *cycle* iff $\mathbf{s}c_1 = \mathbf{t}c_K$. There are only finitely many simple cycles, and these can be enumerated. Most of the algorithms surveyed in [MD76] assume that arcs are pairs of nodes, but this assumption is easily avoided. Given a simple cycle $\psi = (c_1, \cdots, c_K)$ and a node n that appears in ψ, we consider a simple cycle $\phi = (\psi:n)$ that is like ψ but begins at n. Specifically, there is a unique i with $\mathbf{s}c_i = n$ and we have $\phi = (c_i, c_{i+1}, \cdots, c_K, c_1, \cdots, c_{i-1})$. The graph G is *monocyclic* iff it has a simple cycle ψ such that every other simple cycle is merely $(\psi:n)$ for some node n in ψ.

LEMMA 2.6. *If G is a monocyclic or acyclic, then any scheme $(G, \mathbf{E})$ has a flow cover.*

Proof. We deal with the monocyclic case first. Let ψ be as above, and let n have ψ: $n \to n$. Given m in $\mathbf{E}$ and p in $\mathbf{N}_G$, there are finitely many ways to choose a simple path μ: $m \to n$ and a simple path π: $n \to p$. Let $Y(m, p)$ be the result of formally $\wedge$-ing together all the corresponding expressions $(\pi @ \psi) \circ \mu$. Let $Z(m, p)$ be the result of formally $\wedge$-ing together all simple paths from m to p. Let $X(m, p)$be $Y(m, p) \wedge Z(m, p)$. (If there are no paths in the first place from m to p then $X(m, p)$ is taken to be $\top$. If there are paths but none that meet ψ, then $X(m, p)$ is taken to be $Z(m, p)$.) We must show that X is a flow cover. To show that X is a solution in Definition 2.2 we need $[X(m, p):f] \leqq f\mathbf{c}$ for any $\mathbf{c}$: $m \to p$ and f: $\mathbf{A}_G \to M$. This is trivial for simple paths, so we may assume $\mathbf{c}$ has the form $\mu \bullet \psi^r \bullet \pi$ (where $\bullet$ is concatenation of sequences of arcs) for some expression $(\pi @ \psi) \circ \mu$ in $Y(m, p)$. In Definition 1.6 we get

$$\begin{aligned}[X(m,p):f] &\leqq ([\pi:f] @ [\psi:f]) \circ [\mu:f]\\ &\leqq ([\pi:f] \circ [\psi:f]^r) \circ [\mu:f]\\ &= [(\mu \bullet \psi^r \bullet \pi):f] = f\mathbf{c}.\end{aligned}$$

To show that X is a flow cover in Definition 2.3 it will suffice to show that $Jp \leqq [Y(m,p):f]Jm$ for any fixpoint J, since $Jp \leqq [Z(m,p):f]Jm$ follows from $Jp \leqq (f\mathbf{c})Jm = [\mathbf{c}:f]Jm$ for each simple path $\mathbf{c}: m \to p$. Consider any expression $W = (\pi @ \psi) \circ \mu$ in $Y(m,p)$. By Definition 1.6, we can show $Jp \leqq [W:f]Jm$ by showing

$$Jp \leqq ([\pi:f] \circ [\psi:f]^* \circ [\mu:f])Jm. \tag{1}$$

By (1.5.3) and $Jn \leqq [\psi:f]Jn \wedge Jn$, $Jn \leqq [\psi:f]^*Jn$. Therefore

$$Jp \leqq [\pi:f]Jn \leqq [\pi:f][\psi:f]^*Jn \leqq [\pi:f][\psi:f]^*[\mu:f]Jm$$

and (1) holds. Of course a scheme with an acyclic graph has a flow cover by the obvious specialization of the above proof. □

A graph that is not monocyclic or acyclic is *polycyclic*. Auxiliary schemes with polycyclic graphs will occasionally arise in high-level analysis, so we need to cope with them. But to cope with them efficiently is not very urgent. Polycyclic auxiliary schemes do not arise at all in analyzing the very broad class of programs considered in § 5. The following lemma formally implies Lemma 2.6, but the proof uses a more complex algorithm that yields different flow covers, less convenient for § 5. The problem of optimizing this algorithm or replacing it altogether, so as to find flow covers efficiently for polycyclic flow schemes, can be left open here. To save space we forego some easy optimizations.

LEMMA 2.7. *Any scheme* $(G, \mathbf{E})$ *has a flow cover.*

Proof. A flow cover for $(G, \mathbf{E})$ can be derived by restricting one for $(G, \mathbf{N}_G)$, so we may assume $\mathbf{E} = \mathbf{N}_G$. We use induction on the number of arcs in G. If G has no arcs then let

$$X(m,n) = (\textbf{if } m = n \textbf{ then } \lambda \textbf{ else } \top).$$

Now suppose that G has $r+1$ arcs and choose an arc c. By the induction hypothesis (for graphs with r arcs) there is a flow cover Y for the scheme $(H, \mathbf{E})$, where H is the result of removing the arc c from G. The definition of $X(m,n)$ in terms of $Y(m,n)$ uses auxiliary expressions

$$\begin{aligned}R_1(m,n) &= Y(\mathbf{t}c, n) \circ (c) \circ Y(m, \mathbf{s}c);\\ R_2(m,n) &= (Y(\mathbf{t}c, n) @ [(c) \circ Y(\mathbf{t}c, \mathbf{s}c)]) \circ (c) \circ Y(m, \mathbf{s}c);\\ R_3(m,n) &= (Y(\mathbf{s}c, n) @ [Y(\mathbf{t}c, \mathbf{s}c) \circ (c)]) \circ Y(m, \mathbf{s}c).\end{aligned}$$

Let pH^*q iff there is a path from p to q in H. Then

```
X(m, n) = if mH*sc
              then if tcH*sc
                       then Y(m, n) ∧ R2(m, n) ∧ R3(m, n)
                       else Y(m, n) ∧ R1(m, n)
              else Y(m, n).
```

The verification that X is a flow cover is very much like the corresponding verification for Lemma 2.6, although the notation is a little more complex. In the case where $\mathbf{s}c$ is reachable from both m and $\mathbf{t}c$, $Y(m, n)$ represents the paths from m to n without c. Paths with c of the form

$$(m \to \mathbf{s}c) \bullet (c) \bullet [(\mathbf{t}c \to \mathbf{s}c) \bullet (c) \bullet \cdots \bullet (\mathbf{t}c \to \mathbf{s}c) \bullet (c)] \bullet (\mathbf{t}c \to n)$$

are represented by $R_2(m, n)$, while paths of the form

$$(m \to \mathbf{s}c) \bullet [(c) \bullet (\mathbf{t}c \to \mathbf{s}c) \bullet \cdots \bullet (c) \bullet (\mathbf{t}c \to \mathbf{s}c)] \bullet (\mathbf{s}c \to n)$$

are represented by $R_3(m, n)$. □

The proof of the above lemma adjusts a given flow cover to allow for added arcs. Allowing for deleted arcs is much easier and will be useful in § 6. Before stating the lemma for deleted arcs, it will be helpful to note some rules for simplifying flow expressions. A rule $\Theta \to \Omega$ is *valid* in a rapid closed context (L, M) iff, whenever X is a flow expression for a flow scheme $(G, \mathbf{E})$ and Y is the result of replacing a subexpression of the form Θ by the corresponding subexpression of the form Ω, then $[X:f] = [Y:f]$ for any f: $\mathbf{A}_G \to M$. There are three rules for expressions containing the null path λ. These rules are valid in all rapid closed contexts because $[\lambda:f] = \mathbf{1}$ in Definition 2.1(1) and Definition 1.1.

(2.8) $$\lambda \circ Z \to Z \quad \text{and} \quad Z \circ \lambda \to Z \quad \text{and} \quad Z @ \lambda \to Z.$$

With one exception, the six rules (2.9) for simplifying expressions containing $\top$ are valid in all rapid closed contexts.

(2.9.1) $$\top \wedge Z \to Z \quad \text{and} \quad Z \wedge \top \to Z;$$

(2.9.2) $$\top \circ Z \to \top \quad \text{and} \quad [Z \circ \top \to \top];$$

(2.9.3) $$\top @ Z \to \top \quad \text{and} \quad Z @ \top \to Z.$$

The bracketed rule in (2.9.2) is only valid in contexts with $U\top_L = \top_L$ for all U in M. To use the bracketed rule is to consider flow covers relative to the special class S of rapid closed contexts with this property. Any rapid closed context (L, M) can be transformed to one in S. First we add a new maximum $\top_{\text{new}}$ to L, forming $L' = L \cup \{\top_{\text{new}}\}$. Then we extend each U in M to U': $L' \to L'$ with $U'\top_{\text{new}} = \top_{\text{new}}$. The rapid closed context (L', M') with $M' = \{U' | U \in M\}$ is in S.

LEMMA 2.10. *Let X be a flow cover for $(G, \mathbf{E})$ and let H be the result of deleting some arcs from G. For all (m, n) in $\mathbf{E} \times \mathbf{N}_H$, let $Y(m, n)$ be derived from $X(m, n)$ in two stages. First, replace each path* $\mathbf{c}$ *with $K > 0$ occurrences of deleted arcs by the corresponding expression* $\mathbf{e}_K \circ \top \circ \mathbf{e}_{K-1} \circ \cdots \circ \mathbf{e}_1 \circ \top \circ \mathbf{e}_0$, *where* $\mathbf{c}$ *has the form* $\mathbf{e}_0 \bullet (d_1) \bullet \mathbf{e}_1 \bullet \cdots \bullet \mathbf{e}_{K-1} \bullet (d_K) \bullet \mathbf{e}_K$ *for deleted arcs* $d_1, \cdots, d_K$. *Second, simplify according to the rules* (2.8) *and* (2.9). *Then Y is a flow cover for* $(H, \mathbf{E})$.

Proof. Given any f: $\mathbf{A}_H \to M$ we can specify f': $\mathbf{A}_G \to M$ by letting $f'c = \top_M$ if c is in $\mathbf{A}_G - \mathbf{A}_H$. It now follows by induction on subexpressions that $[X(m, n):f'] = [Y(m, n):f]$ for all m in $\mathbf{E}$ and all n in $\mathbf{N}_G = \mathbf{N}_H$. This implies that Y is a solution because X is a solution. Moreover, any fixpoint for a problem $(H, f, \mathbf{E}, E)$ derived from $(H, \mathbf{E})$ is a fixpoint for the corresponding problem $(G, f', \mathbf{E}, E)$, so Y is a flow cover because X is a flow cover. □

3. Hierarchies of graphs. In [Ro77a] the control flow in a program is represented by a hierarchy of small graphs rather than by one large graph alone. The relevant abstract definitions from [Ro77a, § 2] will be restated here for ease of reference. The definitions list those properties of the particular hierarchy considered in [Ro77b] that are exploited there. By stating the properties abstractly we can deal with other important hierarchies. For example, the definitions and lemmas of this section let us generalize interval analysis as formulated in [AC76] to work with arbitrary rapid closed contexts. Doing this work abstractly lets us generalize [AC76] and [Ro77b] simultaneously, a fact that helps justify the tedium of this section. The reader may wish to skim this material causally, then return to it as necessary when later sections refer to details.

As in [Ro77a], [Ro77b], it is convenient to assume that no two arcs have the same sources and targets, so that the arc c may be identified with the pair $(\mathbf{s}c, \mathbf{t}c)$ of nodes. A *nesting structure* for a graph G is a finite partially ordered set Σ with a maximum π, together with a set $N\alpha$ of nodes and a set $A\alpha$ of arcs for each α in Σ. The following properties are required:

(3.1.1) $$N\pi = \mathbf{N}_G \quad \text{and} \quad A\pi = \mathbf{A}_G;$$

(3.1.2) $$\beta \leqq \alpha \quad \text{in } \Sigma \quad \text{implies} \quad (N\beta \subseteq N\alpha \text{ and } A\beta \subseteq A\alpha);$$

(3.1.3) $$(c \text{ in } \mathbf{A}_G \text{ has } \mathbf{s}c, \mathbf{t}c \text{ in } N\alpha) \quad \text{implies} \quad (c \text{ is in } A\alpha).$$

The use of "implies" rather than "iff" in (3.1.3) is deliberate. Escapes and jumps in programs will introduce arcs with c in $A\alpha$ but with $\mathbf{t}c$ not in $N\alpha$. Also deliberate is the weakness of (3.1.2). For the moment there is no need to require all the properties connoted by "nesting." A nesting structure *with entrances and exits* consists of G and Σ as in (3.1) together with, for each α in Σ, sets of *designated entrances* and *designated exits*.

(3.2.1) $$\text{DENTR}\alpha \subseteq N\alpha \quad \text{and} \quad \text{DEXIT}\alpha \subseteq N\alpha$$

such that, whenever $\beta < \alpha$ in Σ,

(3.2.2) $$\text{DENTR}\alpha \cap N\beta \subseteq \text{DENTR}\beta \quad \text{and} \quad \text{DEXIT}\alpha \cap N\beta \subseteq \text{DEXIT}\beta.$$

Entrance and exit sets are defined by

(3.2.3) $$\text{ENTR}\alpha = \text{DENTR}\alpha \cup \{\mathbf{t}c \in N\alpha \mid c \in \mathbf{A}_G \;\&\; \neg\, \mathbf{s}c \in N\alpha\};$$

(3.2.4) $$\text{EXIT}\alpha = \text{DEXIT}\alpha \cup \{\mathbf{s}c \in N\alpha \mid c \in \mathbf{A}_G \;\&\; \neg\, \mathbf{t}c \in N\alpha\}.$$

Given a nesting structure with entrances and exists, consider any α in Σ, n in ENTRα, and p in EXITα. Let $\Pi(\alpha, n, p) = 1$ if

(3.3) there is a path from n to p in G touching only nodes of $N\alpha$

and $\Pi(\alpha, n, p) = 0$ otherwise. These *path bits* can be computed bottom-up, beginning with choices of α that are minimal in Σ. Path bits for α can be determined from previously computed path bits for *parts* β of α, where

(3.4) $$\text{PART}\alpha = \{\beta < \alpha \mid \text{No } \gamma \text{ in } \Sigma \text{ has } \beta < \gamma < \alpha\}.$$

As in [Ro77b, § 4], we construct the *induced graph* $G\alpha$ for each α in Σ. The set of nodes is

(3.5.1) $$NG\alpha = N_0\alpha \cup \bigcup_{\beta \in \text{PART}\alpha} (\text{ENTR}\beta \cup \text{EXIT}\beta),$$

where

(3.5.2) $$N_0\alpha = N\alpha - \bigcup_{\beta \in \text{PART}\alpha} N\beta.$$

The set of *real* arcs of $G\alpha$ is

(3.5.3) $$\mathrm{RAG}\alpha = \{c \in A\alpha \mid \mathbf{s}c, \mathbf{t}c \in NG\alpha\}.$$

For each β in PARTα there is a set of *imaginary* arcs

(3.5.4) $$IAG\alpha[\beta] = \{(n, p) \mid n \in \mathrm{ENTR}\beta \;\&\; p \in \mathrm{EXIT}\beta \;\&\; \Pi(\beta, n, p) = 1\}.$$

The total sets of arcs is

(3.5.5) $$AG\alpha = RAG\alpha \cup \bigcup_{\beta \in \mathrm{PART}\alpha} IAG\alpha[\beta]$$

with the obvious definitions of sources and targets. (Some arcs are both real and imaginary.)

Examples of these definitions can be constructed by letting Σ by any family of subsets of $\mathbf{N}_G$ that includes $\mathbf{N}_G$ itself, with set inclusion for $\leqq$. In particular, the regions [U173, p. 200] defined by interval analysis of control flow may be used. The induced graphs are then quite similar to the intervals that occur in various places in the reduction sequence leading from G to a single node graph. See [AC76] and the works cited there for details on interval analysis. In the example of most interest here, Σ is the set of all ⟨statement⟩ nodes in the parse tree of a high-level program with an ALGOL-like syntax. Details are in §§ 5 and 6. Here the hierarchy of induced graphs can be constructed directly from the parse tree and symbol table. Under the circumstances considered below, data flow analysis can deal only with induced graphs.

Definition 3.6. A nesting structure with entrances and exits is *locally covered* by assigning a flow cover to each *induced flow scheme* ($G\alpha$, ENTRα).

One can solve any global flow problem $\mathcal{P} = (G, f, \mathbf{E}, E)$ without computing on G, provided that G has a locally covered nesting structure and $\mathbf{E} = \mathrm{DENTR}\pi$. The algorithm does compute with Σ and the induced graphs, and it uses a given flow cover $X\alpha$ for each induced scheme ($G\alpha$, ENTRα). In § 5 we will exploit structured programming to optimize away most of the work with induced graphs. The algorithm requires some preprocessing that globalizes the local information in f: $\mathbf{A}_G \to M$. Instead of asking only what must happen when control flows along an arc of G, we ask what must happen when control flows along any of the paths summarized by an arc in an induced graph $G\alpha$. For each α in Σ there are maps

(3.7.1) $$f_\alpha : AG\alpha \to M \quad \text{and} \quad F_\alpha : \mathrm{ENTR}\alpha \times NG\alpha \to M$$

such that, for all c in $AG\alpha$ and all (m, p) in ENTR$\alpha \times NG\alpha$,

(3.7.2) $$f_\alpha c = (\textbf{if } c \in RAG\alpha \textbf{ then } fc \textbf{ else } \top_M) \wedge \bigwedge\{F_\beta c \mid \beta \in \mathrm{PART}\alpha \;\&\; c \in IAG\alpha[\beta]\};$$

(3.7.3) $$F_\alpha(m, p) = [X\alpha(m, p) : f_\alpha].$$

If α is minimal in Σ then $IAG\alpha = \varnothing$ and (3.7.2) defines f_α to agree with f. If α is nonminimal then f_α still agrees with f on real arcs that are not imaginary, but on imaginary arcs f_α is determined by the various F_β maps for β in PARTα. These will be available when we try to compute f_α, provided we begin at the bottom in a linearization of the partial order on Σ.

Now consider the maximum π in Σ. Because $\mathbf{E} = \mathrm{DENTR}\pi \subseteq NG\pi$, there is a global flow problem

(3.8.1) $$\mathcal{P}_0 = (G\pi, f_\pi, \mathbf{E}, E),$$

and Lemma 2.4 yields

(3.8.2) $$I_0 : NG\pi \to L, \quad \text{a good solution to } \mathcal{P}_o.$$

For each part β of π we can now specify a global flow problem

(3.8.3)
$$\mathcal{P}[\beta] = (G[\beta], f[\beta], \mathbf{E}[\beta], E[\beta])$$

where $G[\beta]$ is the graph whose set of nodes is $N\beta$ and whose set of arcs is $A\beta \cap (N\beta \times N\beta)$, with sources and targets as in G. (This is not the induced graph $G\beta$ and will generally be much larger.) The map $f[\beta]$ is the restriction of f to arcs in $G[\beta]$. For entry information we use

(3.8.4)
$$\mathbf{E}[\beta] = \mathrm{ENTR}\beta \quad \text{and} \quad E[\beta] = (I_0 \text{ restricted to } \mathbf{E}[\beta]),$$

so that $\mathcal{P}[\beta]$ depends on the choice of I_0 in (3.8.2). As will be seen in detail shortly, $G[\beta]$ has a nesting structure with $\Sigma[\beta] = \{\alpha \text{ in } \Sigma | \alpha \leqq \beta\}$, so that induction on the size of Σ should lead to good solutions $I[\beta]$ of $\mathcal{P}[\beta]$ for each β in PARTπ. Combining these with I_0 should yield a good solution to the original problem $\mathcal{P}$. Several technical points must be checked. Because M may not be distributive, we do not have anything as simple as the induced graph theorem [Ro77b, § 4] to establish the relevance of the auxiliary problems $\mathcal{P}_0$ and $\mathcal{P}[\beta]$ to the original problem $\mathcal{P}$. The first step is to note that $\mathcal{P}[\beta]$ does indeed inherit a nesting structure, with induced maps (3.7) as well as entrances and exits, from $\mathcal{P}$.

LEMMA 3.9. *For any part β of π, the graph $G' = G[\beta]$ in* (3.8.3) *has a nesting structure with*

(1)
$$\Sigma' = \{\alpha \textit{ in } \Sigma | \alpha \leqq \beta\} \quad \textit{with } |\Sigma'| < |\Sigma|.$$

Moreover, for all $\alpha \leqq \beta$,

(2)
$$N'\alpha = N\alpha \quad \textit{and} \quad A'\alpha = A\alpha \cap (N\beta \times N\beta).$$

This becomes a nesting structure with entrances and exits after designating

(3)
$$\mathrm{DENTR}'\alpha = \mathrm{ENTR}\alpha \quad \text{and} \quad \mathrm{DEXIT}'\alpha = \mathrm{EXIT}\alpha,$$

such that the induced graph $G'\alpha$ for each α in Σ' is the same as the induced graph $G\alpha$ for α considered in Σ. Moreover, the construction (3.7) *applied to $\mathcal{P}' = \mathcal{P}[\beta]$ yields the same induced maps as* (3.7) *applied to $\mathcal{P}$*:

(4)
$$f'_\alpha = f_\alpha \quad \textit{and} \quad F'_\alpha = F_\alpha \quad \textit{for all } \alpha \leqq \beta.$$

Proof. The proof is direct from the definitions. □

The above lemma justifies induction on $|\Sigma|$ in proving the following three lemmas on the relevance of the auxiliary problems.

LEMMA 3.10. *If J is a fixpoint for $\mathcal{P}$ then the restriction J_0 of J to nodes in $NG\pi$ is a fixpoint for $\mathcal{P}_0$.*

Proof. We may assume the assertion holds for problems $\mathcal{P}'$ with posets Σ' such that $|\Sigma'| < |\Sigma|$. For any β in PARTπ, the restriction J' of J to $N\beta$ is a fixpoint of $\mathcal{P}' = \mathcal{P}[\beta]$. By Lemma 3.9 and the induction hypothesis, the restriction J'_0 of J' to $NG'\beta$ is a fixpoint of the problem $\mathcal{P}'_0$ defined by (3.8.1) applied to $\mathcal{P}'$. Now consider any path $\mathbf{c}$: $m \to p$ in $G\beta$, where m is in ENTRβ and p is in EXITβ. We get

$$Jp \leqq [X\beta(m, p) : f_\beta]Jm = F_\beta(m, p)Jm.$$

Therefore any imaginary arc (m, p) in $G\pi$ has

$$Jp \leqq \bigwedge\{F_\beta(m, p)Jm | (m, p) \text{ in } IAG\pi[\beta]\}$$

and $Jp \leqq [f(m, p)]Jm$ if (m, p) is also real, so that $Jp \leqq [f_\pi(m, p)]Jm$. Any real arc (m, p)

in $G\pi$ that is not imaginary has $Jp \leqq [f(m, p)]Jm = [f_\pi(m, p)]Jm$ also, so J_0 is fixpoint of $\mathcal{P}_0$. □

LEMMA 3.11. *Suppose β is in* PARTπ *and* $\mathbf{c}$: $m \to p$ *in* $G[\beta]$, *where m is in* ENTRβ *and p is in* EXITβ. *Then* $F_\beta(m, p) \leqq f\mathbf{c}$.

Proof. The proof is by induction on $|\Sigma|$. □

LEMMA 3.12. *Extend I_0 from* (3.8.2) *to have domain* $\mathbf{N}_G$ *by setting* $I_0 n = \top_L$ *whenever n is not in $NG\pi$. Suppose that $I[\beta]$ is a good solution to $\mathcal{P}[\beta]$ for each β in* PARTπ. *Then* I: $\mathbf{N}_G \to L$ *defined by*

(1) $$In = I_0 n \wedge \bigwedge\{I[\beta]n \mid \beta \in \text{PART}\pi \ \& \ n \in N\beta\}$$

is a good solution to $\mathcal{P}$.

Proof. To show that I is a solution to $\mathcal{P}$, consider any path $\mathbf{c}$: $m \to n$ where m is in $\mathbf{E}$. We must show that $In \leqq (f\mathbf{c})Em$. The path $\mathbf{c}$ can be parsed as a concatenation of (possibly null) sequences of arcs:

$$\mathbf{c} = \mathbf{c}_0 \bullet \mathbf{d}_1 \bullet \mathbf{c}_1 \bullet \cdots \bullet \mathbf{d}_K \bullet \mathbf{c}_K,$$

where each $\mathbf{c}_k$ is a sequence of arcs in $G\pi$ and each $\mathbf{d}_k$ is a sequence of arcs in G that are not in $G\pi$. For each $\mathbf{d}_k$ there is β in PARTπ such that $\mathbf{d}_k$ is a nonnull path in β with a source in ENTRβ. If the target of $\mathbf{d}_k$ is not in EXITβ then we have $k = K$ and $\mathbf{c}_K = \lambda$ and n is the target. Each arc in $\mathbf{c}_k$ is real but may also be imaginary.

Case 1 [$\mathbf{c}_K \neq \lambda$]. For each k with $1 \leqq k \leqq K$ there is an imaginary arc in $G\pi$ from the source of $\mathbf{d}_k$ to the target of $\mathbf{d}_k$. Let $\mathbf{i}_k$ be the path in $G\pi$ consisting of just this arc, so that $\mathbf{c}_0 \bullet \mathbf{i}_1 \bullet \cdots \bullet \mathbf{i}_K \bullet \mathbf{c}_K$ is path from m to n in $G\pi$. By (3.8.2) and (1),

(2) $$In \leqq I_0 n \leqq [(f_\pi \mathbf{c}_K) \circ (f_\pi \mathbf{i}_K) \circ \cdots \circ (f_\pi \mathbf{i}_1) \circ (f_\pi \mathbf{c}_0)]Em.$$

Let (s, t) be the arc in $\mathbf{i}_k$. Then (3.7.2) and Lemma 3.11 imply

(3) $$f_\pi \mathbf{i}_k = \bigwedge\{F_\beta(s, t) \mid \beta \in \text{PART}\pi \ \& \ (s, t) \in IAG\pi[\beta]\} \leqq f\mathbf{d}_k.$$

But $f_\pi c \leqq fc$ for all c in $RAG\pi$, so $f_\pi \mathbf{c}_k \leqq f\mathbf{c}_k$. Applying this and (3) to (2) yields $In \leqq (f\mathbf{c})Em$.

Case 2 [$\mathbf{c}_K = \lambda$]. If $K = 0$ then $m = n$ and $In \leqq I_0 m \leqq Em = (f\mathbf{c})Em$ because $\mathbf{c}$ is null. We may assume $K > 0$ and $\mathbf{c}_{K-1} \neq \lambda$. Let p be the source of $\mathbf{d}_K$ and the target of $\mathbf{c}_{K-1}$. Then the Case 1 reasoning applies to the prefix of $\mathbf{c}$ leading from m to p, so (2) implies

(4) $$I_0 p \leqq [(f\mathbf{c}_{K-1}) \circ (f\mathbf{d}_{K-1}) \circ \cdots \circ (f\mathbf{d}_1) \circ (f\mathbf{c}_0)]Em.$$

Now p is in ENTRβ for some part β of π, and $I_0 p = E_\beta p$ in the subproblem $\mathcal{P}[\beta]$. The path $\mathbf{d}_K$ from p to n is a path in $G[\beta]$, so

$$In \leqq I[\beta]n \leqq (f[\beta]\mathbf{d}_K)E_\beta p = (f\mathbf{d}_K)I_0 p$$

because $I[\beta]$ solves $\mathcal{P}[\beta]$. By (4) and isotonicity of $f\mathbf{d}_K$, this yields what is wanted:

$$In \leqq [(f\mathbf{d}_K) \circ (f\mathbf{c}_{K-1}) \circ (f\mathbf{d}_{K-1}) \circ \cdots \circ (f\mathbf{d}_1) \circ (f\mathbf{c}_0)]Em.$$

To show that I is a good solution we must show $I \geqq J$ for any fixpoint J. If n is in $G\pi$ then $I_0 n \geqq Jn$ because I_0 is a good solution for $\mathcal{P}_0$ and Lemma 3.10 says J on $NG\pi$ is a fixpoint for $\mathcal{P}_0$. If n is in $N\beta$ for β in PARTπ then $I[\beta] \geqq Jn$ because J on the nodes of $G[\beta]$ is a fixpoint for $\mathcal{P}[\beta]$. Therefore (1) implies $In \geqq Jn$ because In is a *greatest* lower bound. □

4. High-level data flow analysis. Throughout this section we suppose that a rapid closed context (L, M) is given. Flow covers could be relative to any class of contexts that includes (L, M). Given a global flow problem $\mathcal{P} = (G, f, \mathbf{E}, E)$ with $\mathbf{E} = \mathrm{DENTR}\pi$ such that G is locally covered by a flow cover $X\alpha$ for each induced scheme $(G\alpha, \mathrm{ENTR}\alpha)$ we can use the results of the previous section to find a good solution for $\mathcal{P}$. We begin by computing the globalized local information of (3.7). Descending through Σ recursively, we solve the auxiliary problems of (3.8) and combine the results as in Lemma 3.12(1). Lemmas 3.9 and 3.12 lead to a correctness proof by induction on $|\Sigma|$. But how much does all this cost? Can the method be optimized to exploit regularities in well-structured programs? To address such questions we program the algorithm more formally, but still at a very high level. We also impose an additional condition on the nesting structure: for all α, β in Σ,

$$(4.1) \qquad \neg(\alpha \leqq \beta \text{ or } \beta \leqq \alpha) \quad \text{implies} \quad N\alpha \cap N\beta = \varnothing.$$

This is true in all the examples of interest here, and it greatly simplifies the analysis of computational complexity. A global flow problem satisfying all the conditions imposed so far is said to be *well-nested.* (These conditions are reviewed in stating Theorem 4.4 below.)

The following procedure RECURSOLVE takes as argument some γ in Σ. The well-nested problem $\mathcal{P}$ is accessed by RECURSOLVE as a global variable. The purpose of calling RECURSOLVE is to have.a side effect on I, a global variable whose values are maps from the nodes of G into L. The effect of **call** RECURSOLVE(γ) is to change In for each n in $N\gamma$, so that I on this set of nodes becomes a solution to the problem $\mathcal{P}' = \mathcal{P}[\gamma]$ defined as in (3.8.3) but with arbitrary γ in Σ in place of β restricted to be a part of π. To achieve this effect, RECURSOLVE begins by calling LOCALSOLVE to find a good solution for $\mathcal{P}'_0$ defined as in (3.8.1) but with γ in place of π. (The procedures do not construct $\mathcal{P}'$ and $\mathcal{P}'_0$, but we do in the correctness proof.)

```
RECURSOLVE: proc ($\gamma$: member of the poset $\Sigma$)
    [  call LOCALSOLVE($\gamma$);
       #On nodes in $G\gamma$, $I$ is now a good solution to $\mathcal{P}'_0$ for $\mathcal{P}' = \mathcal{P}[\gamma]$.#
       for all $\beta$ in PART$\gamma$ do call RECURSOLVE($\beta$)
    ]
```

Note that RECURSOLVE does not explicitly combine the solutions to the subproblems $\mathcal{P}'[\beta]$ as might be expected from Lemma 3.12(1). We have already begun to optimize the method in light of (4.1). Using ○○○ as a placeholder for implicit procedure bodies (such as the body of RECURSOLVE above), we write the program SOLVE that gets $\mathcal{P}$ as input and puts out a good solution I.

```
SOLVE:
    [  dcl $\mathcal{P}$ well-nested global flow problem;
       dcl $I$ map from $\mathbf{N}_G$ into $L$;
       dcl $f_\alpha$ map from $AG\alpha$ into $M$ for all $\alpha$ in $\Sigma$;
       dcl $F_\alpha$ map from ENTR$\alpha \times NG\alpha$ into $M$ for all $\alpha$ in $\Sigma$;
       dcl LOCALMAPS proc ○○○;
       #This procedure sets up $f_\alpha$ and $F_\alpha$ for each $\alpha$ in $\Sigma$ as in (3.7).#
       dcl LOCALSOLVE proc($\gamma$: member of $\Sigma$) ○○○;
       #This procedure finds a good solution as in Lemma 2.4.#
       dcl RECURSOLVE proc($\gamma$: member of $\Sigma$) ○○○;
       get $\mathcal{P}$; call LOCALMAPS;
    for all $n$ in $\mathbf{N}_G$ do
        if $n \in \mathbf{E}$ then $In \leftarrow En$ else $In \leftarrow \top_L$;
    call RECURSOLVE($\pi$); put $I$
]
```

For LOCALMAPS we assume an arbitrary listing ($\delta = \alpha_1, \alpha_2, \cdots, \alpha_{|\Sigma|} = \pi$) of Σ, such that $\alpha_i \leqq \alpha_j$ implies $i \leqq j$. Thus a variable ranging over Σ can be stepped from δ to π like an integer variable.

```
LOCALMAPS: proc
  [ dcl $\alpha$ member of $\Sigma$;
    for $\alpha$ from $\delta$ to $\pi$ do
       [ for all $c$ in $AG\alpha$ do
              [ if $c \in RAG\alpha$ then $f_\alpha c \leftarrow fc$ else $f_\alpha c \leftarrow \top_M$;
                if $c \in IAG\alpha$ then $f_\alpha c \leftarrow f_\alpha c \wedge F_\beta c$
                     where $c \in IAG\alpha[\beta]$ for unique $\beta$
                     #Uniqueness follows from (4.1).#
              ];
         for all $(m, p)$ in $ENTR\alpha \times NG\alpha$ do
              $F_\alpha(m, p) \leftarrow [X\alpha(m, p) : f_\alpha]$
       ]
  ]
```

Finally, LOCALSOLVE is as suggested by Lemma 2.4.

```
LOCALSOLVE: proc ($\gamma$: member of $\Sigma$)
  [ for all $n$ in $NG\gamma$do
        $In \leftarrow \bigwedge\{[F_\gamma(m, n)]Im | m \in \mathrm{ENTR}\gamma\}$
  ]
```

The very high-level iterator **for all** n **in** $NG\gamma$ **do** $\cdots$ can be defined in several ways, especially if multiprocessors are available to execute $In \leftarrow \cdots$ and $In' \leftarrow \cdots$ for $n \neq n'$ in parallel. To show that LOCALSOLVE does find a good solution to $\mathcal{P}'_0$ for $\mathcal{P}' = \mathcal{P}[\gamma]$, we prove the following lemmas without needing to choose among the various reasonable definitions of iterators.

LEMMA 4.2. *Let* EITHERSOLVE *stand for* RECURSOLVE *or* LOCALSOLVE. *For each α in Σ,* EITHERSOLVE *is called with actual parameter α exactly once in the course of* SOLVE'*s computation. When control passes through* **call** EITHERSOLVE(α) *in* SOLVE, *any node n such that In changes must be in $N\alpha$.*

Proof. The first assertion follows from (4.1). The second assertion is verified by induction on $\leqq$ in Σ. If α is such that everything below α in Σ has the desired property, then α has the desired property. □

LEMMA 4.3. *Whenever control reaches* **call** LOCALSOLVE(γ) *in* SOLVE, *each m in* ENTRγ *has $Im = E[\gamma]m$. Whenever control leaves* **call** LOCALSOLVE(γ) *in* SOLVE, *I restricted to $NG\gamma$ is a good solution to $\mathcal{P}[\gamma]$.*

Proof. Suppose for the moment that any α which satisfies the first assertion must satisfy the second one. Then both assertions can be proved inductively as follows. First, note that when π is γ we have $\mathbf{E} = \mathrm{DENTR}\pi = \mathrm{ENTR}\pi$ and each m in $\mathbf{E}$ has $Im = E[\pi]m = Em$ by the initialization. Therefore π satisfies the first assertion. Second, consider any α in Σ that satisfies the first assertion. To continue the induction we show that each part β of α satisfies the first assertion. By our supposition and (3.8), each part β does have $Im = E[\beta]m$ for each m in ENTRβ when control leaves **call**

LOCALSOLVE(γ) with α as γ. By Lemma 4.2 and (4.1), this equation remains true when control reaches **call** LOCALSOLVE(γ) with β as γ.

To prove our supposition, let α satisfy the first assertion. The second assertion will follow immediately from Lemma 2.4 *if* the very high-level iterator is defined by independent evaluations of all the $\bigwedge$ expressions followed by simultaneous assignments to In for all n. We must show that a good solution is also obtained even when the iterator is defined by stepping through $NG\gamma$ in some arbitrary order, assigning values as soon as they are found, so that Im might have changed by the time $[F_\gamma(m, n)]Im$ is evaluated for the sake of assigning to In. This is easily done by an inductive argument with the aid of Lemma 2.5. □

To study the correctness and computational complexity of SOLVE it will be appropriate to assume that $\wedge$ and $\circ$ always require "one" step while application of a member of M to a member of L also takes "one" step. See [GW76, p. 178] for more on the reasonableness of this assumption. For the time bound (but not for the correctness assertion) in the following theorem we assume that each iterator **for all** VAR **in** SET is defined by stepping VAR through SET in some fixed order.

THEOREM 4.4. *Let $\mathcal{P} = (G, f, \mathbf{E}, E)$ be a well-nested global flow problem: there is given a nesting structure for G with entrances and exits such that* $\mathbf{E} = \text{DENTR}\pi$ *and* (4.1) *holds, and there is a given flow cover $X\alpha$ for each induced global flow scheme* ($G\alpha$, ENTRα). *Then* SOLVE *with input $\mathcal{P}$ finishes with output I such that I is a good solution for $\mathcal{P}$. Moreover, let all iterators be defined by stepping through their index sets. Then the time required* (*apart from input/output*) *is*

$$O(T\alpha_1 + T\alpha_2 + \cdots + T\alpha_{|\Sigma|} + |\mathbf{N}_G|), \tag{1}$$

where $(\alpha_1, \cdots, \alpha_{|\Sigma|})$ *is a listing of* Σ *and each* α *in* Σ *has*

$$T\alpha = |AG\alpha| + (S\alpha)_1 + \cdots + (S\alpha)_{\text{pairs}(\alpha)} + \text{pairs}(\alpha), \tag{2}$$

where pairs (α) *is* |ENTR$\alpha \times NG\alpha$| *and each* $(S\alpha)_j$ *is the time required to evaluate* $[X\alpha(m, p){:}f_\alpha]$ *by* LOCALMAPS *for the j-th pair* (m, p) *in* ENTR$\alpha \times NG\alpha$.

Proof. Correctness follows from Lemmas 3.9, 3.12, 4.2, and 4.3 by induction on the depth of recursion in RECURSOLVE. The time required by LOCALMAPS for each α is $O(|AG\alpha| + (S\alpha)_1 + \cdots + (S\alpha)_{\text{pairs}(\alpha)})$, while the time required by the call on LOCALSOLVE with $\gamma = \alpha$ is $O(\text{pairs}(\alpha))$. By Lemma 4.2, the calls on LOCALMAPS and RECURSOLVE in SOLVE require time $O(T\alpha_1 + \cdots + T\alpha_{|\Sigma|})$. Finally, the $|\mathbf{N}_G|$ term comes from initializing I. □

In practice the above time bound is not so elaborate as it looks. Global flow problems do not appear spontaneously. They arise when compilers attempt to optimize programs, and programs are written in programming languages. By studying the control structures in a language, we can derive sharp estimates on the parameters in the time bound. Detailed analysis must wait for more definitions in the next section.

5. Control structures: an important special case. The input to a compiler is a program, not a global flow problem. While constructing a problem from a program for the sake of optimization, a compiler can also construct a nesting structure with entrances and exits. When the graph and nesting structure are chosen in the way described in [Ro77a], [Ro77b], the relation between the parse tree and the hierarchy of small graphs is so transparent that most calculations with graphs can be optimized away. (Here we are optimizing the compiler itself, before using it to optimize programs.) For purposes of this section, a program is always a single statement: we use ALGOL-like syntax. Most types of statement are *complex*: built up from smaller statements by control operators like **if** · · · **then** · · · **else** · · · or **while** · · · **do** · · · . The *simple* statements contain no smaller statements. This section explains how G is constructed, but it is quite terse. See any one of [Ro76], [Ro77a], [Ro77b] for a more leisurely discussion with examples. We deliberately confuse a statement node in the parse tree with the corresponding fragment of program text. The set Σ of all statements in a program is partially ordered by $\beta \leqq \alpha$ iff β is a descendant of α in the parse tree. The maximum π in Σ is the whole program, while the minimal elements δ are the simple statements. The graph G and the hierarchy of induced graphs can both be constructed in a bottom-up pass through the parse tree. The induced graphs are as in (3.5). For G itself, each statement contributes nodes and arcs given by

(5.1.1) $$N\alpha = N_0\alpha \cup \bigcup_{\beta<\alpha} N\beta \quad \text{and} \quad A\alpha = A_0\alpha \cup \bigcup_{\beta<\alpha} A\beta,$$

where the *new* nodes $N_0\alpha$ and arcs $A_0\alpha$ are determined by the control operator used to form α when α is complex. We will allow control operators like **case** that take a varying number of arguments, and then a phrase like "the control operator used to form" should be interpreted as "the control operator and number of arguments used to form." For all statements considered here, $N_0\alpha$ has two distinguished nodes *entering* α and *leaving* α, and the designated entrances and exits of (3.2) are given by

(5.1.2) $$\text{DENTR}\alpha = \{\textit{entering}\ \alpha\} \quad \text{and} \quad \text{DEXIT}\alpha = \{\textit{leaving}\ \alpha\}.$$

The three operators emphasized in [Ro77b] contribute no new nodes beyond *entering* α and *leaving* α to G. They are

α: **if** · · · **then** β_1 **else** β_2	(conditional statements)
α: **while** · · · **do** β	(**while** statements)
α: $[\beta_1; \cdots; \beta_K]$	(sequential compound statements)

Figure 5.1 is a more concise version of Figs. 1–3 in [Ro77b], giving names to the arcs in $A_0\alpha$ and indicating their sources and targets, for each of these three control operators. The arcs named q or q_k for integers k are *not* in $A_0\alpha$. They are used to display how we expect the induced graph $G\alpha$ to look when α is a conditional or a **while** or a sequential compound statement. Conditions under which these expectations are fulfilled will be studied shortly, but first we consider four more control operators. They are

α: **if** · · · **then** β	(one-part conditional statements)
α: **case** · · · **of** $\beta_1; \cdots; \beta_K$ **esac**	(**case** statements)
α: **do** β **until** · · ·	(**until** statements)
α: **for** · · · **from** · · · **to** · · · **by** · · · **do** β	(stepped iteration statements)

With varying punctuation and choices of keywords, the **until** and stepped iteration statements are very widespread. The concrete syntax above reflects the author's liking for short mnemonic keywords and clean ALGOL-like punctuation. These two statements add new nodes *testing* α as well as *entering* α and *leaving* α to G. (The **while-until** statement from [WFSW75] would require two new nodes *testing*1 α and *testing*2 α.) The new arcs are as indicated in Figs. 5.2. As in Fig. 5.1, the arcs named q or q_k are not in A_0 but are expected to be in the induced graph $G\alpha$. In contrast to the pictures commonly drawn in discussions of structured programming or graph grammars for generating classes of flowcharts [DDH72, § I.7], [FKZ76], [LM75], [WFSW75], Figs. 5.1 and 5.2 have no sourceless or targetless arcs and no unstated conventions for joining graphs along such dangling arcs. The advantages of our greater explicitness become apparent when control structures other than those of classical structured programming are diagrammed, as in [Ro77a]. The seven control operators considered so far will be

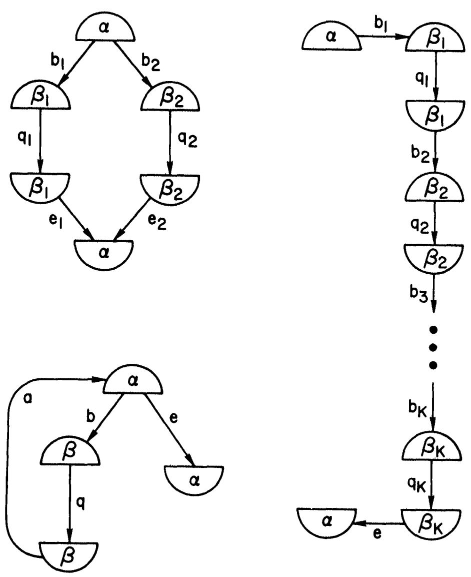


FIG. 5.1. *The node* entering *α appears as an upper half circle marked α. The node* leaving *α appears as a lower half circle marked α. Conditional,* **while**, *and sequential compound statements are illustrated.*

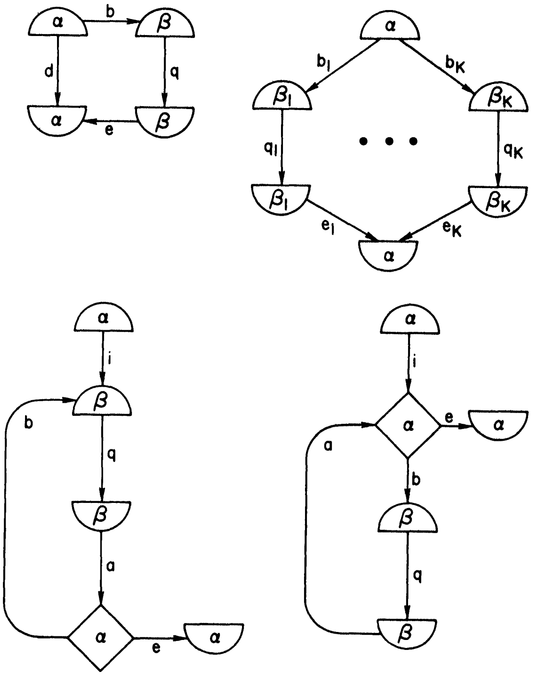


FIG. 5.2. *The node* testing *α appears as a diamond marked α. One-part conditional,* **case**, **until**, *and stepped iteration statements are illustrated.*

collectively called *CSP* operators (for classical structured programming). In addition to (5.1), the important properties of *CSP* operators are as follows. Associated with each statement α built from a *CSP* operator is a graph $EG\alpha$, the *expected* induced graph for α, that is determined (apart from the names of the nodes and arcs) by the operator used to form α. Thus $EG\alpha_1$ and $EG\alpha_2$ for any two **while** statements α_1 and α_2 are as in Fig.

5.1, but with α_1 and α_2 in place of α and with names of the loop bodies of α_1 and α_2 in place of β. Apart from renaming and variations in the number K of statement arguments taken by each control operator like **case** or sequential composition, there are only finitely many expected induced graphs, one for each *CSP* operator in our language, and each such graph *EG* has

(5.2.1) *EG* is monocyclic or acyclic;

(5.2.2) $|\mathbf{N}_{EG}|$ and $|\mathbf{A}_{EG}|$ are $O(K)$;

(5.2.3) there is a path from *entering* α to *leaving* α in *EG*;

(5.2.4) no arc has source *leaving* α in *EG*.

The linear functions of K in (5.2.2) will change if the family of CSP operators is changed. We have omitted the more elaborate loop building operators like **while-until** because the escape statements in the next section are simpler and more powerful means to the same ends. The importance of (5.2.4) will emerge in Lemma 6.6.

To compose structured programs in the classical sense of [DDH72, § I.7], one builds all complex statements with CSP operators. Each simple statement has no effect on the flow of control: control enters the statement, something happens, and control leaves the statement. This motivates the following definition.

DEFINITION 5.3. The expected induced graph $EG\delta$ for a simple statement δ has two nodes *entering* δ and *leaving* δ and one arc (*entering* δ, *leaving* δ). A simple statement δ is *classical* iff the nodes and arcs contributed by δ to G are given by

(1) $$N\delta = \{\mathit{entering}\ \delta, \mathit{leaving}\ \delta\};$$

(2) $$A\delta = \{(\mathit{entering}\ \delta, \mathit{leaving}\ \delta)\}.$$

A program is *classical* iff every simple statement in it is classical and every complex statement is built by a CSP operator.

LEMMA 5.4. *Let G be the graph for a classical program. Then each statement* α *has*

$$\mathrm{ENTR}\alpha = \{\mathit{entering}\ \alpha\} \quad \mathit{and} \quad \mathrm{EXIT}\alpha = \{\mathit{leaving}\ \alpha\} \quad \mathit{and} \quad G\alpha = EG\alpha.$$

Proof. The assertion holds if α is simple by Definition 5.3. Otherwise α is complex and the assertion may be assumed to hold for its parts. By (5.1) and (5.2.3), the assertion holds for α. □

We can now do much of the work of data flow analysis for classical programs at the time of language definition, long before any specific programs are compiled. With each *expected flow scheme* ($EG\alpha$, {*entering* α}) we can associate an *expected flow cover* $EX\alpha$ as soon as pictures like those in Fig. 5.1 and Fig. 5.2 are available. By Lemma 5.4, $EX\alpha$ will indeed be a flow cover for the actual induced scheme ($G\alpha$, ENTRα) when we encounter a specific statement α in a specific program. By Theorem 4.4, SOLVE can find a good solution to a global flow problem by moving through the parse tree and evaluating formal expressions tabulated by the expected flow covers. Having fixed the set of *CSP* operators, we can estimate the costs of evaluating the expressions rather sharply. The elaborate time bound in Theorem 4.4 will reduce to the assertion that the cost of data flow analysis is $O(|\Sigma| \cdot t_{@})$. The expected flow covers are derived from those constructed in Lemma 2.6 by elementary calculations. Since m is always *entering* α in $EX\alpha(m, p)$, it will be convenient to write ☆ rather than *entering* α.

LEMMA 5.5. *Let* α *be a simple statement. Then* $EX\alpha$ *is a flow cover for the expected flow scheme, where* $EX\alpha(☆, ☆) = \lambda$ *and* $EX\alpha(☆, \mathit{leaving}\ \alpha) = (c)$ *for* $c = (☆, \mathit{leaving}\ \alpha)$.

LEMMA 5.6. *Let α be a one-part conditional statement* **if** $\cdots$ **then** β. *Then $EX\alpha$ is a flow cover for the expected flow scheme, where $EX\alpha(☆, ☆) = \lambda$ and (for b, q, d, e as in Fig. 5.2)*

$$EX\alpha(☆, \textit{entering } \beta) = (b) \quad \textit{and} \quad EX\alpha(☆, \textit{leaving } \beta) = (b, q);$$

$$EX\alpha(☆, \textit{leaving } \alpha) = [(e) \circ (b, q)] \wedge (d).$$

In most kinds of data flow analysis we know that $f: \mathbf{A}_G \to M$ will have $fe = \mathbf{1}$: nothing *happens* when control flows from *leaving* β to *leaving* α. Indeed, the usual low-level control flow graphs do not even distinguish these two nodes. For such kinds of analysis the above flow cover can be optimized by using (b, q) instead of $(e) \circ (b, q)$. For available expressions it is also known that $fb = fd = \mathbf{1}$, so $EX\alpha(☆, \textit{leaving } \alpha) = (q) \wedge \lambda$ would suffice. As above, we will only display the most general form for each flow cover. For each specific kind of problem, such as available expressions, the implementor can optimize the expected flow cover in light of whatever is known about the maps from arcs into M that can actually arise from programs.

LEMMA 5.7. *Let α be a two-part conditional or* **case** *statement with parts $\beta_1, \cdots, \beta_K$. Then $EX\alpha$ is a flow cover for the expected flow scheme, where $EX\alpha(☆, ☆) = \lambda$ and*

$$EX\alpha(☆, \textit{entering } \beta_k) = (b_k) \quad \textit{and} \quad EX\alpha(☆, \textit{leaving } \beta_k) = (b_k, q_k) \quad \textit{for } k = 1, \cdots, K;$$

$$EX\alpha(☆, \textit{leaving } \alpha) = [(e_1) \circ (b_1, q_1)] \wedge \cdots \wedge [(e_K) \circ (b_K, q_K)].$$

LEMMA 5.8. *Let α be a sequential compound statement* $[\beta_1; \cdots; \beta_K]$. *Then $EX\alpha$ is a flow cover for the expected flow scheme, where*

$$EX\alpha(☆, ☆) = \lambda \quad \textit{and} \quad EX\alpha(☆, \textit{entering } \beta_1) = (b_1);$$

$$EX\alpha(☆, \textit{leaving } \beta_k) = (q_k) \circ EX\alpha(☆, \textit{entering } \beta_k) \quad \textit{for } k = 1, \cdots, K;$$

$$EX\alpha(☆, \textit{entering } \beta_k) = (b_k) \circ EX\alpha(☆, \textit{leaving } \beta_{k-1}) \quad \textit{for } k = 2, \cdots, K;$$

$$EX\alpha(☆, \textit{leaving } \alpha) = (e) \circ EX\alpha(☆, \textit{leaving } \beta_K).$$

LEMMA 5.9. *Let α be a* **while** *statement and let ψ be (b, q, a) in Fig. 5.1 for α. Then $EX\alpha$ is a flow cover for the expected flow scheme, where*

$$EX\alpha(☆, ☆) = \lambda \mathbin{@} \psi \quad \textit{and} \quad EX\alpha(☆, \textit{entering } \beta) = (b) \mathbin{@} \psi;$$

$$EX\alpha(☆, \textit{leaving } \beta) = (b, q) \mathbin{@} \psi \quad \textit{and} \quad EX\alpha(☆, \textit{leaving } \alpha) = (e) \mathbin{@} \psi.$$

Proof. We derive $\lambda \mathbin{@} \psi$ for $EX\alpha(☆, ☆)$ by optimizing the flow cover X provided by Lemma 2.6, which has

$$X(☆, ☆) = Y(☆, ☆) \wedge Z(☆, ☆);$$

$$Y(☆, ☆) = [(\lambda \mathbin{@} \psi) \circ \lambda] \wedge [(\psi \mathbin{@} \psi) \circ \lambda] \wedge [(\lambda \mathbin{@} \psi) \circ \psi] \wedge [(\psi \mathbin{@} \psi) \circ \psi];$$

$$Z(☆, ☆) = \lambda \wedge \psi.$$

One of the six expressions $\wedge$-ed together here is equivalent to $\lambda \mathbin{@} \psi$, so we want to show that the other five can be omitted. By Definition 2.3 it will suffice to show that X is still a solution when the five subexpressions are omitted. Any path $\mathbf{c}$: ☆ → ☆ does have the form $\lambda \bullet \psi^r \bullet \lambda$ for r in $\mathbf{N}$, so $[\lambda \mathbin{@} \psi : f] \leqq f\mathbf{c}$ just as in the proof of Lemma 2.6. For the other nodes p the reasoning is similar, with fewer subexpressions in $X(☆, p)$ to be omitted. □

LEMMA 5.10. *Let α be an* **until** *statement and let ψ be (q, a, b) in Fig. 5.2 for α. Then*

EXα is a flow cover for the expected flow scheme, where

$$EX\alpha(\star, \star) = \lambda \quad \text{and} \quad EX\alpha(\star, \textit{entering}\ \beta) = [\lambda \ @\ \psi] \circ (i);$$

$$EX\alpha(\star, \textit{leaving}\ \beta) = [(q)\ @\ \psi] \circ (i) \quad \text{and} \quad EX\alpha(\star, \textit{testing}\ \alpha) = [(q, a)\ @\ \psi] \circ (i);$$

$$EX\alpha(\star, \textit{leaving}\ \alpha) = [(q, a, e)\ @\ \psi] \circ (i).$$

LEMMA 5.11. *Let α be a stepped iteration statement and let ψ be (b, q, a) in Fig. 5.2 for α. Then EXα is a flow cover for the expected flow scheme, where*

$$EX\alpha(\star, \star) = \lambda \quad \text{and} \quad EX\alpha(\star, \textit{testing}\ \alpha) = [\lambda\ @\ \psi] \circ (i);$$

$$EX\alpha(\star, \textit{entering}\ \beta) = [(b)\ @\ \psi] \circ (i) \quad \text{and} \quad EX\alpha(\star, \textit{leaving}\ \beta) = [(b, q)\ @\ \psi] \circ (i);$$

$$EX\alpha(\star, \textit{leaving}\ \alpha) = [(e)\ @\ \psi] \circ (i).$$

THEOREM 5.12. *Let $\mathcal{P}$ be a global flow problem (with a rapid closed context) derived from a classical program. Then $\mathcal{P}$ is well-nested. Using the given flow cover EXα for each statement α,* SOLVE *finds a good solution. The time required (apart from input/output) is* $O(|\Sigma| \cdot t_@)$.

Proof. That $\mathcal{P}$ is well-nested follows from Lemmas 5.4–5.11, with (4.1) holding because the parse tree is indeed a tree. By Theorem 4.4, SOLVE finds a good solution. All that remains is to specialize the time bound in the theorem. When the pairs (m, p) in ENTR$\alpha \times N G\alpha$ are ordered in the obvious ways suggested by the displays of $EX\alpha$ in the lemmas, the jth $EX\alpha(m, p)$ evaluation in LOCALMAPS requires $(S\alpha)_j$ steps with

$$(S\alpha)_j \leqq 1 \text{ except } (S\alpha)_{2K+2} = 2K - 1 \quad \text{in Lemma 5.7;}$$

$$(S\alpha)_j \leqq t_@ + 3 \quad \text{in the other lemmas.}$$

Therefore $T\alpha$ in Theorem 4.4 is $O(|AG\alpha| + |NG\alpha| \cdot t_@ + |NG\alpha|)$. By $t_@ \geqq 1$ and (5.2.2), $T\alpha$ is $O(Kt_@)$ for $K = |\text{PART}\alpha|$. Therefore $T\alpha_1 + \cdots + T\alpha_{|\Sigma|}$ is $O(|\Sigma| \cdot t_@)$. But $|\mathbf{N}_G|$ is $O(|\Sigma|)$ by (5.2.2) and (5.1.1) and $|N_0\alpha| \leqq |NG\alpha|$, so the total time is $O(|\Sigma| \cdot t_@)$. □

For example, consider the syntax-directed available expressions analysis in [TK76, p. 362]. The stated system of equations is inadequate because Case 4 uses OUT($S1$, x) for $x \neq$ IN($S1$) but the four cases only define OUT(S, x) for x = IN(S). After the obvious corrections it is clear that the method of [TK76] is a special case of SOLVE with $V\ @\ U = V \wedge (V \circ U)$ for idempotent contexts.

Theorem 5.12 above and Theorem 6.7 in the next section show that SOLVE is reasonably fast for a large class of graphs. This class is not universal in programming, but SOLVE is *robust.* It works, perhaps slowly, for any graph. As increasingly complicated graphs are considered, the running time deteriorates gradually. There is no sudden need for a difficult new global analysis in order to somehow apply a method to graphs for which it was not originally designed. In this respect SOLVE is like [BJ78a], [CC77], [KU76], [Ke75], [Ki73], [Ta75], [Ta76], [We75] and unlike [AC76], [FKZ76], [FKU75], [GW76], [TK76], [U173], [Wu75], [ZB74]. Among robust algorithms, SOLVE has an unusual combination of simplicity, algebraic generality, and ability to exploit common regularities in the graphs of well-written programs.

6. Control structures: escapes and jumps. Complex statements in programs commonly have goals, as when one writes a statement that searches a file for the record with a given key. For a large file with a complicated indexing structure, the search statement could be quite complicated. For example, α: (find the record) might be elaborated in a top-down manner as

(6.1.1) α: [β_1: (look in fast memory); β_2: (look in slow memory if necessary)]

if part of the file is in fast memory and part is in slow. If β_1 does find the record, then there is no need to proceed to β_2. The goal of α has been accomplished and the programmer wants to ensure that control will leave α without anything else happening. In a language like Bliss/11 [Wu75] this intention can be expressed directly by writing a simple statement

(6.1.2) δ: **leave** LABEL(α)

anywhere within α, perhaps as the **then** part of a conditional statement. Of course LABEL(α) is the identifier used to label α in the program: any statement important enough to be left is important enough to be labelled. As an operator, **leave** takes the *name* of a statement as an argument rather than the statement itself. With (6.1.2) we have $N\delta = \{$*entering* δ, *leaving* $\delta\}$ as in Definition 5.3(1), but $A\delta = \{($*entering* δ, *leaving* $\alpha)\}$ instead of Definition 5.3(2). The expected induced graph from Definition 5.3 will not be relevant for **leave**. Designated entrances and exits are still as in (5.1.2), but now *entering* δ is also in EXITβ for any β with $\delta \leqq \beta < \alpha$ by (3.2.4). Of course one can avoid (6.1) and stay within classical structured programming by writing something like

```
          α: [  NOTYET := true;
                β₁: (look in fast memory);
(6.2.1)         if NOTYET then
                    β₂: (look in slow memory)
             ]
```

and being careful to put enough assignments

(6.2.2) NOTYET := **false**

into the elaboration of β_1. (Tests of NOTYET will also be needed.)What useful purpose is served by this exercise? One can agree with Ledgard and Marcotty's criticisms of some of the literature recommending escapes or jumps [LM75, p. 638] while contending that escapes (and perhaps jumps) should nevertheless be provided.

Note that clarity and reliability are the main reasons for preferring (6.1) to (6.2). The greater efficiency of (6.1), before the optimizations contemplated on page 638 of [LM75], is just a pleasant byproduct of expressing programmer intentions as directly as possible. No good is done when the programmer cleverly codes (6.1) as (6.2) and then the compiler cleverly optimizes (6.2) back to (6.1). Finally, note that **leave** is quite unlike the escape statements emphasized in [LM75] and its main references. Unlike escaping from the r-th enclosing loop or escaping to the next iteration of the r-th enclosing loop's body, **leave** escapes from *whatever* the programmer wants to escape from, referring to it by whatever name the programmer wants to use. There is no extraneous counting and no entangling of escapes with loops. If nonclassical simple statements are to be allowed at all, there is no point in having them be simultaneously less powerful, less readable, and more bothersome to implement than **leave**.

One sometimes wants more power than **leave** provides. In our file searching example, suppose now that there may or may not be a record matching the given key. One would like to follow the search with the appropriate action, depending on whether it was terminated by success or by exhaustion of the file. This can be accomplished by setting a variable EVENT within the search and then testing EVENT after it, but a great many spurious control flow paths will be introduced by this maneuver. As in (6.1) and (6.2), one would like a more direct means of expressing intended control flow. The "event-driven case statement" [Kn74], [Za74] addresses this need, but the name and syntax are burdened with confusing puns on other control structures. It is confused with the CSP **case** statement in Fig. 9(a) of [LM75] and with the CSP **until** statement in § 5 of [Han77]. These problems are avoided in the more general but simpler "followed statement" introduced in [Ro77b, § 8]. To save space, we will only deal explicitly with the simpler escape statement **leave** in applying Theorem 4.4. Those who wish to handle more powerful escapes will find it easy to generalize this section in light of [Ro77b, § 8].

DEFINITION 6.3. A complex statement α is *semiclassical* iff it is built by a CSP operator and satisfies

(1) $$\mathrm{ENTR}\alpha = \{\textit{entering}\ \alpha\};$$

(2) $$RAG\alpha - IAG\alpha = A_0\alpha \cup \bigcup_{\beta\in\mathrm{PART}\alpha}\{e \in A\alpha \mid \mathbf{s}e \in \mathrm{N}\beta\,\&\ \mathbf{t}e = \textit{leaving}\ \alpha\}.$$

A simple statement δ is *semiclassical* iff it is either classical or a **leave** statement. A program is *semiclassical* iff every simple statement in it is semiclassical and every complex statement is built by a CSP operator.

As will be proved shortly, every complex statement in a semiclassical program is semiclassical. Equations (1) and (2) above are as in [Ro77b, (4.5) and (4.7)], where "semiclassical" is only applied to complex statements. The other applications have been added to bring out the parallels between semiclassical programs and the classical programs of Definition 5.3. The next lemma is like Lemma 5.4.

LEMMA 6.4. *Let G be the graph for a semiclassical program. Then each complex statement α is semiclassical and $G\alpha$ can be obtained from $EG\alpha$ by*

(1) *deleting all imaginary arcs* (*entering* β, *leaving* β) *with* β *in* $\mathrm{PART}\alpha$ *and* Π (β, *entering* β, *leaving* β) $= 0$;

(2) *adding all nodes in* $\mathrm{EXIT}\beta - \mathrm{DEXIT}\beta$ *with* β *in* $\mathrm{PART}\alpha$;

(3) *adding all real arcs* (n, *leaving* α) *for* n = *entering* δ *with* $[\delta\colon \mathbf{leave}\,\mathrm{LABEL}(\alpha)] < \alpha$;

(4) *adding all imaginary arcs* (*entering* β, n) *for* n *in* $\mathrm{EXIT}\beta - \mathrm{DEXIT}\beta$ *with* β *in* $\mathrm{PART}\alpha$ *and* Π (β, *entering* β, n) $= 1$.

Proof. We claim that any complex statement α satisfies Definition 6.3(1) and Definition 6.3(2). For Definition 6.3(1), suppose there is an arc (p, m) in G with p not in $N\alpha$ and m in $N\alpha$. We must show that m = *entering* α. Let γ be the statement with (p, m) in $A_0\gamma$. Suppose first that γ is complex, so that (p, m) is as in Fig. 5.1 or Fig. 5.2. Thus $\alpha \leqq \gamma$, with $=$ ruled out because p is not in $N\alpha$, so α is a part of γ and m can only be *entering* α. Now we show that γ must indeed be complex by showing that it cannot be a classical simple statement or a **leave** statement. In either case we would have p = *entering* γ and m = *leaving* β for a statement $\beta \geqq \gamma$. Because p is not in $N\alpha$, this implies $\neg(\gamma \leqq \alpha)$. Because m is in $N\alpha$, it also implies $\beta \leqq \alpha$. But $\gamma \leqq \beta \leqq \alpha$ implies $\gamma \leqq \alpha$, a contradiction.

For Definition 6.3(2), the right side of the equation is already a subset of the left side in the induced graph construction (3.5). Now consider any arc e in $RAG\alpha - IAG\alpha - A_0\alpha$, so that e is in $A\beta$ for some part β of α and in $A_0\gamma$ for some $\gamma \leqq \beta$. No matter which type of statement γ is, $\mathbf{s}e$ is in $N\gamma$ and so $\mathbf{s}e$ is in $N\beta$. We need $\mathbf{t}e$ = *leaving* α. It will suffice to show that γ has the form **leave** $\mathrm{LABEL}(\alpha)$.

Case 1 [β has the form **leave** LABEL(α')]. Then $\beta < \alpha'$ but $\alpha' \leqq \alpha$ because **t**e is in $N G\alpha$. Therefore $\alpha' = \alpha$. But $\gamma = \beta$ because $\gamma \leqq \beta$ and β is simple, so γ has the desired form.

Case 2 [β is a complex statement or a classical simple statement]. Each arc in $G\alpha$ with source in ENTRβ is imaginary. Because **s**e is in $NG\alpha \cap N\beta \subseteq \mathrm{ENTR}\beta \cup \mathrm{EXIT}\beta$ and e is not imaginary, **s**e is in EXITβ. Because e is not in $A_0\alpha$, **s**e is in EXITβ − DEXITβ. Therefore β is complex and γ has the form **leave** LABEL(α') with $\beta < \alpha'$. As in Case 1, $\alpha' = \alpha$.

Now consider the problem of obtaining $G\alpha$ from $EG\alpha$. All nodes of $EG\alpha$ are in $G\alpha$, but some imaginary arcs may not be. The missing arcs are exactly the ones deleted by (1). Because any part β of α has ENTR$\beta = \{$*entering* $\beta\}$, the additional nodes in $G\alpha$ are exactly those added by (2). By Definition 6.3(2), the additional real arcs in $G\alpha$ are exactly those arcs e not in $A_0\alpha$ with **s**e in $N\beta$ and **t**e = *leaving* α. The statements δ that could have such arcs e in $A_0\delta$ are exactly those considered in (3). By ENTRβ = $\{$*entering* $\beta\}$, the additional imaginary arcs in $G\alpha$ are exactly those (*entering* β, n) such that n is in EXIT − DEXITβ and is reachable from *entering* β, and these are exactly the arcs added by (4). □

LEMMA 6.5. *Let α be a semiclassical simple statement. Then $X\alpha$ is a flow cover for the induced flow scheme* ($G\alpha$, ENTRα), *where*

$$X\alpha = EX\alpha \quad \textit{if } \alpha \textit{ is classical};$$

$$X\alpha(\star, \star) = \lambda \quad \textit{and} \quad X\alpha(\star, \textit{leaving } \alpha) = \top \quad \textit{if } \alpha \textit{ is a } \textbf{leave} \textit{ statement.}$$

Proof. If α is classical then $G\alpha = EG\alpha$ and Lemma 5.5 applies. Otherwise, $G\alpha$ has nodes $\{\star$, *leaving* $\alpha\}$ and no arcs while ENTR$\alpha = \{\star\}$. □

LEMMA 6.6. *Let α be a complex statement in a semiclassical program. Let $Y\alpha$ be the result of adjusting the expected flow cover $EX\alpha$ to avoid using any imaginary arc of $EG\alpha$ that is not in $G\alpha$, as in Lemma* 2.10. *Let $EA\alpha$ be the set of all e in $RAG\alpha - IAG\alpha - A_0\alpha$ such that* $\Pi(\beta$, *entering* β, **s**$e) = 1$, *where β is the part of α with $e \in A\beta$. Then $X\alpha$ is a flow cover for the induced flow scheme* ($G\alpha$, $\{\star\}$), *where each node p in $NG\alpha$ has $X\alpha(\star, p)$ determined by one of the following cases*:

(1) $X\alpha(\star, p) = Y\alpha(\star, p)$ *if p is in $EG\alpha$ but $p \neq$ leaving* α;

(2) $X\alpha(\star, p) = \top$ *if* $p \in \mathrm{EXIT}\beta - \mathrm{DEXIT}\beta$ *and* $\Pi(\beta$, *entering* $\beta, p) = 0$;

(3) $X\alpha(\star, p) = (i) \circ X\alpha(\star$, *entering* $\beta)$
if $p \in \mathrm{EXIT}\beta - \mathrm{DEXIT}\beta$ *and* i
= (*entering* β, p) *is in* $IAG\alpha$;

(4) $X\alpha(\star, p) = Y\alpha(\star, p) \wedge \bigwedge_{e \in EA\alpha} [(e) \circ X\alpha(\star, \mathbf{s}e)]$ *if p = leaving* α.

Proof. By Lemmas 5.6–5.11, $EX\alpha$ is a flow cover for ($EG\alpha$, $\{\star\}$). By Lemma 2.10, $Y\alpha$ is a flow cover for (H, $\{\star\}$), where H is the result of deleting all imaginary arcs in $EG\alpha$ that are not in $G\alpha$. The actual induced flow scheme is formed from (H, $\{\star\}$) by adding nodes and arcs as in Lemma 6.4, and $X\alpha$ is derived from $Y\alpha$ by taking into account the additional paths in $G\alpha$. For p in (1) the paths from $\star$ to p in $G\alpha$ are exactly the paths from $\star$ to p in H, as follows from (5.2.4) and Lemma 6.4: no arc in $G\alpha$ has source *leaving* α and the only arcs with sources in EXITβ − DEXITβ have target *leaving* α. For p in(2) there are no paths from $\star$ to p. For p in (3) the paths from $\star$ to p are exactly those of the form **c** • (i) where **c** is a path from $\star$ to *entering* β for the unique β in PARTα with p in EXITβ − DEXITβ. For p in (4) the paths from $\star$ to p are exactly those from $\star$ to p in H together with those of the form **c** • (e) for e in $EA\alpha$ and c: ☆ → **s**e. Therefore $X\alpha$ is a flow cover because $Y\alpha$ is a flow cover. □

We can now generalize Theorem 5.12 to allow escape statements. The new time bound in the following theorem is roughly similar to the bound for the method of Graham and Wegman as applied to semiclassical programs [GW76, Thms. 4.2 and 5.4].

THEOREM 6.7. *Let $\mathcal{P}$ be a global flow problem (with a rapid closed context) derived from a semiclassical program. Then $\mathcal{P}$ is well-nested. Using the flow cover $X\alpha$ from Lemma* 6.5 *or Lemma* 6.6 *for each statement α,* SOLVE *finds a good solution. The time required (apart from input/output) is $O(|\Sigma| \cdot t_@ + (newexits))$, where (newexits) is the sum over all α in Σ of the numbers* $|\mathrm{EXIT}\alpha| - 1$.

Proof. We proceed much as in the proof of Theorem 5.12, using Lemmas 5.4–5.11 and 6.4–6.6. The $X\alpha(m, p)$ evaluations for p in $EG\alpha$ with $p \neq leaving\ \alpha$ are ordered as suggested by the displays of $EX\alpha$ in the lemmas from § 5. Then $X\alpha(m, p)$ is evaluated for each p in $G\alpha$ not in $EG\alpha$ (and hence in $\mathrm{EXIT}\beta - \mathrm{DEXIT}\beta$ for β a part of α). Finally, we evaluate $X\alpha(m, leaving\ \alpha)$. As in Theorem 5.12, the time $(S\alpha)_j$ for the jth $X\alpha(m, p)$ evaluation with p in $EG\alpha$ has $(S\alpha)_j \leqq t_@ + 3$ except perhaps for the last one, with $p = leaving\ \alpha$. In this one case Lemma 6.6(4) yields

$$(S\alpha)_j \leqq 1 + (\mathrm{old}) + (\mathrm{leavenum} - 1) + (\mathrm{leavenum}),$$

where (old) is the old bound from the proof of Theorem 5.12 on the time required to evaluate $EX\alpha$ and hence $Y\alpha$ for $(m, leaving\ \alpha)$ and (leavenum) is the number of **leave** LABEL(α) statements within α. There are also (leavenum) + $|\mathrm{EXIT}\alpha| - 1$ choices of j with p not in $EG\alpha$ and $(S\alpha)_j \leqq 1$ in Lemma 6.6(2) and Lemma 6.6(3). The term $T\alpha$ from Theorem 4.4(2) is therefore

$$O(|AG\alpha| + (|NEG\alpha| - 1) \cdot (t_@ + 3) + |\mathrm{EXIT}\alpha| - 1 + (\mathrm{old}) + (\mathrm{leavenum}) + \mathrm{pairs}(\alpha)),$$

where $EG\alpha$ has node set $NEG\alpha$ and arc set $AEG\alpha$ such that

$$|AG\alpha| \leqq |AEG\alpha| + 2 \cdot (\mathrm{leavenum});$$

$$\mathrm{pairs}(\alpha) \leqq |NEG\alpha| + (\mathrm{leavenum}) + |\mathrm{EXIT}\alpha| - 1.$$

Now (leavenum) is at most the sum of all the numbers $|\mathrm{EXIT}\beta| - 1$ for β in PARTα. For $K = |\mathrm{PART}\alpha|$, $|\mathrm{AEG}\alpha|$ and $|NEG\alpha|$ are $O(K)$ by (5.2.2) and (old) is linear in $K \cdot t_@$, so $T\alpha$ is

$$O(K \cdot t_@ + (|\mathrm{EXIT}\beta_1| - 1) + \cdots + (|\mathrm{EXIT}\beta_K| - 1) + (|\mathrm{EXIT}\alpha| - 1)).$$

But $|\mathbf{N}_G|$ is still $O(|\Sigma|)$, so the total time from Theorem 4.4(1) is $O(T\alpha_1 + \cdots + T\alpha_{|\Sigma|})$, which is the desired bound. □

Vague assertions about the importance of "single-entry/single-exit control structures" are ubiquitous in the literature and folklore of structured programming. One precise meaning for such assertions might be that syntax and semantics are so simply related as to permit syntax-directed data flow analysis at a cost linear in some reasonable measure of program size. When the size of a program is the number of statements in it, multiplied by $t_@$, Theorem 5.12 shows that classical structured programming is sufficient for such analyzability. Theorem 6.7 shows that the escapes needed for practical structured programming do not destroy this analyzability, provided they are used in *moderation*: the number (newexits) should be fairly small compared to the size of the program.

Because LOCALMAPS in SOLVE uses a given flow cover for each statement's induced flow scheme, the time bound in Theorem 6.7 would be of little interest if finding flow covers required more time than using them did. Happily, Lemma 6.6 shows that we can pass from the expected flow cover to the actual one very quickly, provided we know the various path bits $\Pi(\beta, m, n)$. Finding all these bits can be done in time $O(|\Sigma|+$ (newexits)) by the obvious adaptation of the rules for computing whether a variable can be preserved along some path through a statement [Ro77b], and the use of $X\alpha(m, p)$ in LOCALMAPS can be reinterpreted as including discovery of $X\alpha(m,p)$ without changing the time bound. This suggests a strategy for dealing with programs that are not known to be semiclassical. For definiteness, let us assume that the only kind of statement besides those already considered is the simple jump:

$$(6.8) \qquad \delta: \textbf{goto}\ \text{LABEL}(\alpha).$$

With (6.8) we have $N\delta = \{\textit{entering } \delta, \textit{ leaving } \delta\}$ as in Definition 5.3(1), but $A\delta = \{(\textit{entering } \delta, \textit{ entering } \alpha)\}$ instead of Definition 5.3(2). As with (6.1.2), the expected induced graph $EG\delta$ is irrelevant, designated entrances and exits are still as in (5.1.2), and *entering* δ is also in EXITβ for any β with $\delta \leqq \beta < \alpha$ except in the special case where α happens to be δ. But now we may have nondesignated entrances as well. In (3.2.3) we find that *entering* α is also in ENTRβ for any β with $\neg(\delta \leqq \beta)$ and $\alpha < \beta$. There is a new real arc in $G\gamma$ for the smallest γ that includes both δ and α. Unlike the arcs added by escape statements, these new arcs may add many new paths to $G\gamma$, including cycles. There is no neat formula like Definition 6.3(2).

Our strategy for coping with jumps is based on a cautiously optimistic version of Murphy's Law: WHATEVER CAN GO WRONG WILL, BUT NOT OFTEN. A compiler's lexical analysis phase can easily detect the presence of jumps in a program. Well-written programs in well-designed languages will often be free of jumps and hence semiclassical. Most of the statements in a program that does have jumps will still be semiclassical, though they may have exits due to **goto** as well as to **leave**. The lemma after this paragraph implies that Lemma 6.6 actually holds for any semiclassical complex statement, even when the program as a whole is not semiclassical. The flow cover discovery in LOCALMAPS can be written as an easy test for whether α is semiclassical, followed by use of Lemma 6.5 or Lemma 6.6 in the common case where this is true. When α is complex and not semiclassical, the flow cover discovery routine will need to construct the induced graph $G\alpha$, using the path bits that will be available for the parts of α and a list of the arcs contributed by jumps. When this graph is monocyclic or acyclic (as can quickly be tested [MD76]), a flow cover can be obtained from Lemma 2.6. (In this context it may be worthwhile to optimize the proof of the lemma to avoid enumerating so many sets of simple paths.) Lemma 2.7 covers the polycyclic case. Let Q be a set of arcs in $G\alpha$ such that every cycle includes at least one arc from Q. The methods of [SMR75] can be adapted so as to choose Q in time linear in the size of $G\alpha$, with $|Q| \ll |AG\alpha|$ in many examples but with no minimality assured. If Lemma 2.6 takes time T_0 to find a flow cover for $(H, NG\alpha)$, where h is the result of removing all arcs in Q from $G\alpha$, then an easy optimization of Lemma 2.7 will find a flow cover for the actual induced flow scheme in time $O(T_0 + |Q|n^2 + t)$, where $n = |NG\alpha|$ and t is the time required to find the transitive closure of H if this does not fall out from the Lemma 2.6 implementation. If the flow cover for $(H, NG\alpha)$ can be evaluated for all (m, n) in T_1 steps, then the flow cover for the actual induced flow scheme can be evaluated in at most $T_1 + |Q|n^2(2t_@ + 7)$ steps. By compressing the induced graph in various ways (e.g. [Ro77a, (2.6) and (2.7)]), we can use $n < |NG\alpha|$ and then add a term $O(|NG\alpha|)$. It is not clear how bad the above bounds really are. Even with $Q = AG\alpha$, so that the crude algorithm is cubic in the size of $G\alpha$, it is only the size of $G\alpha$ which occurs. No practical estimates of the size ratio between this graph and the graph of the entire program are available. *If* jumps are used in such a way that few statements fail to be semiclassical and those that do have small induced graphs, then even crude handling of these statements by LOCALMAPS need not drastrically change the running time of SOLVE on large programs. As with escape statements, moderation is the key.

LEMMA 6.9. *Let G be the graph of a program wherein every complex statement is built by a CSP operator and every simple statement is either semiclassical or a* **goto**. *Let α be a complex statement. Then*

$$(1) \qquad \mathrm{ENTR}\alpha - \mathrm{DENTR}\alpha \subseteq \bigcup_{\beta\in\mathrm{PART}\alpha} \mathrm{ENTR}\beta.$$

Moreover, α is semiclassical iff each part β of α satisfies

$$(2) \qquad \mathrm{ENTR}\beta = \{\mathit{entering}\,\beta\} \quad \text{and} \quad \mathit{entering}\,\beta \notin \mathrm{ENTR}\alpha$$

and, for all $[\delta\colon \mathbf{goto}\ \mathrm{LABEL}(\alpha')] \leqq \beta$,

$$(3) \qquad \alpha' \leqq \alpha \quad \text{implies} \quad \alpha' \leqq \beta.$$

In that case $G\alpha$ can be obtained from $EG\alpha$ by the operations from Lemma 6.4(1−4).

Proof. The only way that m with $m \neq \mathit{entering}\,\alpha$ can be an entrance to α is for m to be *entering* α', where $\delta\colon\mathbf{goto}$ LABEL(α') has $\alpha' < \alpha$ but $\neg\delta \leqq \alpha$. (Recall the first paragraph in the proof of Lemma 6.4.) Then $\alpha' \leqq \beta$ for some β in PARTα and $m \in \mathrm{ENTR}\beta$. This proves (1).

Suppose (2) and (3). We show that α is semiclassical. Definition 6.3(1) follows from (1) and (2):

$$\mathrm{ENTR}\alpha - \mathrm{DENTR}\alpha \subseteq \bigcup_{\beta\in\mathrm{PART}\alpha} \mathrm{ENTR}\beta \cap \mathrm{ENTR}\alpha = \varnothing.$$

For Definition 6.3(2) the reasoning from the proof of Lemma 6.4 is valid as far as it goes, but the case analysis is no longer obviously exhaustive. In Case 2 there seems to be a new subcase: perhaps the statement γ with e in $A_0\gamma$ has the form **goto** LABEL(α') with $\neg\alpha' \leqq \beta$. But then $\neg\alpha' \leqq \alpha$ by (3), so e cannot be in $G\alpha$ after all. Similarly, the apparent Case 3, with β a **goto** statement, cannot arise.

Conversely, suppose α is semiclassical. Let β be a part of α, so that *entering* β is not an entrance to α by Definition 6.3(1). To complete the proof of (2) we suppose $m \neq \mathit{entering}\,\beta$ is in ENTRβ and derive a contradiction. Some $\delta\colon\mathbf{goto}$ LABEL(α') has $m = \mathit{entering}\,\alpha'$ and $\alpha' < \beta$ and $\neg\delta \leqq \beta$, but $\delta \leqq \alpha$ because m is not an entrance to α. Therefore $\delta \leqq \gamma$ for some part $\gamma \neq \beta$ and δ contributes an arc to $G\alpha$ not allowed by Definition 6.3(2), a contradiction. To prove (3), suppose $[\delta\colon\mathbf{goto}\ \mathrm{LABEL}(\alpha')] \leqq \beta$ and $\alpha' \leqq \alpha$. But $\alpha' \neq \alpha$ lest $G\alpha$ have an arc not allowed by Definition 6.3(2). Therefore $\alpha' \leqq \gamma$ for a part γ with $\gamma = \beta$ lest $G\alpha$ have an arc not allowed by Definition 6.3(2). Finally, the proof that $G\alpha$ is obtained from $EG\alpha$ by the indicated operations is as before. □

COROLLARY 6.10. *Let $\mathcal{P}$ be a global flow problem* (*with a rapid closed context*) *derived from a program wherein each complex statement is built by a CSP operator and each simple statement is either semiclassical or a* **goto**. *Using the flow cover $X\alpha$ from Lemma* 6.5 *or Lemma* 6.6 *for each semiclassical statement α, and using a flow cover $X\alpha$ constructed during the call on* LOCALMAPS *for each complex statement α that is not semiclassical,* SOLVE *finds a good solution.*

In an ambitious optimizing compiler along the lines of [Kn74], [Lo77] we must also cope with *introduced* jumps. The original input to the compiler has few if any jumps, but subsequent processing can add them in two ways. Some optimizations introduce jumps so as to avoid tests when their results are predictable at compile time or so as to make procedure linkages more efficient [Kn74, esp. pp. 281, 282]. Some high-level language features need to be paraphrased by less concise but equivalent uses of several lower-level features to reveal opportunities for optimization, and such paraphrases may add jumps. For example, a **while** loop governed by a Boolean expression that may have side effects can be paraphrased by a block that declares a temporary variable, explicitly evaluates the expression and assigns the result to the temporary, and then enters a loop governed by the temporary. The need to reevaluate the expression after each iteration leads to a jump [Ro77b, (9.2.2)]. Fortunately, there is no need to treat added jumps like original jumps. At the time when a compliler adds a jump like the one in [Ro77b, (9.2.2)], it is easy to determine which statements may have their entrance and exit sets or induced graphs changed. Precisely because jumps are added according to "well-understood · · · [and] well-documented "mechanical" operations" [Kn74, p. 282], there is no need to apply Lemma 2.6 or Lemma 2.7 to the new induced graphs. Part of understanding and documenting the operations is drawing figures like [Ro77b, Fig. 5] for **while** loops governed by Boolean expressions with side effects. As with the loops of classical structured programming, we can find the flow cover long before the compiler encounters a specific **while** statement in a specific program. In writing the compiler we need only write the use of this known flow cover into the handling of **while** statements by LOCALMAPS.

Definition 2.1 can be extended to define *data* expressions with values in L in addition to *flow* expressions with values in M. Then LOCALMAPS and LOCALSOLVE can be rewritten so as to use $|\mathrm{ENTR}\alpha \times \mathrm{EXIT}\alpha| + |NG\alpha - \mathrm{EXIT}\alpha|$ expressions for each statement α instead of the present $|\mathrm{ENTR}\alpha \times \mathrm{NG}\alpha|$ expressions. This change complicates SOLVE but may improve its running time if $|\mathrm{ENTR}\alpha| > 1$. Because multiple exits are more common than multiple entrances in structured programming, the use of data expressions is more likely to be worthwhile as a prelude to reversing the roles of entrances and exits. Reversing is needed to solve problems like the detection of dead variables, wherein information flows backwards along arcs and is given initially for exit nodes at the time of exit.

7. Semilattices of finite height. Most of the semilattices that arise in data flow analysis are of finite height: not only are they well-founded, but there is a uniform bound on the lengths of strictly descending chains.

DEFINITION 7.1. Let H be a positive integer. The semilattice L has *height* H iff there is a strictly descending chain $x_0 > x_1 > \cdots > x_H$ such that all other strictly descending chains are at most this long.

If L has height H then any context (L, M) is rapid, but a tighter bound on $t_@$ in Definition 1.6 can be obtained if we consider the "effective height" instead of the actual height. To define effective heights we recall the general cartesian product construction. The general construction begins with an arbitrary *family* of sets: for each q in some set Q, we are given a set S_q. Given such a family, a Q-*tuple* is any map x with domain Q, such that the value x_q of x at q is a member of S_q. We wrote x_q rather than our usual xq to stress the similarity between general Q-tuples and the more familiar n-tuples $(x_1, \cdots, x_n)$, which correspond to the special case $Q = \{1, \cdots, n\}$. The *cartesian product* of the family $\{S_q | q \in Q\}$ is the set of all Q-tuples. It is denoted $\times_{q\in Q} S_q$. If each S_q happens to be a (complete) semilattice, then the product is also a (complete) semilattice, with the obvious ordering $x \leqq y$ in $\times_{q\in Q} S_q$ iff $x_q \leqq y_q$ in S_q for all q.

DEFINITION 7.2. A *factorization* of the algebraic context (L, M) is any family $\{(L_q, M_q) | q \in Q\}$ of algebraic contexts such that there are semilattice isomorphisms

$$L \to \times_{q\in Q} L_q \quad \text{and} \quad M \leftrightarrow \times_{q\in Q} M_q$$

that make the following diagram diagram commute:

$$\begin{tikzcd}
M \times L \arrow[r, leftrightarrow] \arrow[d, "\textit{apply a map to an argument}"'] & \times_{q \in Q} M_q \times \times_{q \in Q} L_q \arrow[d, "\textit{apply the q-th map to the q-th argument}"] \\
L \arrow[r, leftrightarrow] & \times_{q \in Q} L_q
\end{tikzcd}$$

For example, let Q be the set of all expressions in a program. For each expression q, we can indicate whether the expression is available or not by using the semillatice $L_q = \{0, 1\}$ with the usual ordering and the monoid M_q generated by the maps describing how a block of text "kills" or "generates" the expression [GW76, p. 178], [U173, p. 193]. The height of L_q is 1. Instead of solving a separate global flow problem for each expression q, it is usual to use a "bit vector" with one position for each expression: the parallelism in AND and OR operations on arrays $|Q|$ long of bits is exploited to get the net effect of solving $|Q|$ problems in $\{0, 1\}$ by solving one problem in $\times_{q \in Q} L_q$. The above definition says why this *works*. In the bit vector example $\times_{q \in Q} L_q$ has height $|Q|$, but it acts as if it has height 1 for all purposes of data flow analysis.

DEFINITION 7.3. The semilattice L has *effective height* H in the context (L, M) iff there is a factorization $\{(L_q, M_q) | q \in Q\}$ of (L, M) such that some L_q has height H and all L_q have height at most H.

LEMMA 7.4. *Suppose L has effective height H in the context (L, M). Given V and U in M, the* loop products *$@_1$ and $@_2$ defined by*

$$V @_2 U = V \circ U^* \quad \textit{and} \quad V @_2 U = \bigwedge \{V \circ U^r | r \in \mathbf{N}\}$$

can be computed in at most

(1) $t_@ = \lceil \log_2 H \rceil + 2$ *steps for* $@ = @_1$;

(2) $t_@ = 2H + 1$ *steps for* $@ = @_2$.

Proof. Suppose first that L has height H. For (1), it takes one step to find $U \wedge \mathbf{1}$ and then $\lceil \log_2 H \rceil$ steps to find U^* by repeated squaring, as in [GW76, p. 182]. One more step finds $V \circ U^*$. For (2), it takes $H+1$ steps to find the sequence $(V \circ U, V \circ U^2, \cdots, V \circ U^{H+1})$ and then H steps to find the sequence $(\phi_0, \cdots, \phi_H)$ with $\phi_0 = V \circ U$ and $\phi_h = \phi_{h-1} \wedge V \circ U^{h+1}$. For any x in L we have $\phi_0 x \geqq \phi_1 x \geqq \cdots \geqq \phi_H x$, so ϕ_H is $V @_2 U$.

In the general case Definition 7.2 implies that $V @_k U$ corresponds to a Q*-tuple* with $(V @_k U)_q = V_q @_k U_q$, the $@_k$ on the right being in M_q. Because L_q has height H or less, $V_q @_k U_q$ can be found in the asserted number of steps within M_q. By Definition 7.2 again, $V @_k U$ can be found in the asserted number of steps within M. □

COROLLARY 7.5. *If L has effective height H in the context (L, M) then (L, M) is rapid with $t_@ \leqq 2H + 1$. If (L, M) is also distributive then $@_1 = @_2$ and $t_@ \leqq \lceil \log_2 H \rceil + 2$.*

For any of the traditional bit vector problems the effective height is 1. Effective heights can be unpleasantly high in constant propagation [KU76, p. 167] or in analyzing the ranges of values of variables [Har77b]. In these more ambitious forms of data flow analysis the contexts are also not distributive, so $@_1$ is cheaper to compute but $@_2$ yields sharper information. The programmer could be asked to specify which loop product is to be used when invoking an ambitious optimizing compiler. After run time measurements prior to optimization have revealed which areas of the program are critical, the compiler could even be told to use $@_2$ in a few critical places and $@_1$ elsewhere. With high-level analysis the programmer and the compiler *both* visualize control flow in relation to the syntactic structure, so detailed communication is possible without an elaborate interface.

For a simple first example of heights greater than 1 and of the difference between the two loop products, let L_{012} be as in Fig. 7.1(left), so that L_{012} has height 2. Intuitively, this semilattice can be used to count the number of times a loop body is executed, with 2 standing for "2 or more." Consider the following isotone maps on L_{012}:

$$b'x = (\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else if } x = 1 \textbf{ then } 2 \textbf{ else } x);$$

$$i'x = 0 \text{ and } e'x = (\textbf{if } x = \perp \textbf{ then } \perp \textbf{ else } \top).$$

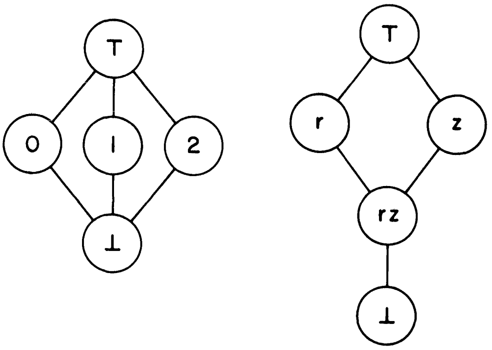


FIG. 7.1. *A semilattice for counting iterations and an elementary semilattice for subscript range analysis.*

These maps together with the constant map $\top_M x = \top$ generate a closed monoid of isotone maps M_{012}. (It is not distributive.) Consider the program statement

$$\alpha: \textbf{do } \beta: \textbf{get}(\text{VAR}) \textbf{ until } \text{VAR}$$

where VAR is a boolean variable. In $G\alpha = EG\alpha$ from Fig. 5.2 we map arcs to members of M_{012} with $fi = i'$; $fq = \mathbf{1}$; $fa = \mathbf{1}$; $fb = b'$; $fe = e'$. The maximum solution to $(G\alpha, f, \{\star\}, E)$ with $E(\star) = \perp$ has

$$I(\star) = I(\textit{entering } \beta) = I(\textit{leaving } \beta) = I(\textit{testing } \alpha) = \perp \text{ but } I(\textit{leaving } \alpha) = \top.$$

The maximum fixpoint has $Jn = \perp$ for all n. The method of [GW76] finds J, as does our method with $@_1$. But with $@_2$ we find I. This example is contrived, but the same behavior could appear in a practical (but much more complicated) situation, where we want to optimize the code following the loop α. Even when the justification for the optimization depends on how many times the loop body was executed, a programmer polishing the code can reason as follows. After 0 iterations the optimization is permissible because of R_0; after 1 iterations it is permissible because of R_1; after 2 or more iterations it is permissible because of R_2; therefore it is permissible. By using a map e' that extracts an appropriate justification from each distinguishable number of iterations, an ambitious optimizing compiler with $@_2$ can mimic this kind of human reasoning in its data flow analysis. (For an **until** loop the case of 0 iterations is trivial, but this kind of loop is most convenient for a simple comparison with [GW76].)

For a more complicated second example that displays the subtleties of the effective height concept, let P be a set of integer valued variables and let L_i be as in Fig. 7.1(right) for each i in P. As in [We75, p. 276], these variables can be used as array indices so as to simulate pointer variables. An array $A[1], \cdots, A[n]$ is given, and the assertions to be made about i are that i is in the range $\{1, \cdots, n\}$ (the r in the figure), that i represents a null pointer by being zero (the z in the figure), and that i represents a pointer by being either in range or zero (the rz in the figure). Then $\perp$ says nothing about the value of i while $\top$ is added for technical convenience. (Strong assertions are high in the semilattice.) Let $L = L_{\text{ptr}}$ be $\times_{i \in V} L_i$ and let $M = M_{\text{ptr}}$ be generated by the constant $\top_M x = \top_L$ and by assignment and input statements, as follows. Consider any i, j in P and any arithmetic expression AE not in P. Each statement δ in the set $\{i := j, i := AE, \textbf{get}(i)\}$ defines (δ) in M with $(\delta)x = y$ in L_{ptr} having $y_k = x_k$ for all k in $P - \{i\}$. But y_i depends on δ and on x as follows.

$$\begin{array}{lll}
y_i = x_j & \text{for} & i := j; \\
y_i = z & \text{for} & i := 0; \\
y_i = r & \text{for} & i := AE \text{ with } AE \text{ constant in } \{1, \cdots, n\}; \\
y_i = \perp & \text{for} & i := AE \text{ otherwise}; \\
y_i = \perp & \text{for} & \textbf{get}(i).
\end{array}$$

The context $(L_{\text{ptr}}, M_{\text{ptr}})$ is rapid because L_{ptr} has height $3|P|$. If $|P| \geqq 3$ then this context is not idempotent and not fast, as can be seen by considering U^2 and $U \wedge \mathbf{1}$ for

$$U = (j := k) \circ (i := j)$$

whenever i, j, k are distinct members of P. Unlike M_{012} and more elaborate monoids for analyzing ranges of array indexing variables, M_{ptr} is distributive. Therefore $t_@$ varies as $\log_2 |P|$. For large P this could be toublesome, but here effective heights will often be useful. Let Q be a *partition* of P: Q is a family of disjoint nonempty subsets of P whose union is P. Suppose each assignment $i := j$ that actually occurs in a program has i and j in the same subset $q \in Q$. This can be accomplished trivially by letting Q be $\{P\}$, but for any *one* program we can probably use a nontrivial partition. Then M_{ptr} can be replaced by the smaller monoid $M_{\text{ptr}}[Q]$ generated by allowing only assignments $i := j$ that do occur. There is a factorization of $(L_{\text{ptr}}, M_{ptr}[Q])$ such that L_{ptr} has effective height $3v$, where v is the largest $|q|$ for q in Q. Letting M vary with the program to be analyzed is a notational complication not considered in the preceding sections, but it is *only* a notational complication. A formally correct treatment would add L and M to the 4-tuple that specifies a global data flow problem. We prefer to minimize notation by letting the slow variation of L and M be tacit. Let e be the larger of $|P|$ and the number of pairs (i, j) such that $i := j$ occurs in a given program. The equivalence relation corresponding to the best partition Q for this program can then be found in time $O(|P|e)$ by the method of [HSU77].

Because of technical complexities in comparing SOLVE with any low-level method, we will only present one simple but instructive general comparison theorem. Assume L has effective height H. Obvious generalizations of the method of Graham and Wegman as presented in [GW76] allow it to be applied to any global flow problem for a classical (as in Definition 5.3) program. (There is one nonobvious point, concerning the entry node n_0. The assumption $En_0 = \perp$ is only needed in [GW76] when n_0 has inarcs. To avoid this assumption we assume instead that n_0 lacks inarcs. A new entry node can always be added to the graph if necessary.) For a classical program [GW76] finds the same solution that SOLVE does when @ is defined in a way that is a little better than $@_1$. The same comparison holds for any semiclassical program, but a proof would be much too tedious to appear here. For each U in M let

$$U^{(*)} = (\textbf{if } U \text{ is fast } \textbf{then } U \wedge \mathbf{1} \textbf{ else } U^*), \tag{7.6.1}$$

so that $U^{(*)}$ can be found in $\lceil \log_2 H \rceil + 3$ steps when testing for $\geqq$ between $U \circ U$ and $U \wedge \mathbf{1}$ is also counted as "one" step. The (L, M) is rapid with

$$V \mathbin{@_{\mathrm{GW}}} U = V \circ U^{(*)}, \tag{7.6.2}$$

which may have $V \mathbin{@_{\mathrm{GW}}} U > V \mathbin{@_1} U$ when U is fast.

THEOREM 7.7. *Let $\mathcal{P} = (G, f, \{entering\ \pi\}, E)$ be a global flow problem derived from a classical program π. Let I_{GW} be the good solution found by the method of Graham and Wegman and let I_{SO} be the good solution found by the method of Graham and Wegman and let I_{SO} be the good solution found by* SOLVE *with* $@_{\mathrm{GW}}$. *Then* $I_{\mathrm{GW}} = I_{\mathrm{SO}}$.

Proof. To each node v in G, the method of [GW76] assigns a map Φ_v in M such that $\Phi_v E n_0 = I_{\mathrm{GW}} v$, where $En_0 = E$ (*entering* π) and n_0 is either *entering* π or a new entry node such that $f(n_0, \textit{entering}\ \pi) = \mathbf{1}$. A variable graph G' and a variable assignment f' of members of M to arcs in G' are initialized to G and f. In the statement of "Algorithm A" [GW76, p. 184], G' is denoted G and the computations on f' are implicit. Details are in the lemmas from [GW76, § 4]. Algorithm A eventually reduces G' to a graph with n_0 as the only node and with no arcs. For $v = n_0$, $\Phi_v = \mathbf{1}$.

For $v \neq n_0$, Φ_v is determined when v is removed from G' by the T_2' or T_3' transformation. In both cases v has a unique inarc (u, v) just before its removal from G', and $\Phi_v = f'(u, v) \circ \Phi_u$. Of course Φ_u is not known at this time, but associating the pair $(u, f'(u, v))$ with v will allow Φ_v to be computed later in a pass over the nodes of G that reverses the order in which they were removed from G'. The algorithm is such that (u, v) can only appear in this way in G' if u dominates v in G, and then $f'(u, v)$ at the time v is removed from G' will summarize what is known about paths from u to v that do not return to u before reaching v. Thus the intution behind $f'(u, v)$ is much like that behind $f_\alpha c$ in (3.7.2), but the pairs (u, v) that arise in [GW76] are governed by properties of the entire control flow graph. Only in classical structured programming do these properties relate to source level syntax in a simple way, as described below.

If α is one of the three kinds of loop statement, let *head* α be *entering* α if α is a **while** statement and *testing* α if α is a stepped iteration statement. Otherwise α has the form **until** $\cdots$ **do** β and *head* α is *entering* β. The set T initially computed by Algorithm A is precisely the set of all *head* α for α a loop statement in π. The initial choice of h from T is the head of an innermost loop, and subsequent computations of T just remove the previous h value and choose the next one to be innermost among the remaining members of T. For $h = \textit{head}\ \alpha$ and $\{\beta\} = \mathrm{PART}\alpha$ the corresponding set S in Algorithm A includes h, *testing* α if α is an **until** or stepped iteration statement, and all nodes of $N\beta$ that are still in G'. There are no other nodes in S. The call on *Reduceset*(S, h) removes all nodes in $S - \{h\}$ except those with outarcs whose targets are not in S. If α is an **until** statement then *testing* α persists after *Reduceset*(S, h), but otherwise all nodes in $S - \{h\}$ have been removed. At this point the next loop to be processed is chosen. When all loops have been processed in this way, G' will be acyclic except for arcs from nodes to themselves.

The above relation between Algorithm A and the syntax of π underlies the proof that $\Phi_v = \Psi_v$ for all nodes v, where Ψ_v is defined by moving top-down through Σ. If v is in $N_0\pi$ then $\Psi_v = F_\pi(n_0, v)$. Otherwise v is in $N_0\beta$ for unique β in $\mathrm{PART}\alpha$ for unique α, and Ψ_u for $u = \textit{entering}\ \alpha$ is already available. Let $\Psi_v = F_\beta(\textit{entering}\ \beta, v) \circ F_\alpha(u, \textit{entering}\ \beta) \circ \Psi_u$. That $\Phi_v = \Psi_v$ can be proved by annotating Algorithm A with appropriate inductive assertions. This is tedious but not difficult. Each call on LOCALSOLVE in SOLVE has $\mathrm{ENTR}\gamma = \{\textit{entering}\ \gamma\}$, so $I_{\mathrm{SO}} v = \Psi_v E n_0$ follows from the programming in § 4. Finally, $I_{\mathrm{GW}} v = \Phi_v E n_0 = \Psi_v E n_0 = I_{\mathrm{SO}} \mathrm{v}$. □

COROLLARY 7.8. *As in Theorem* 7.7, *the good solution I_2 found by* SOLVE *with* $@_2$

has $I_{GW} \leqq I_2$.

Proof. If $U_1 \leqq U_2$ then $V\ @_{GW}\ U_1 \leqq V\ @_2\ U_2$ and $U_1\ @_{GW}\ V \leqq U_2\ @_2\ V$. This implies that interpreting a flow expression with $@_{GW}$ for @ yields a value that is ≦ the value obtained with $@_2$ for @. Therefore $I_{SO} \leqq I_2$ and so $I_{GW} \leqq I_2$. □

8. Conclusions and open problems. The algebraic contexts of data flow problems can be classified by the presence or absence of three properties: idempotence, distributivity, and rapidity. The last is a new concept that generalizes the fastness concept from [GW76]. When conjoined with distributivity, a special case [KU76, Obs. 7] of fastness implies quick convergence of an iterative data flow algorithm [KU76, Thm. 2]. The more complicated elimination algorithm of [GW76, § 5] finds a good solution quickly whenever the context is fast and the graph is reducible. (The algorithm is easily extended to work for arbitrary rapid contexts.) See [Ta75] for a generalization of [GW76] that handles nonreducible graphs and [Ta76] for additional results on the complexity of iterative algorithms in distributive contexts.

Some contexts are not distributive, as when constant propagation uses compile time arithmetic [KU76, p. 167] or when elaborate analyses of the ranges of values of variables are exploited [Har77b]. One wants solutions better than fixpoints but cannot hope for optimal solutions [KU77, Thm. 7]. Like [GW76], [Ta75], our algorithm SOLVE finds a *good* solution: one at least as large as any fixpoint. In analyzing "structured" programs that avoid escapes and jumps, SOLVE does as well as [GW76] and sometimes better. By choosing the loop product @ in various ways, the implementor can trade time for sharpness of information. Roughly speaking, [GW76] corresponds to choices that emphasize speed. (Similarly for [Ta75].) For programs that fall within classical structured programming or that use escapes but not jumps, the time bounds for SOLVE and [GW76] are roughly similar. Because we did not assume $|\mathbf{E}| = 1$ in § 1 and because the basic heirarchy in § 3 is symmetric regarding entrances and exits, it is easy to adapt SOLVE to problems like the detection of dead variables, wherein information flows backwards along arcs and is given initially for exit nodes at the time of exit. Like most methods, that of [GW76] can be so adapted. The lack of entrance/exit symmetry makes the task more difficult [GW76, pp. 199, 200] than with SOLVE.

Contrary to what might be expected from the trend in programming complexity observed in going from [KU76] to [GW76] to [Ta75], SOLVE is a remarkably simple algorithm. This combination of power and simplicity is obtained by using a hierarchical representation of control flow instead of the usual large graph representing the entire program after translation to a relatively low-level intermediate text. Low-level methods solve a *problem* regardless of where it came from. High-level methods like SOLVE remember and exploit the structure of the *program* that gives rise to a problem. We have used structure expressed by the parse tree, but the general formulation of SOLVE and its correctness/cost theorem in § 4 are applicable to other hierarchies as well. High-level analysis also leads directly to concise but informative data flow diagnostics at source level [Ro77a, § 6]. In addition to the compiling applications we have emphasized, high-level representations of control flow are useful in denotational semantics [Ro77a, § 3] and program proving [La77, p. 140], [Ro76], [Ro77a, § 7].

The problem of efficiently finding a flow cover for any global flow scheme with a polycyclic graph has been left open here. Such schemes will be extremely rare in structured programming, even with the escapes and occasional jumps of practical structured programming. The problem is still worthwhile, and it could become urgent in an application where the hierarchy does not come from the parse tree. The techniques in [Ta75] may be useful here. Another open problem lacks a crisp mathematical formulation but is quite important. Many small examples are known where [GW76] or

SOLVE finds a good solution better than any fixpoint. But real compilers deal with programs, often very large ones, that are written to compute something rather than to illustrate the pros and cons of data flow analysis methods. In ambitious compilers with nondistributive data flow contexts, are these nonfixpoint solutions *significantly* better, in that they permit more extensive optimization? The traditional data flow contexts are idempotent as well as distributive. Data flow problems can be solved very quickly, but the answers are less informative than with ambitious contexts. But are they significantly less informative for optimization in the real world? Finally, we have assumed that $U \circ V$ or $U \wedge V$ can be found from U and V in "one" step: members of M can be represented so as to make the actual time required to find the representation of $U \circ V$ or $U \wedge V$ from representations of U and V be roughly constant. For some choices of M this idealization is reasonable [GW76, p. 178], but for others the sizes of the representations and hence the actual time will vary as U and V range over M. See [FKU75, § II] for an example where the time does vary. We conjecture that the actual members of M encountered for each real world program will have representations of roughly constant size, or at least that the variations will not invalidate comparisons between this paper and others that share the same idealization, such as [GW76], [Ta75]. Such open questions will not be answered simply YES/NO and will require a combination of theoretical and experimental investigation.

Ambitious compilers are *alert* to the new opportunities that one optimization creates for another, and they reserve their most elaborate and expensive optimizations for *selected* areas of a program that are critical to its run time behavior. Such areas could be determined by measuring the run time behavior after an unoptimized compilation [KS73], [Si78]. (The alertness and selectivity properties are abstracted from scattered hints in [AS78], [Ca77], [Har77a], [Har77b], [Kn74], [Lo77].) The theoretical literature on data flow analysis was inspired by an earlier compiler organization [A169], [LM69], in which analysis is followed by optimization in a fixed order that detects some new opportunities but misses others. A uniform degree of optimization is applied throughout. There is a fairly close relation between the total cost of data flow analysis in earlier optimizing compilers and the partial cost that is usually estimated: given one large global flow problem, how much does it cost to find a good solution to the problem? In order to be fully alert, however, ambitious compilers need to *update* the results of data flow analysis to reflect program changes. The updated information is only needed for the relatively critical areas of the program that are still being optimized. For noncritical areas (i.e. for most of the program, at least after the easiest initial optimizations) the compiler could save time by treating them as black boxes with known effects on data flow information. Noncritical areas could be treated like simple statements linking the selected critical areas. Specifically, consider a sequential compound statement of the form $[\alpha; \beta; \gamma]$, possibly within a loop. Suppose α and γ are critical, but not β. Suppose a costly computation **C** appears in γ and has results that are unaffected by β. (For example, **C** might use very high-level operators like **sort** (file) **on** (key) while β might never update or rearrange the file to be sorted.) Finally, suppose that α is optimized to α' with the byproduct that **C** becomes available on exit from α'. Then **C** is available on entrance to γ, as is obvious to us because we think hierarchically, with β considered as a single "statement" no matter how large it is. But a compiler using any of the usual data flow analysis methods will require at least $O(|\beta|)$ time to propagate the good news from the end of α' to the beginning of γ. (A partial exception is interval analysis [AC76], where a hierarchy specially constructed from a low-level representation may happen to resemble the parse tree. See the proof of Theorem 7.7 or [Ro77a, esp. p. 43] for more on the complicated relation between the usual low-level representations and the source text.) With high-level analysis such intuitive phrases as "the end of α'" or "the beginning of γ" have precise meanings, even when the program falls outside of classical structured programming. We can carry the good news from *leaving* α' to *entering* γ in one step by applying F_β(*entering* β, *leaving* β) to the information that **C** is available. The advantages of high-level analysis for updating are also sketched in [Ro77b, § 12].

We have tried to minimize the *total* cost of data flow analysis in ambitious compilers. Left open is the problem of estimating the cost of selective updating. The explicit time bounds in this paper deal only with the usual partial cost of data flow analysis: we solve flow problems singly without concern for selective updating. To study total cost thoroughly one would have to study the uses of data flow information and the sequencing of optimizations with the same mathematical thoroughness that is normally applied to solving flow problems singly but to no other aspect of global optimization. Babich and Jazayeri [BJ78b] suggest that total cost can be lowered by means of *demand* analysis that answers data flow questions as they arise in the course of optimization. This is selective updating pushed to its logical conclusion, and it deserves to be thoroughly studied. Our algorithm can be rewritten in a demand-driven form. Studies of the usual partial cost will continue to be important in the theory of data flow analysis, but the mathematics of total cost for demand analysis should gradually assume a theoretical prominence commensurate with its practical significance.

Acknowledgments. The author appreciates comments on the presentation by the anonymous referees and by A. L. Brown, J. L. Carter, P. Cousot, P. C. Goldberg, H. Holley, S. C. Johnson, J. D. Ullman, and M. N. Wegman.

REFERENCES

[Al69] F. E. Allen, *Program optimization*, Ann. Rev. in Autom. Prog., 5 (1969), pp. 239–307.

[AC76] F. E. Allen and J. Cocke, *A program data flow analysis procedure*, Comm. ACM, 19 (1976), pp. 137–147.

[AS78] M. A. Auslander and H. R. Strong, *Systematic recursion removal*, Ibid., 21 (1978), pp. 127–134.

[BJ78a] W. A. Babich and M. Jazayeri, *The method of attributes for data flow analysis* (*Part I. Exhaustive analysis*), Acta Informatica, 10 (1978), pp. 245–264.

[BJ78b] ———, *The method of attributes for data flow analysis* (*Part II. Demand analysis*), Ibid., 10 (1978), pp. 265–272.

[Ca77] J. L. Carter, *A case study of a new code generating technique for compilers*, Comm. ACM, 20 (1977), pp. 914–920.

[CK76] B. J. Cornelius and G. H. Kirby, *A programming technique for recursive procedures*, BIT, 16 (1976), pp. 125–132.

[CC77] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proc. 4th ACM Symp. on Principles of Programming Languages, January 1977, pp. 238–252.

[DDH72] O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, London-New York, 1972.

[FKZ76] R. Farrow, K. W. Kennedy and L. Zucconi, *Graph grammars and global program data flow analysis*, Proc. 17th IEEE Symp. on Foundations of Computer Science, October 1976, pp. 42–56.

[FKU75] A. Fong, J. Kam and J. D. Ullman, *Application of lattice algebra to loop optimization*, Proc. 2nd ACM Symp. on Principles of Programming Languages, January 1975, pp. 1–9.

[GRW77] H. Ganziger, K. Ripken and R. Wilhelm, *Automatic generation of optimizing multipass compilers*, Information Processing 77, B. Gilchrist, ed., North-Holland, Amsterdam, 1977, pp. 535–540.

[GW76] S. L. Graham and M. Wegman, *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23 (1976), pp. 172–202.

[Han77] M. Z. HANANI, *An optimal evaluation of Boolean expressions in an online query system*, Comm. ACM, 20 (1977), pp. 344–347.

[Har77a] W. H. HARRISON, *A new strategy for code generation—the general purpose optimizing compiler*, Proc. 4th ACM Symp. on Principles of Programming Languages, January 1977, pp. 29–37.

[Har77b] ———, *Compiler analysis of the value ranges for variables*, IEEE Trans. on Software Engineering, 3 (1977), pp. 243–250.

[HSU77] H. B. HUNT, T. G. SZYMANSKI AND J. D. ULLMAN, *Operations on sparse relations*, Comm. ACM, 20 (1977), pp. 171–176.

[KU76] J. B. KAM AND J. D. ULLMAN, *Global data flow analysis and iterative algorithms*, J. Assoc. Comput. Mach., 23 (1976), pp. 158–171.

[KU77] ———, *Monotone data flow analysis frameworks*, Acta Informatica, 7 (1977), pp. 305–317.

[Ke75] K. W. KENNEDY, *Node listings applied to data flow analysis*, Proc. 2nd ACM Symp. on Principles of Programming Languages, January 1975, pp. 10–21.

[Ki73] G. A. KILDALL, *A unified approach to global program optimization*, Proc. ACM Symp. on Principles of Programming Languages, October 1973, pp. 194–206.

[Kn74] D. E. KNUTH, *Structured programming with goto statements*, Computing Surveys, 6 (1974), pp. 261–302.

[KS73] D. E. KNUTH AND F. R. STEVENSON, *Optimal measurement points for program frequency counts*, BIT, 13 (1973), pp. 313–322.

[LA77] L. LAMPORT, *Proving the correctness of multiprocess programs*, IEEE Trans. on Software Engineering, 3 (1977), pp. 125–143.

[LM75] H. F. LEDGARD AND M. MARCOTTY, *A genealogy of control structures*, Comm. ACM, 18 (1975), pp. 629–639.

[Lo77] D. B. LOVEMAN, *Program improvement by source to source transformation*, J. Assoc. Comput. Mach., 24 (1977), pp. 121–145.

[LM69] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[MD76] P. MATETI AND N. DEO, *On algorithms for enumerating all circuits of a graph*, this Journal, 5 (1976), pp. 90–99.

[Ro76] B. K. ROSEN, *Correctness of parallel programs: the Church-Rosser approach*, Theoret. Computer Sci., 2 (1976), pp. 183–207.

[Ro77a] ———, *Applications of high-level control flow*, Proc. 4th ACM Symp. on Principles of Programming Languages, January 1977, pp. 38–47.

[Ro77b] ———, *High-level data flow analysis*, Comm. ACM, 20 (1977), pp. 712–724.

[Si78] R. L. SITES, *Programming tools: statement counts and procedure timings*, SIGPLAN Notices, 13 (12) (December 1978), pp. 98–101.

[SMR75] H. R. STRONG, A. MAGGIOLO-SCHETTINI AND B. K. ROSEN, *Recursion structure simplification*, this Journal, 4 (1975), pp. 307–320.

[TK76] K. TANIGUCHI AND T. KASAMI *An $O(n)$ algorithm for computing the set of available expressions of D-charts*, Acta Informatica, 6 (1976), pp. 361–364.

[Ta75] R. E. TARJAN, *Solving path problems on directed graphs*, Rept. STAN-CS-75-528, Computer Sci. Dept., Stanford U., November 1975.

[Ta76] ———, *Iterative algorithms for global flow analysis*, Rept. STAN-CS-76-547, Computer Sci. Dept., Stanford U., March 1976.

[Ul73] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[We75] B. WEGBREIT, *Property extraction in well-founded property sets*, IEEE Trans. on Software Engineering, 1 (1975), pp. 270–285.

[WFSW75] D. S. WISE, D. P. FRIEDMAN, S. C. SHAPIRO AND M. WAND, *Boolean valued loops*, BIT, 15 (1975), pp. 431–451.

[Wu75] W. A. WULF, ET AL., *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

[Za74] C. T. ZAHN, *A control statement for natural top-down structured programming*, Lecture Notes in Computer Sci., 19 (1974), pp. 170–180.

[ZB74] M. V. ZELKOWITZ AND W. G. BAIL, *Optimization of structured programs*, Software Practice and Experience, 4 (1974), pp. 51–57.

SIAM J. COMPUT.
Vol. 9, No. 1, February 2980

FINDING THE VERTEX CONNECTIVITY OF GRAPHS*

ZVI GALIL†

Abstract. New implementation of known algorithms improve the running time of the best known algorithms for finding the vertex connectivity of undirected and directed graphs.

Let G be a finite undirected graph, with no self-loops and no parallel edges with set of vertices V and set of edges E. A set of vertices, S, is called an *(a, b) vertex separator* if $\{a, b\} \subseteq V - S$ and every path connecting a and b passes through at least one vertex of S. Clearly, if a and b are connected by an edge, no (a, b) vertex separator exists. For a and b in V let $N^G(a, b)$ ($N(a, b)$ when G is known from the context) be defined as follows: If $(a, b) \in E$ then $N(a, b) = |V| - 1$, otherwise $N(a, b)$ is the least cardinality of an (a, b) vertex separator. The *vertex connectivity* of G, k_G, is defined to be $\min_{a,b \in V} N^G(a, b)$.

The best algorithm for computing the vertex-connectivity of a given graph is due to Even and Tarjan [4]. It is based on the following facts:

(1) k_G can be computed by finding $N(v, v')$ for at most k_G v's and all v''s in V.

(2) If $(a, b) \notin E$, $N(a, b)$ can be found by finding the maximum flow from a to b in a network obtained from G by defining the capacity of each edge and each vertex except a and b to be 1. Call such networks (with all edges and vertices of capacity 1) *special networks.*

(3) Dinic's algorithm [2] when applied to special networks takes at most $O(|V|^{1/2})$ phases the cost of each of which is $O(|E|)$.

So to compute k_G, we need to solve $O(k_G|V|)$ flow problems on special networks. Therefore, Even and Tarjan's method yields an algorithm with running time $T_1 = O(k_G|E|\,|V|^{3/2})$. If we take into account that $k_G \leqq 2|E|/|V|$ [4] this bound and the others below can be expressed in terms of $|V|$ and $|E|$ only.

In [3] Even solves the "easier" problem: Given an undirected graph G and an integer k, find whether $k_G \geqq k$. We denote this problem by $P_{G,k}$ and Even's solution to $P_{G,k}$ by $A_{G,k}$.

First we review $A_{G,k}$: Let $V = \{v_1, \cdots, v_{|V|}\}$, $V_j = \{v_1, \cdots, v_j\}$ and let $\tilde{G}_j$ be the graph obtained from G by introducing a new vertex a and connecting it to all the vertices in V_j.

Let

$$\alpha_k = \min_{u,v \in V_k} N^G(u, v),$$

$$\beta_k = \min_{k \leqq j < |V|} N^{\tilde{G}_j}(a, v_{j+1})$$

and

$$\gamma_k = \min(\alpha_k, \beta_k).$$

Even proved that $k_G \geqq k$ iff $\gamma_k = k$ (note that $\beta_k \leqq k$ because $N^{\tilde{G}_k}(a, v_{k+1}) \leqq k$). He computes γ_k by checking in at most $k^2 + |V|$ flow problems if the value of the maximum flow is at least k. In each flow problem he uses Ford and Fulkerson's algorithm [5] to

* Received by the editors June 19, 1978.

† Department of Mathematical Sciences, Computer Science Division, Tel-Aviv University, Ramat-Aviv, Tel-Aviv, Israel. This research was supported in part by the Israel Commission for Basic Research. Part of this work was done while the author was visiting the Mathematical Sciences Department at the IBM T. J. Watson Research Center, Yorktown Heights, New York.

look for at least k flow augmenting paths (f.a.p's), and it takes $O(|E|)$ steps to find one. Therefore Even claims a time bound of $O(k^3|E|+k|V|\,|E|)$. One can get slightly better results by using Dinic's algorithm to find whether the flow is at least k. The number of phases in Dinic's algorithm is $O(\min(k, |V|^{1/2}))$, and therefore $A_{G,k}$ can be implemented in time

$$T(k)=O((k^2+|V|)|E|\min(|V|^{1/2}, k))=O((\max(k, |V|^{1/2}))^2\min(|V|^{1/2}, k)|E|)$$
$$=O(\max(k, |V|^{1/2})k|V|^{1/2}|E|).$$

$T(k)$ is better than Even's bound if k is larger than $|V|^{1/2}$. Although for $k=O(|V|^{1/2})$ Even's bound and $T(k)$ are the same, it is probably better to use Dinic's algorithm because it may find many f.a.p.'s in one phase. (In fact P.M.G. Apers [1] has recently claimed that in the average the number of phases of Dinic's algorithm is bounded by a constant when applied to certain models of random networks with unit edge capacity.)

One can slightly simplify $A_{G,k}$: Let $\gamma'_k=\min(\alpha_k, \beta'_k)$, where $\beta'_k=\min_{k\leqq j<|V|} N^{\tilde{G}_k}(a, v_{j+1})$. One can show that $k_G\geqq k$ iff $\gamma'_k=k$. (Lemmas 1 and 2 in [3] still hold with L replaced by V_k.) This modification of $A_{G,k}$ which we denote by $A'_{G,k}$ is somewhat simpler because the $|V|-k$ flow problems for computing β_k are on $|V|-k$ different graphs, while the flow problems for computing β'_k are on the same graph that is the smallest of these $|V|-k$ graphs.

Let $T_2=T(k_G)$. One can use $A_{G,k}$ to find k_G: One way is to solve $P_{G,1}, P_{G,2}, \cdots$ until $P_{G,k+1}$ yields a negative answer and $k_G=k$. The naive implementation of this way yields an $O(k_GT_2)$ algorithm. An alternative way is to do a "careful binary search" in order to find k_G, each time solving $P_{G,k}$ with a well chosen k. By a careful binary search we mean the following procedure: first by doubling k find the smallest k which is a power of 2 such that $k\leqq k_G<2k$, and then find k_G by a binary search on the interval $[k, 2k]$. This approach yields an $O(\log k_GT_2)$ algorithm. Surprisingly, the first approach can be implemented in time $O(T_2)$.

Our solution B_G for finding k_G solves $P_{G,1}, P_{G,2}\cdots$. Assume we have solved $P_{G,k}$ and found that $k_G\geqq k$. At this time we execute at most k^2 Dinic's algorithms that compute $N^G(u, v)$ for all pairs u, v in V_k such that (u, v) is not an edge in G and $|V|-k$ Dinic's algorithms that compute $N^{\tilde{G}_i}(a, v_{j+1})$, $k\leqq j<|V|$, and all flows are at least k.

In order to find quickly which flow problem to solve next we maintain a set of queues $\{Q_i\}$. At a certain time, the pair (a, b) is in Q_i if at that time the flow in the corresponding problem is i. It is obvious how to maintain these queues so that the overhead is $O(|V|)$.

We now explain how to solve $P_{G,k+1}$ after having solved $P_{G,k}$.

(i) We execute enough phases of Dinic's algorithm in each of the flow problems that correspond to the pairs (v, v_{k+1}) for $v\in V_k$ such that $(v, v_{k+1})\notin E$ until the flow in each is at least k, and then put each pair in the corresponding queue.

(ii) Except the pair (a, v_{k+1}) that is deleted from Q_k we execute a phase of Dinic's algorithm for the pairs (u, v) in Q_k and then put them in the appropriate queues.

We say that a step succeeds if each phase in Dinic's algorithm executed in it leads to a flow increase. Step (i) must succeed by the correctness of $A_{G,k}$. If step (ii) succeeds, then $\alpha_{k+1}\geqq k+1$, and $\beta_{k+1}=k+1$ and thus $k_G\geqq k+1$. Also, as a result we now have the initial conditions for $P_{G,k+2}$. If step (ii) fails, then $k_G=k$, and we stop.

The total time bound is $O(T_2)$ since at most $(k_G+1)^2+|V|$ flow problems are solved by Dinic's algorithm. Note that unless $k_G=\theta(|V|)$, T_2 is always smaller than T_1. In particular when $k_G=O(|V|^{1/2})$ the factor of improvement is $|V|^{1/2}$. When $k_G=\theta(|V|)$, T_2 and T_1 are comparable, but are no better than the time bound of the obvious algorithm that computes $N^G(u, v)$ for all u and v. Note also that by running the $k_G|V|$ Dinic's algorithms that arise in Even and Tarjan's algorithm in parallel (in a similar way to B_G), the bound T_1 can be improved to $T'_1 = O(k_G|V|\,|E| \min(k_G, |V|^{1/2}))$. T_2 is still smaller than T'_1 unless $k_G = O(1)$ or $k_G = \theta(|V|)$ in which case they are comparable. In particular when $k_G = \theta(|V|^{1/2})$ the factor of improvement is still $|V|^{1/2}$.

The price we pay is that we must use extra space. While Even and Tarjan's algorithm uses $O(|E|)$ space because it solves each flow problem in turn, B_G may solve simultaneously $\theta(k_G^2 + |V|)$ flow problems, and as a result the naive implementation of B_G uses $O((k_G^2 + |V|)|E|)$ space. M. Ben–Ari has pointed out to us that one can save space by using the following trick: Maintain just one copy of G, and in each of the $k_G^2 + |V|$ flow problems represent an edge by a bit (1 if it is saturated, 0 otherwise). The reader can verify that B_G can be implemented with essentially no time loss and with using $O(|E|)$ registers to represent G plus $O((k_G^2 + |V|)|E|)$ bits.

One can obtain a modified algorithm B'_G from $A'_{G,k}$ in the same way B_G is derived from $A_{G,k}$. To compute β'_k we need to solve $|V| - k$ flow problems on $\tilde{G}_k$ and when k is increased by 1 we have to add an edge to $\tilde{G}_k$ to get $\tilde{G}_{k+1}$. So, to compute β'_{k+1} we look for an additional f.a.p. in $|V| - k - 1$ flow problems. Consequently, B_G is simpler than B'_G because in the latter when k changes the graphs change, and we lose the advantage of using Dinic's algorithm for computing β_k. (Note that although $A'_{G,k}$ was simpler than $A_{G,k}$, B_G is simpler than B'_G).

The case of directed graphs is similar: Even and Tarjan's algorithm can be slightly modified to apply to directed graphs because facts 1)–3) still hold in this case [4]. Even's algorithm for solving $P_{G,k}$ can also be modified as follows [3]: Let $\tilde{G}_j^{\rightarrow}$ [$\tilde{G}_j^{\leftarrow}$] be $\tilde{G}_j$ with all new edges directed from [to] the new vertex a and let

$$\vec{\gamma} = \min(\alpha_k, \vec{\beta}_k) \quad \text{where} \quad \vec{\beta}_k = \min\left[\min_{k \leqq j < |V|} G^{\tilde{G}_j^{\rightarrow}}(a, v_{j+1}), \min_{k \leqq j < |V|} N^{\tilde{G}_j^{\leftarrow}}(v_{j+1}, a)\right]$$

and $k_G \geqq k$ iff $\vec{\gamma}_k = k$. So it is obvious how to modify $A_{G,k}$ (or $(A'_{G,k})$) and B_G so that they apply to directed graphs.

Finally, we make some comments on the easier problem of finding edge connectivity. Even and Tarjan's algorithm [4] takes $O(\min(|E|^{1/2}, |V|^{2/3})|V|\,|E|)$ time and Schnorr's algorithm [6] takes $O(k|V|\,|E|)$ time, where k is the edge connectivity. The two algorithms can be combined to yield an $O(\min(k, |V|^{2/3})|V|\,|E|)$ algorithm.

Acknowledgment. I am grateful to S. Even, A. L. Rosenberg and C. P. Schnorr for their helpful suggestions.

REFERENCES

[1] P. M. G. APERS, *Average case analysis of Dinic–Karzanov network flow algorithm*, Tech. Rep. HP 77-7-001, Dept. of Information Sciences, Univ. of California, Santa Cruz, June 1977.

[2] 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.

[3] S. EVEN, *An algorithm for determining whether the connectivity of a graph is at least k*, this Journal, 4 (1975), pp. 393–396.

[4] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 507–518.

[5] L. R. FORD AND D. R. FULKERSON, *Flow in Networks*, Princeton Press, Princeton, NJ, 1962.

[6] C. P. SCHNORR, *Multiterminal network flow and connectivity in unsymmetrical networks*, Proc. 5th Colloquium on Automata Languages and Programming (Udine, July 1978), Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 425–439.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0017 $01.00/0

VORONOI DIAGRAMS IN L_1 (L_∞) METRICS WITH 2-DIMENSIONAL STORAGE APPLICATIONS*

D. T. LEE† AND C. K. WONG‡

Abstract. In this paper we study the problem of scheduling the read/write head movement to handle a batch of n I/O requests in a 2-dimensional secondary storage device in minimum time. Two models of storage systems are assumed in which the access time of a record (being proportional to the "distance" between the position of the record and that of the read/write head) is measured in terms of L_1 and L_∞ metrics, respectively. The scheduling problem, referred to as the Open Path Problem (OPP), is equivalent to finding a shortest Hamiltonian path with a specified end point in a complete graph with n vertices. We first show in this paper that there exists a natural isometry between the L_1 and L_∞ metrics. Consequently, the existence of a polynomial time algorithm for the OPP in one metric implies the existence of a polynomial time algorithm for the same problem in the other metric. Based on a result by Garey, Graham and Johnson, it is easy to show that the OPP in L_1 (hence in L_∞) metric is *NP*-complete. A heuristic to solve the OPP is therefore presented. It is based on a geometric structure called the Voronoi diagram in L_1 metric. An optimal (worst-case) algorithm of time complexity $O(n \log n)$for constructing the diagram for a set of n points in a plane is described. Using this diagram one can build a near-optimal path through each point either by constructing a minimum spanning tree or by the closest insertion method.

Both algorithms are shown to take $O(n \log n)$ time which is the time for the construction of the diagram and yield an approximate solution within a factor of 2. The bound is also shown to be tight in the worse case. For the average case, simulation results show that the minimum spanning tree approach is better than the closest insertion method. As expected, they are far better than the sequential one in which the request is processed one at a time on the first-come–first-served basis.

Key words. L_1 metric, L_∞ metric, L_p metric, Voronoi diagram, minimum spanning tree, closest insertion method, nearest neighbor method, scheduling of read/write heads, 2-dimensional storage medium, *NP*-completeness, near-optimal algorithm, approximate algorithm, worst-case analysis, average case performance, open path problem, computational geometry, complexity of algorithm, Hamiltonian path, traveling salesman problem

1. Introduction. A major technological trend for large database systems is the introduction of mass storage. This allows computing centers to maintain on-line their program libraries, less frequently used data files, backup copies, etc. under unified system control.

In this paper, we consider two kinds of mass storage systems. Both of them are 2-dimensional arrays, which can be represented by the grid points (x, y) in a plane where x, y are integers and $1 \leqq x, y \leqq N$. A record is stored at each grid point. In the first kind of system records are accessed by an electromechanical fetching mechanism, known as the read/write head. The time for the read/write head to move from point (u, v) to a point (x, y) is proportional to $\max(|x-u|, |y-v|)$. The second kind is a magnetic bubble device, where the time to move the read/write head from (u, v) to (x, y) is measured by $|x-u|+|y-v|$. (Section 2 contains a more detailed description.) It is assumed in both systems that the read/write head, after accessing a record, will

* Received by the editors October 6, 1977, and in revised form April 4, 1979.

† Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201. The work of this author was supported in part by the National Science Foundation under Grant MCS-76-17321 and by the Joint Service Electronics Programs under Contract DAABH07-72-C-0259. Part of this work was done while the author was a summer visitor at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York from the University of Illinois at Champagne-Urbana.

‡ Department of Electrical Engineering and Computer Science, Columbia University, New York, New York 10027, on leave from IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

remain at the position of the record until the next request is issued. In [12], [22], the first kind of storage system is studied. It is assumed, however, that requests to records are processed sequentially, i.e. one request at a time on the first-come–first-served basis. The problem considered there is mainly the assignment of records to grid points based on request probabilities so that the expected access time between consecutive requests is minimized. 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 [19] and is omitted here. (For a study of the same problem, but in a linear storage, we refer to [23]).

Under this model, two problems arise immediately. First, for a given batch of requests, how do we schedule the head movement so that it goes through the requested records in minimum time (distance)? The distance is measured as the total length of the path starting with the position of the last record processed in the previous batch, and going through each requested record once. This problem is equivalent to finding a shortest Hamiltonian path with a fixed starting point. The problem will be referred to as the Open Path Problem (OPP). The second problem is the assignment problem of records as in [12], [22] except that batched processing is assumed. More specifically, assume the request probabilities of the records are known. Let π be an assignment of the records to the N^2 grid points and let B be the batch size. Let A be an algorithm for the OPP with the expected total distance $D_B(\pi)$. The objective is to find π such that $D_B(\pi)$ is minimized. (See also § 4.) The second problem seems to be very difficult and requires further research. We therefore assume in this paper that all request probabilities are equal and the assignment problem disappears.

In § 2, we shall define L_p metric, $1 \leqq p \leqq \infty$ and we shall describe our basic models in greater details and show that the two systems are equivalent as far as the computational complexity is concerned. Thus, only the second kind of system will be considered further.

In practice, we cannot afford to spend too much time in solving the OPP. Unfortunately, in § 3 this problem is shown to be NP-complete in the sense of Cook [4] and Karp [8], thus necessitating the consideration of efficient heuristics. Our proposed heuristics are based on the construction of Voronoi diagrams [15], [18] in L_1 metric. For other applications of the Voronoi diagram, see [18]. Using this diagram one can build a near-optimal path through the given set of points either by constructing a minimum spanning tree or by closest insertion method as discussed in [16]. We show that the construction of the Voronoi diagram for n points in a plane in L_1 (L_∞) metric takes time $O(n \log n)$. In [14], [18] an $O(n \log n)$ algorithm for constructing the Voronoi diagram for n points in a plane in L_2 (Euclidean) metric is presented. While our construction follows the same basic approach as in [14], [18], it needs some nontrivial modifications. Some new observations are also needed in order to achieve the bound $O(n \log n)$.

As a passing remark, we tried without success to construct Voronoi diagrams in L_p metric, $3 \leqq p < \infty$, in $O(n \log n)$ time. This is yet another example of the phenomenon noted in [12], [20] that L_1, L_∞ metrics are much harder to study than L_2 metric, while L_p metrics, $3 \leqq p < \infty$, are just plain impossible. It should also be pointed out that even the intuitively appealing heuristic of moving to the nearest neighbor takes $O(n^2)$ time for a general metric. It is not known if the construction time can be cut down to $O(n \log n)$ in L_1 (L_∞) metric.

In § 4, we consider expected performance of our heuristic. Unfortunately, no significant analytic results have been obtained. Instead, we present simulation results for various grid sizes and batch sizes.

2. Basic models and equivalence of L_1 and L_∞ metrics. We first define L_p metrics. Given two points q_i and q_j in the plane R^2 with coordinates (x_i, y_i) and (x_j, y_j), respectively, the distance between them is defined as $d_p(q_i, q_j) = (|x_i - x_j|^p + |y_i - y_j|^p)^{1/p}$ for $p = 1, 2, \cdots$, and $d_\infty(q_i, q_j) = \max(|x_i - x_j|, |y_i - y_j|)$. We shall denote the plane in which L_p metric is the distance measure by R_p^2.

In the first kind of storage system, the records are accessed by an electromechanical fetching mechanism which has equal and constant moving speeds along the x-axis and y-axis. Suppose the fetching mechanism moves from point (x_i, y_i) to point (x_j, y_j); then the path it takes consists of two segments: a straight line at 45° to the x-axis followed by either a horizontal or a vertical line segment. The time it takes to complete the journey is therefore proportional to $\max(|x_i - x_j|, |y_i - y_j|)$. To see this, assume without loss of generality that $x_i = y_i = 0$, $y_j > x_j > 0$. Then the path consists of two line segments: one at 45° and of length $\sqrt{2}x_j$, the other being vertical and of length $y_j - x_j$. If the speed in the horizontal and vertical directions is 1, then the diagonal movement has speed $\sqrt{2}$. The total distance is therefore $(\sqrt{2}x_j)/\sqrt{2} + (y_j - x_j) = y_j$. It therefore corresponds to the L_∞ metric.

The second kind of storage system is the magnetic bubble device described in [3]. Intuitively we have a square array of records, and the read/write head, which is a magnetic sensor, can only move along the grid lines. For example, to move from (x_i, y_i) to (x_j, y_j) where $x_i < x_j$, $y_i < y_j$, the head moves first horizontally from (x_i, y_i) to (x_j, y_i), then vertically from (x_j, y_i) to (x_j, y_j). Thus, the total distance traveled is $|x_i - x_j| + |y_i - y_j|$. It therefore corresponds to the L_1 metric.

To show the equivalence of the L_1 and L_∞ metrics, we define a linear mapping f from the plane R_∞^2 to the plane R_1^2 as follows. For any point $(x, y) \in R_\infty^2$ we have $(x', y') \in R_1^2$ where $x' = (y + x)/2$ and $y' = (y - x)/2$. The distance between two points q_i and q_j in R_∞^2 can be shown to equal the distance between their images q_i' and q_j' in R_1^2. Suppose the coordinates of q_i and q_j are (x_i, y_i) and (x_j, y_j) respectively and $x_j > x_i$, $y_j > y_i$ and $x_j - x_i > y_j - y_i$. The distance between q_i' and q_j' in R_1^2 is given by $d_1(q_i', q_j') = ((y_j - y_i)/2 + (x_j - x_i)/2) + ((x_j - x_i)/2 - (y_j - y_i)/2)$ which is equal to $x_j - x_i = d_\infty(q_i, q_j)$. Thus, f preserves the distance, i.e. it is an isometry. Figure 1 is an example which

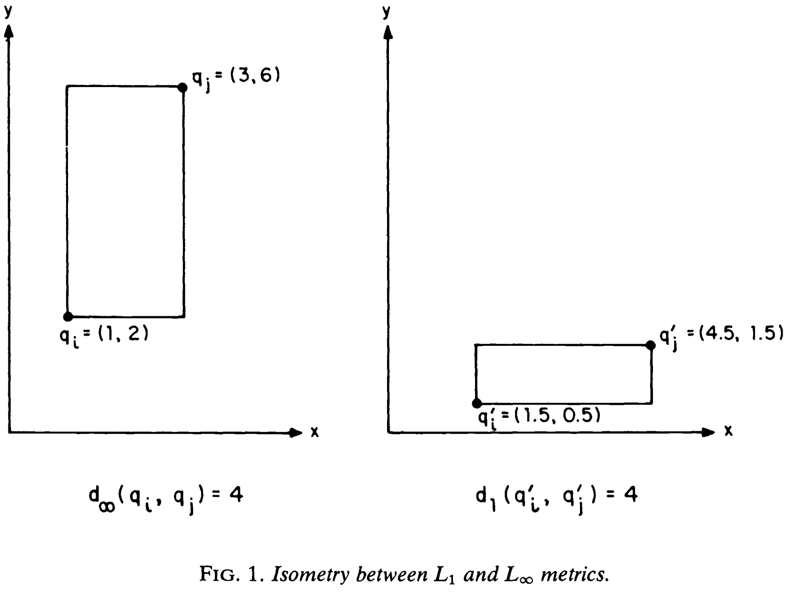


FIG. 1. *Isometry between L_1 and L_∞ metrics.*

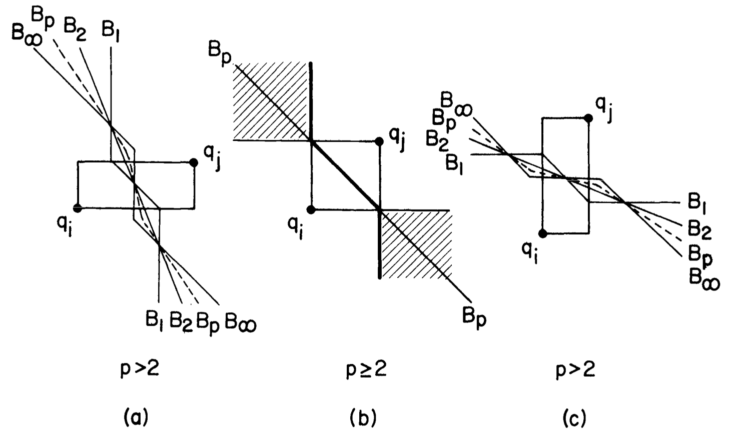
illustrates the isometry between these two metrics. The inverse mapping f^{-1} can also be shown to be an isometry. As a result, the Open Path Problem (OPP) in the L_∞ metric is polynomially equivalent to that in the L_1 metric in the sense that any deterministic polynomial time algorithm for the problem in the L_∞ metric can be applied to solve the same problem in the L_1 metric and vice versa. Throughout this paper, we therefore consider only the L_1 metric.

3. Approximation algorithms for the open path problem.

3.1. *NP*-completeness of the open path problem. In a recent paper [5], the traveling salesman problem for points in R_1^2 has been shown to be *NP*-complete. By a slight modification of the proof it can be shown that the so-called "open" traveling salesman problem, i.e. finding a shortest Hamiltonian path without specifying end points, for points in R_1^2, is also *NP*-complete [7]. To show that the OPP considered here is *NP*-complete, we note first that it is obviously in *NP*. Since the open traveling salesman problem is *NP*-complete, the existence of a polynomial time algorithm for this problem (i.e. $NP = P$) implies the same for the OPP. On the other hand, if the OPP has a polynomial time algorithm, we can apply it n times to solve the open traveling salesman problem in polynomial time by specifying in turn each point as a starting point and then taking the shortest path of the n solutions.

Due to the difficulty of obtaining an optimal solution to this problem, we shall present two approximation algorithms which run in $O(n \log n)$ time where n is the number B of batched requests plus 1 and yield a solution within a factor of 2, of the optimal.

3.2. Construction of Voronoi diagrams in L_1 metric. We first introduce some definitions and notations. Given two points q_i and q_j with coordinates (x_i, y_i) and (x_j, y_j), respectively, in the plane R_p^2, the bisector $B_p(q_i, q_j)$ of q_i and q_j is the locus of points equidistant from q_i and q_j, i.e. $B_p(q_i, q_j) = \{r \mid r \in R_p^2, d_p(r, q_i) = d_p(r, q_j)\}$. In the Euclidean plane, $B_2(q_i, q_j)$ is the perpendicular bisector of the line segment $\overline{q_iq_j}$. The bisectors of q_i and q_j in different metrics are shown in Fig. 2. If $|x_i - x_j| > |y_i - y_j|$ then $B_1(q_i, q_j)$ has two vertical lines and one line segment (Fig. 2a). If $|x_i - x_j| < |y_i - y_j|$ then $B_1(q_i, q_j)$ has two horizontal lines and one line segment. (Fig. 2c). They are referred to as vertical and horizontal bisectors respectively for short. In the case that $|x_i - x_j| = |y_i - y_j|$, $B_1(q_i, q_j)$ has two unbounded regions (crossed area in Fig. 2b) and a line segment. Without creating any significant difference in the following discussion, we shall arbitrarily choose



p>2 (a)

p≥2 (b)

p>2 (c)

FIG. 2. *Bisectors in different metrics.*

the vertical bisector (the thick lines) as $B_1(q_i, q_j)$. Note that the line segment portion of $B_1(q_i, q_j)$ is of slope ±1.

We now define the generalized notion of Voronoi diagrams [15], [18] in L_p metrics. Given a set $\mathcal{S} = \{q_1, q_2, \cdots, q_n\}$ of n points in R_p^2, the locus of points closer to q_i than to q_j, denoted by $h_p(q_i, q_j)$, is one of the "half-planes" determined by the bisector $B_p(q_i, q_j)$, i.e. $h_p(q_i, q_j) = \{r | r \in R_p^2, d_p(r, q_i) \leqq d_p(r, q_j)\}$. The locus of points closer to q_i than to any other point, denoted by $V_p(q_i)$ is thus given by $V_p(q_i) = \bigcap_{i \neq j} h_p(q_i, q_j)$, the intersection of all the half-planes associated with q_i. The region $V_p(q_i)$ is called the *Voronoi polygon* (not necessarily bounded) associated with q_i. The entire set of regions partitions the plane into n regions and is referred to as the *Voronoi diagram* $V_p(\mathcal{S})$ for the set $\mathcal{S}$ of points in L_p metric. The points at which three or more bisectors meet are called *Voronoi points*. We shall refer to the portion of a bisector between two Voronoi points of a Voronoi polygon as an *edge* of the polygon. An edge of the Voronoi polygon in $L_1(L_\infty)$ metric can have at most three line segments. Since we shall deal only with the L_1 metric, unless specified otherwise, the subscript p denoting the metric will be dropped without any confusion. The *body* $HB(\mathcal{S})$ of the set $\mathcal{S}$ is defined as the smallest rectangular region that contains the entire set $\mathcal{S}$. Since any two points of $\mathcal{S}$ define a unique rectangle, the body of the set $\mathcal{S}$ can also be defined as $HB(\mathcal{S}) = \bigcup_i \{HB(\mathcal{S}_i) | \mathcal{S}_i = \{q, r\}, q, r \in \mathcal{S}\}$, i.e. the union of the bodies of all its 2-subsets (subsets of $\mathcal{S}$ with cardinality 2.) In particular, if $\mathcal{S} = \{q_1, q_2\}$ where q_1, q_2 lie on a line parallel to x- or y-axis, the body $HB(\mathcal{S})$ is the line segment $\overline{q_iq_2}$. Figure 3 shows the Voronoi diagram for a set of 9 points, in which "Δ" denotes a Voronoi point, and the region within the dashed rectangle is the body.

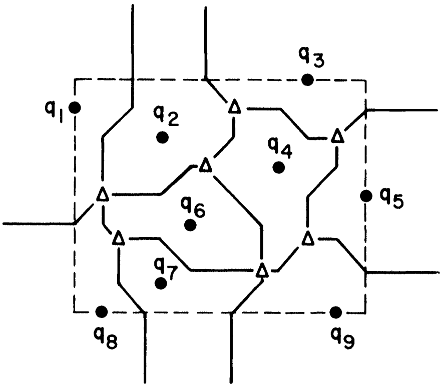


FIG. 3. *Voronoi diagram and the body of set* $\mathcal{S}$.

Several observations need to be made of the Voronoi diagram. First of all, the dual of the diagram is a *planar* graph on the set of n points in which there is an edge between q_i and q_j if the Voronoi polygons $V(q_i)$ and $V(q_j)$ have a common edge, $B(q_i, q_j)$ as their border. Since there is a one-to-one correspondence between an edge of the dual and a bisector of the diagram, and a one-to-one correspondence between a region of the dual and a Voronoi point, the number of bisectors and Voronoi points are both $O(n)$. The minimum spanning tree in L_1 metric can be shown to be a subgraph of the dual of the Voronoi diagram $V(\mathcal{S})$. The proof parallels that of showing the Euclidean minimum spanning tree being embedded in the dual of $V_2(\mathcal{S})$ [18] and hence is omitted here. Secondly, the closest pair of points can be found in $O(n)$ time if the diagram is available. Since the proof that finding the closest pair of points in L_2 metric requires Ω (n log n)

time in the worst case [18] can be carried over to any metric, the construction of the Voronoi diagram must take at least $\Omega(n \log n)$ time. Thirdly, the diagram outside the body of the set (see Fig. 3) consists only of vertical and horizontal lines. By the definition that the body of the set $\mathcal{S}$ is the union of the bodies of all its 2-subsets and the fact that the bisector $B(q, r)$ of any two points q and r in $\mathcal{S}$ is either vertical or horizontal, this observation follows immediately. We note in passing that the last observation is particularly important in the following discussions.

The Voronoi diagram in L_1 metric (or metrics other than L_2) is different from the commonly known Voronoi diagram in Euclidean metric in that the Voronoi polygons in the former diagram are not *convex.* Convexity plays an important role in the construcure of $V_2(\mathcal{S})$ and makes the lower bound $\Omega(n \log n)$ achievable [14], [18]. In the procedure given in [18], the merge process can be accomplished in $O(n)$ time by using the property of convexity of the Voronoi polygon [14]. It is not apparent at all that the same argument can be carried through directly if convexity is no longer available. That is, the question of whether or not the Voronoi diagram in L_p metric can be constructed in $O(n \log n)$ time still remains unsettled. We shall show in the following that the Voronoi diagram in L_1 metric can be constructed in $O(n \log n)$ time by using the above-mentioned properties.

Construction of the Voronoi diagram in L_1 metric. We shall use divide-and-conquer technique to construct the Voronoi diagram $V(\mathcal{S})$. First of all, we presort the data in ascending order of the x-coordinates (and y-coordinates if x-coordinates are equal) and number them 1 through n from left to right. Divide the set $\mathcal{S}$ into two disjoint subsets L and R which contain the leftmost and the rightmost $n/2$ points respectively. Recursively construct the Voronoi diagrams $V(L)$ and $V(R)$ for sets L and R respectively. We shall merge them to form $V(\mathcal{S})$ by constructing a polygonal line T with the property that any point to the left of the line is closer to some point in L and any point to the right is closer to some point in R. After the line is constructed, the portion of $V(L)$ (and $V(R)$) that lies to the right (and left) of the line is discarded, and the resultant $V(\mathcal{S})$ is obtained. Note that the line T can be shown [17] to be monotone with respect to y-axis, i.e. for any three points a, b and c on T, their y-coordinates satisfy either $y(a) \geqq y(b) \geqq y(c)$ or $y(a) \leqq y(b) \leqq y(c)$. If one can show that $O(n)$ time suffices to construct the line, then it follows by the recurrence relation $T(n) = 2T(n/2) + O(n)$ that $T(n) = O(n \log n)$ is enough for the construction of the diagram $V(\mathcal{S})$.

The example shown in Fig. 4 helps illustrate the idea of merging. Consider a set of 18 points numbered from 1 through 18. The left set is $L = \{1, 2, \cdots, 9\}$ and right set is $R = \{10, \cdots, 18\}$. The Voronoi diagrams $V(L)$ and $V(R)$ are shown in short and long dashed lines respectively. The merge process is described as follows. At first, the rightmost point with the smallest index in the set L is found and is denoted by w. Since w is the rightmost point in L, w is on the boundary of the body $HB(L)$ and its associated Voronoi polygon $V(w)$ is unbounded. Consider the horizontal half-line emanating from the point w to infinity in the same direction as the positive x-axis and denote it by $\mathbf{w}$. Take any point z on $\mathbf{w}$. It is easy to verify that $d(z, w) = \min_{u \in L} d(z, u)$, which implies that the entire half-line $\mathbf{w}$ is contained in the polygon $V(w)$. Therefore $V(w)$ is unbounded.

The following process of determining the *starting bisector* is based on the property that any line connecting two points which lie on different sides of T must intersect T. The half-line $\mathbf{w}$ contains a point z whose x-coordinate is sufficiently large satisfying the inequality $d(z, w) > d(z, v)$ for some v in R. That is, z lies to the *right* of T. Since w is known to lie to the *left* of T, an intersection point of the line segment $\overline{wz}$ and T is

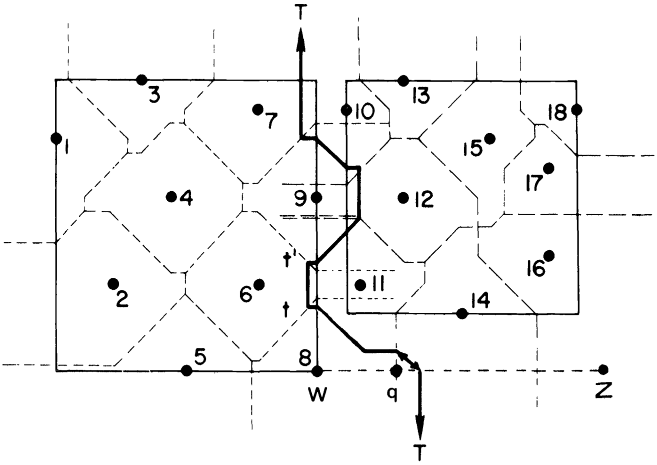


FIG. 4. *Example illustrating merging.*

guaranteed to exist. Since T is a collection of bisectors $B(u, v)$ for some u in L and v in R, we shall look for the bisector $B(w, s)$ for some s in R such that it intersects the line segment $\overline{wz}$. We first find the nearest neighbor r of w among the set R, i.e. $d(w, r) = \min_{v \in R} d(w, v)$, i.e., points 8 and 11, respectively, in Fig. 4. This step takes $O(N)$ time. r being the nearest neighbor of w, its associated polygon $V(r)$ contains w. We shall scan the edges of $V(r)$ to find which edge the line segment $\overline{wz}$ intersects. Suppose $\overline{wz}$ intersects an edge $B(r, r')$ at a point q. If q is found to lie to the *right* of T, i.e. $d(q, r) < d(q, w)$, then T intersects the line segment $\overline{wq}$ at some point t and $B(w, r)$ is our starting bisector. Otherwise we do the same by scanning the edges of $V(r')$ to find which edge intersects $\overline{qz}$. Repeat the process until we either find an intersection point which lies to the *right* of T or fail to find one. In either case, we shall use $B(w, s)$ as our starting bisector, where s is the point whose associated polygon $V(s)$ is currently under consideration. The time involved to find the starting bisector is $O(N)$, for each edge of $V(R)$ is examined at most once. As shown in Fig. 4, $B(8, 14)$ is the starting bisector, and the construction of T shown in solid lines is carried out in two phases, upward and downward, both using $B(8, 14)$ as the starting bisector.

We remark here that this step for determining the starting bisector of the polygonal line differs from that in constructing the line in L_2 metric. In the latter, the starting bisector is determined by the line segments created in forming the union of the convex hulls of L and R [14], [18]. We first proceed by moving upward an imaginary point t following the direction of $B(8, 11)$ until we meet $B(8, 6)$ at which point, since t is closer to 6 than to 8, we follow the bisector $B(6, 11)$ and so on. continue moving upward until we go out of the body $HB(S)$. The downward phase is similar.

We now show that the construction of T takes $O(n)$ time. Before we proceed, we make the following observation. At any time during the merge process, the imaginary point t always lies in two Voronoi polygons, one in $V(L)$, the other in $V(R)$. Whenever a new Voronoi point is created, we will enter a new Voronoi polygon and follow a new direction determined as follows. Suppose $B(a, b)$ was the bisector that we followed in constructing the polygonal line T and $B(b, c)$ was the bisector that intersects $B(a, b)$. The new direction will be following $B(a, c)$. As one can see, at each step we must determine which edge, of the two Voronoi polygons where the imaginary point currently lies, intersects the current polygonal line first. To do this we shall use the following scanning scheme, i.e. scan the edges of the polygon in $V(L)$ in *counterclockwise* direction and those of the polygon in $V(R)$ in *clockwise* direction [14]. This scheme is crucial to make the construction of the polygonal line accomplishable in $O(n)$ time and will be justified later.

Note that when we are in the left (right) body and determining which edge of the polygon in $V(L)$ $(V(R))$ intersects the polygonal line, the only lines that can interfere with the process are those *horizontal* lines emanating from the opposite body; and those horizontal lines are in order of y-coordinate. Thus, during this process some horizontal lines may be *visited* several times, but the total number of visits is proportional to the number of Voronoi points created. This can be seen as follows. Referring to Fig. 4, suppose we are at the point t in the *left* body $HB(L)$ and in Voronoi polygons $V(6)$ and $V(11)$. The bisector of $V(6)$ which intersects the polygonal line T (i.e. $B(6, 11)$) is determined by examining in counterclockwise fashion the two end points (Voronoi points) of each bisector of the polygon until they lie on different sides of the line T. And the intersecting point of T and the horizontal line from $HB(R)$ (the double-dashed line in Fig. 4) is also determined. A simple comparison decides the first intersecting point. A new Voronoi point will be created and we will enter a new Voronoi polygon. The same process is repeated. For example, t' is created and we enter polygon $V(9)$. If T meets an edge of the polygon first and we still are in the left body $HB(L)$, the same horizontal line will be re-visited again in the next iteration. Thus, the number of times that the horizontal line is visited is proportional to the number of Voronoi points created.

Since the polygonal line is monotone with respect to y-axis, at some point it will either meet the horizontal line first or go out of the body $HB(L)$. If T goes out of the body $HB(L)$ and enters the body $HB(R)$, we have a similar situation as before except we change "left" to "right" and "counterclockwise scan" to "clockwise scan". The case in which T meets the horizontal line first needs more careful investigation. Recall that we are at some point t in the left body $HB(L)$. Suppose that before we find the point t' where T intersects the edge of the polygon, we are interfered with by a number of

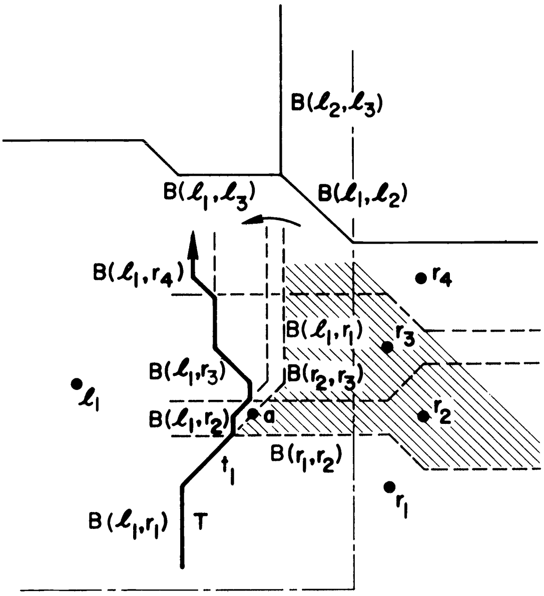


FIG. 5. *T meets the horizontal line first.*

horizontal lines emanating from $HB(R)$. For detailed illustration see Fig. 5, where we are following bisector $B(l_1, r_1)$ and there are a number of interfering horizontal lines $B(r_1, r_2)$, $B(r_1, r_3)$, etc.

We claim that some edge of the polygon in $V(L)$ will be visited several times and once it is eliminated from consideration it will never be visited again, i.e. no backtracking is needed. In Fig. 5, we have the situation that T meets the horizontal line first. Before we know that $B(l_1, r_1)$ meets $B(r_1, r_2)$ first, we have visited $B(l_1, l_2)$ once. At point t_1 which is a new Voronoi point on T, we follow the new bisector $B(l_1, r_2)$. Again, $B(l_1, l_2)$ will be revisited. Since both ends of $B(l_1, l_2)$ lie on the same side of $B(l_1, r_2)$, it is eliminated. The next edge $B(l_1, l_3)$ is visited. Since $B(l_1, r_2)$ meets $B(r_2, r_3)$ before it meets $B(l_1, l_3)$, another Voronoi point t_2 is created and a new bisector $B(l_1, r_3)$ is formed. As we can see, each time a new bisector is formed, it is on the left side of the previous bisector, i.e. the portion of $B(l_1, r_3)$ in T is to the left of the bisector $B(l_1, r_2)$; the portion of $B(l_1, r_2)$ in T is to the left of $B(l_1, r_1)$. Thus, the possible intersecting edge of the polygon $V(l_1)$ moves in a counterclockwise direction. Therefore, no backtracking is needed. To see this, it is sufficient to show that, for example, any point a on $B(l_1, r_1)$, where $y(a) > y(t_1)$, lies on the right side of the new bisector $B(l_1, r_2)$. Since $d(a, r_2) < d(a, r_1) = d(a, l_1)$, it follows that a lies on the right side of $B(l_1, r_2)$. In other words, the dotted-line portion of $B(l_1, r_2)$ is on the left side of $B(l_1, r_1)$. A similar situation occurs if we are in the right body except that the direction of the movement is *clockwise*. This also justifies the scanning scheme mentioned above.

Thus, the number of edge visits in $V(l_1)$ is proportional to the number of edges of $V(l_1)$ and the number of Voronoi points created. Since the polygonal line is monotone with respect to y-axis, it will eventually meet some edge of the polygon $V(l_1)$ and enter a new polygon. At the point, the same process is repeated. Thus, the total number of edge visits in constructing the polygonal line is proportional to the number of bisectors plus the number of Voronoi points on T. Since both of them are $O(n)$, the time for the construction of the polygonal line is $O(n)$. This completes the description of the merge process and verifies that the total construction time for the Voronoi diagram is $O(n \log n)$, which is optimal in the worst-case sense.

3.3. Approximation algorithms.

3.3.1. Minimum spanning tree method. Recall that the MST is a subgraph of the Voronoi dual. After we have constructed the Voronoi diagram, we can find its dual in $O(n)$ time. Now, we can apply any known minimum spanning tree algorithm [2], [21] with time complexity no greater than $O(n \log n)$ to the dual graph. With the minimum spanning tree obtained we can perform a depth-first search starting with the specified point to visit each point once. By the triangle inequality the total path LENGTH must be smaller than that of traversing the minimum spanning tree edges twice to visit all the points. That is, if MST denotes the total length of the minimum spanning tree, we have $\text{LENGTH} < 2 \cdot \text{MST}$ [18]. Since the optimal path is a spanning tree, the total length OPT must be greater than MST. Thus, we have $\text{LENGTH} < 2 \cdot \text{OPT}$. In the worst case, the approximate solution LENGTH may tend to twice the optimal solution. To see this, suppose all the n points are colinear and there are x points on one side of the specified point and $n - x - 1$ points on the other side. Let the distances of the two extreme points to the specified point be d_1 and d_2 respectively. Suppose $d_1 \gg d_2$. The optimal solution would be $2d_2 + d_1$, while the approximate solution could be $2d_1 + d_2$. Therefore $2d_1 + d_2 \approx 2 \cdot \text{OPT}$.

3.3.2. Closest insertion method. This approach is essentially the same as that given in [16] except that we work on a *path* rather than a *circuit*. We start with a path consisting of a single node, i.e. the specified starting point. At each step, we find the uncontained node k closest to any contained node, i.e. find a minimum $d(m, j)$ such that m is in the path and j is not, and take $k = j$. Then we insert the node k to one of the intervals (p, m) and (m, q) where p, q are in the path and (p, m), (m, q) are edges of the path. Suppose the interval (p, m) is chosen. We replace edge (p, m) by (p, k) and (k, m). To implement this procedure, we shall use an AVL tree [13] as our data structure. Whenever a node k is selected, those edges incident with k in the dual graph of the Voronoi diagram will be inserted to the tree. Those edges that are already in the tree are excluded. To find the node k, we search through the tree to find the minimum edge (m, j) and delete it from the tree. If both m and j are already in the path, we keep searching and deleting until an edge (m, j) with an end point, say j, not in the path is found. Take $k = j$. Since the total number of edges in the dual graph is $O(n)$, we at most perform $O(n)$ insertions to and deletions from the AVL tree, and the time required is $O(n \log n)$. This approximation algorithm also yields a solution within a factor of 2 [16] and the bound is tight in the worst case, as described earlier.

4. Experimental results. In this section, we discuss the expected performance of the approximation algorithms. As mentioned in § 1, we assume that all of the N^2 grid points are equally likely to be accessed. Suppose at time t, a batch A_t of B requests is generated. (Note that multiple requests to a grid point are allowed.) Suppose the last

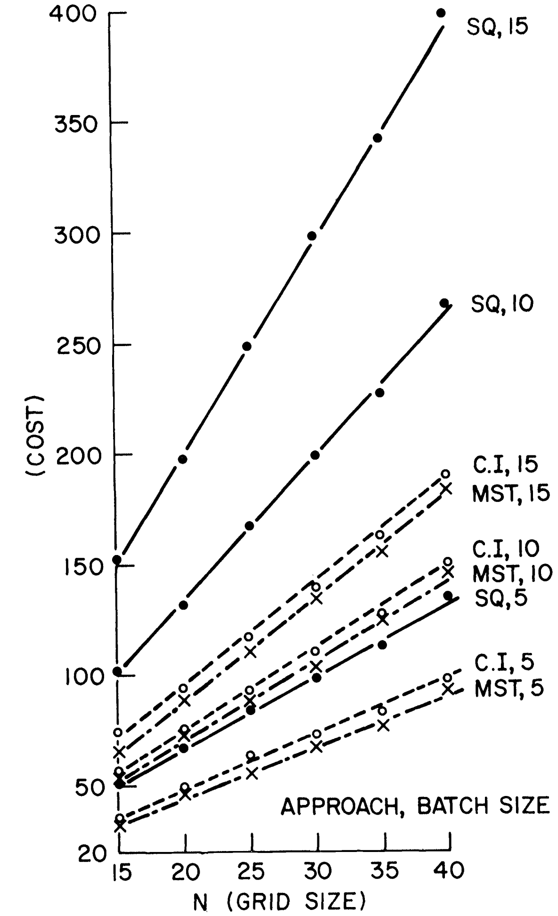


FIG. 6. *Graph of numerical results.*

position of the head is $\mathscr{S}$. (For $t = 0$, $\mathscr{S}$ is randomly chosen.) Let U_t denote the total length of the Hamiltonian path with starting point $\mathscr{S}$ determined by the minimum spanning tree method and W_t denote that by the closest insertion method. Then $\mathsf{U} = \lim_{t\to\infty} (\sum_t \mathsf{U}_t)/t$ is a measure of the expected performance of the minimum spanning tree method. A similarly defined W has the same function for the closest insertion method. Since U and W are very difficult to compute analytically, we resort to simulation. We use a pseudo-random number generator to simulate the random requests. Table 1 shows the outcomes for different values of N and B. The batched processing is far better than the sequential one as indicated in Fig. 6. It also shows that the minimum spanning tree approach is better than the closest insertion method.

TABLE 1

B \ N	15	20	25	30	35	40	
5	50.33	68.01	83.83	98.83	113.41	135.12	Sequential
	33.26	46.06	56.10	65.66	77.21	92.91	Minimum Spanning Tree
	36.77	50.04	63.19	72.95	83.66	98.77	Closest Insertion
10	101.52	131.66	167.22	199.25	227.62	268.97	SQ
	51.08	72.62	88.19	103.91	124.49	145.83	MST
	56.50	75.78	92.88	111.63	128.75	150.96	CI
15	152.30	197.53	249.72	298.95	342.37	399.11	SQ
	66.60	90.62	111.56	136.52	153.98	185.41	MST
	73.38	94.87	118.41	139.93	163.74	190.06	CI

5. Conclusions. This paper can be regarded as a contribution to the growing studies of algorithms in L_1 and L_∞ metrics. These metrics have many important practical applications other than the ones mentioned here. For example, applications in wire layout for printed circuit boards have been thoroughly discussed in [1], [5], [6].

One useful observation made in the present paper is the computational equivalence of L_1 and L_∞ metrics. Based on this, only one of these two metrics needs to be studied. Depending on specific applications, one of them may prove to be more convenient than the other to study.

We have discussed two $O(n \log n)$ methods to construct open Hamiltonian paths with a fixed starting point. Another possible and more intuitive approximation algorithm, for example, is the nearest neighbor method [16] which runs in $O(n^2)$ time for a general metric. Whether or not the Voronoi diagram would help cut down the time complexity is not known. For smaller batch size, this approach might be better than those studied above as far as the time complexity is concerned.

Finally, as to the expected performance, mentioned in § 4, we conjecture that the methods proposed by Karp [9], [10], [11] may well be applicable to this problem.

Acknowledgment. The authors wish to thank Professors F. P. Preparata and Jon Bentley for helpful discussions.

REFERENCES

[1] A. V. AHO, M. R. GAREY AND F. K. HWANG, *Rectilinear Steiner trees: Efficient special case algorithms*, Networks, 7 (1976), pp. 37–58.

[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. K. CHANDRA, H. CHANG AND C. K. WONG, *2-dimensional fast access to contiguous data in 2-dimensional magnetic bubble memories*, U.S. Patent Pending.

[4] S. A. COOK, *The Complexity of theorem proving procedure*, Proc. 3rd Annual ACM Symposium on Theory of Computing, May 1971, pp. 151–158.

[5] M. R. GAREY, R. L. GRAHAM AND D. S. JOHNSON, *Some NP-complete geometric problems*, Proc. 8th Annual ACM Symposium on Theory of Computing, May 1976, pp. 10–22.

[6] M. R. GAREY AND D. S. JOHNSON, *The rectilinear Steiner tree problem is NP-complete*, SIAM J. Appl. Math., 32 (1977), pp. 826–834.

[7] D. S. JOHNSON, private communication.

[8] 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.

[9] ———, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–48.

[10] ———, *The fast approximation solution of hard combinatorial problems*, Proc. 6th Southeastern Conference on Combinatories, Graph Theory and Computing, Utilitas Mathematica, Winnipeg, 1975, pp. 15–31.

[11] ———, *The probabilistic analysis of some combinatorial search algorithms*, Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York 1976, pp. 1–19.

[12] 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.

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

[14] D. T. LEE, *On finding K-nearest neighbors in the plane*, Coordinated Science Laboratory Report R-728, University of Illinois, Urbana, IL, May 1976.

[15] C. A. ROGERS, *Packing and Covering*, Cambridge University Press, London, 1964.

[16] D. J. ROSENKRANTZ, R. E. STEARNS AND P. M. LEWIS II, *An analysis of several heuristics for the traveling salesman problem*, this Journal, 6 (1977), pp. 563–581.

[17] M. I. SHAMOS, *Problems in Computational Geometry*, Springer–Verlag, New York, to be published.

[18] M. I. SHAMOS AND D. HOEY, *Closest-point problems*, Proc. 16th Annual Symposium on Foundations of Computer Science, Oct. 1975, pp. 151–162.

[19] B. SHNEIDERMAN AND V. GOODMAN, *Batched searching of sequential and tree structured files*, ACM Trans. Database Sys., 1 (1976), pp. 268–275.

[20] C. K. WONG AND K. C. CHU, *Average distances in L_p disks*, SIAM Rev., 19 (1977), pp. 320–324.

[21] A. YAO, *An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees*, Information Processing Lett., Sept. 1975, pp. 21–23.

[22] P. C. YUE AND C. K. WONG, *Near-optimal heuristics for the 2-dimensional storage assignment problem*, International J. comput. Information Sci., 4 (1975), pp. 281–294.

[23] J. R. BITNER AND C. K. WONG, *Optimal and near-optimal scheduling algorithms for batched processing in linear storage*, this Journal, 8 (1979), pp. 479–499.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

ON THE COMPLEXITY OF CANONICAL LABELING OF STRONGLY REGULAR GRAPHS*

LÁSZLÓ BABAI†

Abstract. We prove that a canonical labeling can be assigned to the n vertices of a strongly regular graph by an algorithm of $o(\exp(2n^{1/2}\log^2 n))$ running time (in the worst case). This complexity, though still not properly subexponential, is much better than $O(2^n)$.

Key words. isomorphism testing, canonical labeling, strongly regular graph, graph, worst-case analysis, performance bounds.

1. Introduction. A graph Γ is *strongly regular* if each vertex of Γ has the same valence and the number of common neighbours of any two vertices v, w depends only on whether v and w are adjacent or not. Strongly regular graphs are recognized as hard cases for isomorphism testing algorithms (cf. [3], [4], [5], [6]). Though a great number of isomorphism testing algorithms has been published, no theoretical estimate of their worst case running time seems to be available, which would be essentially better than $\exp(cn\log n)$. The aim of the present note is to show that well-known simple ideas result in an algorithm whose *worst case* running time is $o(\exp(2\sqrt{n}\log^2 n))$. In complexity theory it is reasonable to call the growth of $\exp(n^\alpha)$ exponential for any $\alpha > 0$ (from the point of view of polynomial equivalence, $\exp(n^\alpha)$ and $\exp(n^\beta)$ are the same). In this sense, our estimate is still exponential; nevertheless, it is much better than $O(2^n)$, say.

We stress that our aim is to deal with the *worst case* and not with a good average performance. Therefore, the algorithm cannot be used for practice: it will need much time even in cases when more sophisticated algorithms work promptly. One might ask what would be the purpose of an algorithm like this. Let me answer this question. Such a theoretical estimate may serve as a *test of algorithms*: if a sophisticated algorithm is not faster (in worst case) than our straightforward algorithm, then we may well guess that the complicated procedures cover lack of essential ideas.

Another possible use of this note is that Theorem 3.1 on which our estimate is based might be applicable to other similar algorithms, too.

We remark that there exists an algorithm testing graph isomorphism within linear *expected* time [1]. (Actually, a canonical labeling is constructed in linear expected time.)

2. Preliminaries. The algorithm. For Γ a graph, $V\Gamma$ denotes its vertex set. Let $\mathbf{K}$ be a class of graphs having the same vertex set V. We assume that $\mathbf{K}$ is closed under isomorphisms, i.e. if Γ_1 and Γ_2 are isomorphic graphs with $V\Gamma_1 = V\Gamma_2 = V$, then $\Gamma_1 \in \mathbf{K}$ implies $\Gamma_2 \in \mathbf{K}$. Let $|V| = n$.

DEFINITION 2.1. By a *canonical labeling* of the class $\mathbf{K}$ we mean a function L whose domain is $\mathbf{K}$ such that

(i) $L(\Gamma)$ is a labeling of Γ (i.e. a bijection $V \to \{1, \cdots, n\}$) for any $\Gamma \in \mathbf{K}$;

(ii) If Γ_1 and Γ_2 belong to $\mathbf{K}$, then they are isomorphic (if and) only if the map $L(\Gamma_2)^{-1} \circ L(\Gamma_1)$: $V \to V$ is an isomorphism $\Gamma_1 \to \Gamma_2$.

Clearly, a canonical labeling can be used to decide whether the graphs Γ_1 and Γ_2 are isomorphic, provided at least one of them belongs to $\mathbf{K}$. Our aim is to construct a canonical labeling, defined on the class $\mathbf{K}$ of strongly regular graphs. Given $\Gamma \in \mathbf{K}$, the labeling $L(\Gamma)$ will be computed within $o(\exp(2\sqrt{n}\log^2 n))$ time.

* Received by the editors May 14, 1978, and in revised form August 16, 1978.

† Department of Algebra, Eötvös L. University, Budapest, Hungary.

2.2. The graph pK_q, consisting of p disjoint complete graphs is strongly regular. In all other strongly regular graphs, the relation "xRy iff $x = y$ or x and y are adjacent" is *not* an equivalence relation. Similarly, for all strongly regular graphs but the complement of the graphs pK_q, the relation "$xR'y$ iff $x = y$ or x and y are nonadjacent" is not an equivalence relation.

One can recognize and assign canonical orderings to the graphs pK_q and their complements by a straightforward linear time $(=O(n^2))$ algorithm. Henceforth we exclude these graphs when using the term "strongly regular graph."

2.3. For Γ a graph, $\Gamma(v)$ denotes the set of neighbours of $v \in V\Gamma$. ($v \notin \Gamma(v)$.)

DEFINITION. A set $S \subset V\Gamma$ *distinguishes* the vertices v, $w \in V\Gamma$ if $v \neq w$ and either $v \in S$ or $w \in S$ or $\Gamma(v) \cap S \neq \Gamma(w) \cap S$. The set $\Gamma(v) \cap S$ is the *trace* of v in S. We call S a *distinguishing set* if S distinguishes each pair of distinct vertices.

Note that if $S_1 \subset S_2 \subset V\Gamma$ and S_1 is a distinguishing set then so is S_2.

ASSUMPTION 2.4. *The graph Γ has a distinguishing set of a given cardinality s.*

We describe an algorithm to canonically label the class of graphs Γ satisfying this assumption, where $V\Gamma = V$, $|V| = n$.

2.5. The algorithm. We generate the ordered s-tuples of distinct vertices $X_j = (x_1^{(j)}, \cdots, x_s^{(j)})$ $(j = 1, 2, \cdots, \binom{n}{s}s!)$ by some backtrack procedure. Let $S_j = \{x_1^{(j)}, \cdots, x_s^{(j)}\}$. Given X_j, we order the remaining $n - s$ vertices in the lexicographic order of their traces on X_j: we set $v <_j w$ if for the smallest i such that $x_i^{(j)} \in \Gamma(v) \Delta \Gamma(w)$, this $x_i^{(j)}$ belongs to $\Gamma(v)$. (Δ stands for symmetric difference.) If this ordering is not linear (i.e., S_j is not distinguishing), we reject X_j and continue with X_{j+1}. If S_j is a distinguishing set, we obtain a labeling $(x_1^{(j)}, \cdots, x_n^{(j)})$ of the vertices of Γ, and this yields an adjacency matrix M_j. We select the lexicographically first one among all these adjacency matrices (this requires an elementwise comparison of two matrices at each j such that S_j is a distinguishing set). We declare the corresponding labelling *canonical.* (It clearly satisfies our definition of canonicity.)

The space required by the algorithm is $O(n^2)$. We estimate the running time. Given X_j, the labeling of $V\Gamma$ is found (or X_j is rejected) within $O(sn^2)$ time. The comparison of the adjacency matrix M_j with $M_{j'}$ (the lexicographically first among M_k, $k < j$) takes n^2 time. Finding X_{j+1} requires $O(n)$ time.

Summarizing, running time is bounded by

$$Csn^2\binom{n}{s}s! = o(n^{s+3}). \tag{1}$$

2.6. Application to strongly regular graphs. By Theorem 3.1 (below), Assumption 2.4 holds for strongly regular graphs of order $n > n_0$ with $s = \lfloor 2\sqrt{n} \log n \rfloor - 3$ (cf. 2.2). Hence the running time of our algorithm is bounded by

$$o(n^{2\sqrt{n}\log n}) = o(\exp(2\sqrt{n}\log^2 n).$$

3. Distinguishing sets. In this section we prove the existence of distinguishing sets of cardinality $2\sqrt{n}\log n$ in strongly regular graphs of order n. We use a probabilistic argument.

THEOREM 3.1. *Let Γ be a strongly regular graph which is neither the union of disjoint complete graphs nor the complement of such a graph. Let $|V\Gamma| = n > n_0$.*
Then Γ has a distinguishing set of cardinality $\lfloor 2\sqrt{n}\log n \rfloor - 3(\lfloor x \rfloor$ denotes the greatest integer not exceeding x.)

LEMMA. 3.2. *Let n, k be positive integers. Given a graph Γ of order n, assume that*

$$|\Gamma(v) \Delta \Gamma(w)| \geqq k$$

for any pair (v, w) *of distinct vertices. Then,* Γ *has a distinguishing set* $S \subset V$ *of cardinality* $|S| \leqq \lceil 2n \log n/(k+2) \rceil$ *provided* $k > 4 \log n$. ($\lceil x \rceil$ denotes the least integer not smaller than x.)

Proof. Let us fix an integer s and choose a subset $S \subset V$ of cardinality s at random, each subset having probability $1/\binom{n}{s}$ to be chosen. For $v, w \in V$, $v \neq w$, let $A(v, w)$ denote the event that S does not distinguish v and w. Let $P(v, w)$ denote the probability of this event. Clearly, $A(v, w)$ is equivalent to the event that

$$S \cap (\{v, w\} \cup (\Gamma(v) \Delta \Gamma(w))) = \emptyset,$$

hence

$$P(v, w) \leqq \binom{n-k}{s} \Big/ \binom{n}{s}.$$

Let N denote the number of pairs (v, w) such that $A(v, w)$ holds. The expected value of N is

$$E(N) = \sum_{\{v,w\}} P(v, w) \leqq \binom{n}{2}\binom{n-k}{s} \Big/ \binom{n}{s}.$$

Assume now that

$$\binom{n}{2}\binom{n-k}{s} < \binom{n}{s}. \tag{1}$$

Then $E(N) < 1$, hence Prob $(N = 0) > 0$. But $N = 0$ holds iff S is a distinguishing set. This proves that (1) implies the existence of a distinguishing set of size s.

Let now $s \geqq 2n \log n/(k+2)$. All we have to prove is that (1) holds in this case. Using the condition $k > 4 \log n$ we infer

$$2 \log n - \log 2 < sk/n,$$

hence

$$\binom{n}{2} < (\exp(k/n))^s < \left(1 + \frac{k}{n} + \frac{k^2}{n^2}\right)^s < \prod_{i=0}^{s-1}\left(1 + \frac{k}{n-k-i}\right) = \binom{n}{s} \Big/ \binom{n-k}{s}. \qquad \square$$

Remark. This way we have actually proved that almost all s-subsets of V are distinguishing. There is another way to prove Lemma 3.2, using Lovász's estimate on the efficiency of the greedy cover algorithm [2]. As a matter of fact, let us consider the set-system (hypergraph) $\{\Gamma(v) \Delta \Gamma(w) : v, w \in V, v \neq w\}$. A fractional cover of this hypergraph is obtained by assigning the weight $1/k$ to every vertex, hence the optimum fractional cover is $\tau^* \leqq n/k$. Now, by Lovász's estimate, the size s of any cover obtained by the greedy cover algorithm (see [2]) does not exceed $\tau^*(1 + \log D)$, D being the maximum degree in this hypergraph. $D \leqq \binom{n}{2}$ in our case, hence $s \leqq n(1 + 2 \log n)/k$, and this is essentially the same as 3.2.

3.3 In order to formulate the second lemma, we need some definitions. A *hypergraph* is a pair $H = (V, \mathbf{F})$ where V is a nonempty set (the set of vertices) and $\mathbf{F}$ is a family of subsets of V (the edges of H). If $\mathbf{F} = \emptyset$ we say H is empty. H is *r-uniform* if $|E| = r$ for any $E \in \mathbf{F}$. The *degree* of a vertex is the number of edges containing it. H is *regular* if every vertex of H has the same degree.

LEMMA. 3.4. *Let* $1 \leqq d < r < n$ *be integers,* $|V| = n$ *and* $H = (V, \mathbf{F})$ *a nonempty regular r-uniform hypergraph such that* $|E \cap F| \geqq d$ *for any* $E, F \in \mathbf{F}$. *Then* $r^2 > nd$.

Proof. Let t denote the degree of the vertices in H. So, $tn = |\mathbf{F}|r$. Let us select an edge E. Counting those pairs (v, F) satisfying $v \in E, F \in \mathbf{F}, F \neq E$ and $v \in F$ in two ways, we obtain

$$r(t-1) \geqq (|\mathbf{F}|-1)d = \left(\frac{tn}{r}-1\right)d.$$

Rearranging this inequality, we infer

$$t(r^2-nd) \geqq r(r-d).$$

The right side being positive, $r^2 > nd$ follows. □

LEMMA 3.5. *Let Γ be a strongly regular graph of order n. Then*

$$|\Gamma(v)\,\Delta\Gamma(w)| > \sqrt{n}-1$$

for any v, $w \in V\Gamma$. (We assume that neither Γ nor its complement is the disjoint union of complete graphs.)

Proof. Let $r = |\Gamma(v)|$. As the complement of a strongly regular graph is strongly regular, we may assume $r < n/2$. On the other hand, $r \geqq \sqrt{n-1}$ since the diameter of Γ is 2.

Let $|\Gamma(v) \cap \Gamma(w)| = d_1$ or d_2 according to whether v and w are adjacent or not. For $v, w, z \in V\Gamma$ we have

$$\Gamma(v)-\Gamma(w) \subseteq (\Gamma(v)-\Gamma(z)) \cup (\Gamma(z)-\Gamma(w)).$$

If v and w are nonadjacent and z is adjacent to both, we have

$$|\Gamma(v)-\Gamma(w)| = r-d_2;$$

and

$$|\Gamma(v)-\Gamma(z)| = |\Gamma(z)-\Gamma(w)| = r-d_1.$$

Hence

$$r-d_2 \leqq 2(r-d_1). \tag{2}$$

We obtain analogously

$$r-d_1 \leqq 2(r-d_2). \tag{3}$$

Note that in deriving these inequalities we used that Γ is neither the union of disjoint complete graphs nor the complement of such a graph (cf. 2.2).

Let $d = \min(d_1, d_2)$. We have (using (2) and (3)) for any $v, w \in V\Gamma (v \neq w)$

$$|\Gamma(v)\,\Delta\Gamma(w)| = 2(|\Gamma(v)|-|\Gamma(v)\cap\Gamma(w)|) = 2(r-d_i) \geqq r-d.$$

If $d = 0$, we are done since $r \geqq \sqrt{n-1} > \sqrt{n}-1$. If $d \geqq 1$, we may apply Lemma 3.4 to the nonempty r-uniform hypergraph $\{V\Gamma, \Gamma(v)\colon v \in V\Gamma\}$. We infer $r > \sqrt{nd}$, hence

$$r-d > \sqrt{nd}-d = \sqrt{d}(\sqrt{n}-\sqrt{d}) \geqq \sqrt{n}-1.$$

(We used that $1 \leqq d < n/2$, the latter inequality being a consequence of our assumption $r < n/2$.) □

3.6. Proof of the theorem. By the lemma of 3.5, Γ satisfies the assumptions of Lemma 3.2 with $k = \sqrt{n}-1$. We infer by 3.2 that Γ has a distinguishing set S of cardinality $|S| \leqq 1+2n\log n/(\sqrt{n}+1) < 2\sqrt{n}\log n-4$ (if $n \geqq 25$). (The condition $k > 4\log n$ also holds if $n > 800$.) □

Note added in proof. Gary L. Miller proves [7] that isomorphism testing can be performed in exp $(\log^2 n)$ time for certain classes of strongly regular graphs, related to Latin squares.

REFERENCES

[1] L. BABAI AND L. KUČERA, *Graph canonization with linear expected time*, in preparation.

[2] L. LOVÁSZ, *On the ratio of optimal integral and fractional cover*, Discrete Math., 13 (1975), pp. 383–390.

[3] R. MATHON, *Sample graphs for graph isomorphism testing*, Proc. 9th Southeastern Conf. on Comb., Graph Theory and Computing (1978), to appear.

[4] A. J. L. PAULUS, *Conference matrices and graphs of order* 26, Tech. rept., Dept. Math., Technological University, Eindhoven, 1973.

[5] R. C. READ AND D. G. CORNEIL, *The graph isomorphism disease*, J. Graph Theory, 1 (1977), pp. 339–363.

[6] B. WEISFEILER, *On construction and identification of graphs*, Springer Lecture N No. 558, Springer-Verlag, Heidelberg, 1976.

[7] G. L. MILLER, *On the* $n^{\log n}$ *isomorphism technique*, to appear.

SIAM J. COMPUT.
Vol. 9, No. 1, February 1980

0097-5397/80/0901-0018 $01.00/0

CORRIGENDUM. A NEW REPRESENTATION OF THE RATIONAL NUMBERS FOR FAST EASY ARITHMETIC*

E. C. R. HEHNER† AND R. N. S. HORSPOOL‡

p. 125, line −5, change "$0<d_i<b$" to "$0\leqq d_i<b$".
p. 126, line +6, change "22004" to "27004".
p. 127, line 29, change "$0'-3=6'7$" to "$0'-3'=6'7$".
p. 127, line −7, change "127" to "12'7".
p. 127, line −3, change "9'6" to "9'3".

We were informed by D. Knuth of Hensel's and Krishnamurthy's work just in time to introduce a reference into the journal paper. More reference is due, and given in our later paper "Exact Arithmetic using a Variable-Length *P*-adic Representation", Proceedings of 4th Symposium on Computer Arithmetic, IEEE, Santa Monica, October 1978.

* This Journal, 8 (1979), pp. 124–134. Received by the editors July 12, 1979.

† Computer Systems Research Group, University of Toronto, Toronto, Ontario, Canada M5S 1A1.

‡ School of Computer Science, McGill University, Montreal, Quebec, Canada H3A 2K6.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

A MULTI-TERMINAL MINIMUM CUT ALGORITHM FOR PLANAR GRAPHS*

YOSSI SHILOACH†

Abstract. Given an undirected planar graph with n vertices, we present an $O((n \log n)^2)$ algorithm which finds the minimum cuts between all the pairs of vertices.

Key words. algorithm, minimum cut, multi-terminal, planar graphs

1. Introduction. Through this paper, $G=(V, E)$ is an undirected, connected planar graph and $|V|=n$.

Given $s, t \in V$, an (s, t)-cut, C_{st}, is a set of edges, the deletion of which disconnects s and t. $|C_{st}|$ is the *size* of the cut. An (s, t)-cut is a *minimum* (s, t)*-cut* if its size is not greater than that of any other (s, t)-cut.

For any $s, t \in V$, a minimum (s, t)-cut can be (and usually is) found by regarding G as a flow network in which all the capacities are 1 and solving a max flow problem from s to t (or vice-versa). Finding a minimum cut in a general graph can be done in $O(n^{2/3}(n+m))$, (where $m=|E|$), as shown in [1]. This amounts to $O(n^{5/3})$ as far as planar graphs are concerned. Thus, if we employ Gomory and Hu's multi-terminal minimum-cut algorithm given in [2], we obtain an $O(n^{8/3})$ algorithm for computing all the minimum cuts in an undirected planar graph. However, recent results which are given in [3] enable us to reduce the time to $O((n \log n)^2)$. Note that this new algorithm has an average of only $O((\log n)^2)$ for finding one minimum cut.

Given a planar network $N=(G; c; s, t)$ (directed or not) and a positive number k, it is shown in [3] that a flow of value k from s to t can be found if it exists, in time of $O(d_{st} \cdot n \log n)$ where d_{st} is the distance from s to t, i.e., the length of a shortest (s, t) path. The same algorithm also indicates if such a flow does not exist. (The algorithm is designed for directed networks but can be easily applied to undirected networks by regarding each undirected edge as a pair of directed edges in opposite directions.) Following the lines of this algorithm, it is easy to verify that it can be implemented in $O(d_{st} \cdot n)$ time if all the capacities are 1. In this case, the value of a minimum cut is an integer, not greater than $n-1$. Thus we can apply the same algorithm $\lfloor \log n \rfloor$ times by performing a binary search on the set $\{0, 1, \cdots, n-1\}$ to yield the right value of a minimum cut.

This result is summarized in the following lemma.

LEMMA 1.1. *Given a planar* (*directed or not*) *graph and* $s, t \in V$, *we can find a minimum* (s, t)*-cut within* $O(d_{st} \cdot n \log n)$ *time.*

2. How to implement the G–H algorithm efficiently. In their famous multi-terminal minimum cut algorithm, (denoted here as the *G–H algorithm*), R. E. Gomory and T. C. Hu show that a weighted tree T can be constructed by computing just $n-1$ max-flow problems, such that the minimum cut between each pair of vertices can be obtained in $O(n)$ time, by using the information stored in T. Thus, constructing T is the dominant part in the G–H algorithm.

* Received by the editors May 2, 1978.

† Computer Science Department, Stanford University, Stanford, California. Now at IBM Scientific Center, Technion City, Haifa, Israel. This research was supported by a Chaim Weizmann Postdoctoral Fellowship and by National Science Foundation under Grant MCS75-22870.

In this paper we show that the sources, $s_1, \cdots, s_{n-1}$, and the terminals, $t_1, \cdots, t_{n-1}$, of these $n-1$ max-flow problems can be chosen close enough to yield:

$$\sum_{i=1}^{n-1} d_{s_i t_i} = O(n \log n). \tag{2.1}$$

Equation (2.1) combined with Lemma 1.1 yield the time of $O((n \log n)^2)$ for the whole algorithm.

3. The G–H algorithm preserves planarity. During the G–H algorithm, sets of vertices are contracted[1] into one node and it is not very clear that planarity is preserved. This will be proved in this section and will enable us to use the result of Lemma 1.1 for all the graphs which we encounter during the algorithm. We shall use the word "nodes" to denote super-vertices which are formed by contracting several vertices of the original graph. "Vertices" will always denote vertices of the original graph. We will sometimes identify a node with the set of vertices that it represents, and use the same character for both.

Though we assume that the reader is familiar with the G–H algorithm, we are going to describe its kth step in detail. The reason for that is that the same notations and terminology will be used several times later.

The input to the kth step is a weighted tree $T^k = (U^k, A^k, w^k)$, where $U^k = \{U_1^k, \cdots, U_k^k\}$. The U_i^k's are mutually disjoint subsets of V and their union is V. (Note that T^k has exactly k nodes.) $w^k: A^k \to Z^+$ assigns a positive integer to each edge of T^k.

Step k $(k \leqq n-1)$.

1. Choose a node U_i^k such that $|U_i^k| \geqq 2$. Without loss of generality we assume that U_k^k is chosen.
2. Let $T_1^k, \cdots, T_r^k$ denote the connected components which are obtained from T^k if U_k^k is removed.
3. Let $S_j = \{v \in V | v \in U_i^k$ and $U_i^k \in T_j^k\}$, $j = 1, \cdots, r$.
4. Let $G^k = (V^k, E^k)$ be the graph which is obtained from G by contracting each of the S_j's into a single node. (G^k may have multiple edges.)
5. Choose two vertices $s, t \in U_k^k$. (Note that $s, t \in V^k$ too.) Solve one max-flow problem from s to t in G^k, and find a set $C \subset V^k$ such that $s \in C$, $t \in \bar{C} = V^k - C$ and the (s, t)-cut determined by C is a minimum (s, t)-cut having size of c.
6. Let $T^{k+1} = (U^{k+1}, A^{k+1}, w^{k+1})$ be given by: $U^{k+1} = \{U_1^{k+1}, \cdots, U_{k+1}^{k+1}\}$ where $U_i^{k+1} = U_i^k$ for $i = 1, \cdots, k-1$, $U_k^{k+1} = U_k^k \cap C$ and $U_{k+1}^{k+1} = U_k^k \cap \bar{C}$. A^{k+1} and w^{k+1}: If $e = (U_i^k, U_j^k)$ and $i, j \neq k$, then e becomes (U_i^{k+1}, U_j^{k+1}) and retains its weight. If $e = (U_i^k, U_k^k)$ and $U_i^k \subseteq S_l \in C$ then e transforms into (U_i^{k+1}, U_k^{k+1}). If $S_l \in \bar{C}$, then e becomes $(U_i^{k+1}, U_{k+1}^{k+1})$. In both cases it retains its weight. A new edge $(U_k^{k+1}, U_{k+1}^{k+1})$ is added with a weight of c.

The initial tree is $T^1 = (U^1, A^1, w^1)$ where $U^1 = \{U_1^1\} = \{V\}$ and $A^1 = w^1 = \emptyset$. Thus $G^1 = G$. $T^n = T$ is the output tree of the G–H algorithm.

THEOREM 3.1. *G^k is planar for $k = 1, \cdots, n-1$.*

Proof. $G^1 = G$ and therefore planar. In the following we will show that if G^i is planar for $i \leqq k$, so is G^{k+1}.

The next Lemma is a well-known one.

LEMMA 3.1.1. *If $G = (V, E)$ is planar and $(u, v) \in E$, then the contraction of u and v into a single node yields a planar graph.*

[1] Contracting several vertices into one node usually yields multiple edges. Thus, we allow multiple edges to occur. Lemma 1.1 still holds as long as the number of edges remains $O(n)$. Since these edges are original edges of the graph, their number cannot exceed $O(n)$.

LEMMA 3.1.2. *If $G=(V, E)$ is planar and $G'=(V', E')$ is a connected subgraph of G, then the contraction of V' into a single node yields a planar graph.*

This lemma can be easily proved by using the previous one as a basis for induction on $|V'|$ and for the inductive step as well.

DEFINITION. In Step k above, we transformed T^k into T^{k+1} by splitting U_k^k into U_k^{k+1} and U_{k+1}^{k+1}. This operation will be denoted as *splitting* a given node.

LEMMA 3.1.3. *If U_k^k is split in Step k into U_k^{k+1} and U_{k+1}^{k+1} and if U_{k+1}^{k+1} (U_k^{k+1}) is split in the next step, then G^{k+1} can be obtained from G^k by contracting the vertices and nodes of C $(\bar{C})$ into a single node.*

Proof. The connected components of T^{k+1} which are obtained by removing U_{k+1}^{k+1} and its incident edges, remain the same, if they belong to $\bar{C}$. Those which belong to C, together with U_k^{k+1}, form a new connected component (of T^{k+1}) that consists exactly of the vertices of C (see Fig. 1). Thus G^{k+1} can be obtained from G^k by contracting the nodes and vertices of C into a single node. The corresponding assertion in the parentheses is proved in the same way. □

Let $G^k(C)$ denote the‘subgraph of G^k which is induced by C.

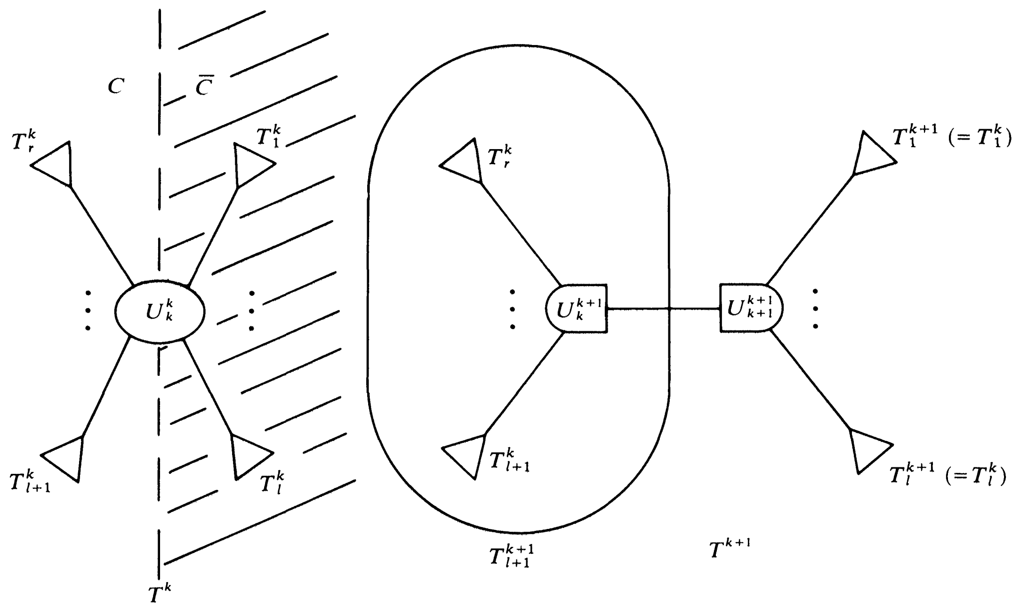


FIG. 1

LEMMA 3.1.4. *$G^k(C)$ $(G^k(\bar{C}))$ is a connected graph.*

Proof. We assumed in the beginning of this paper that G is connected. Contractions preserve connectedness and therefore G^k is also connected. Assume to the contrary $G^k(C)$ has more than one connected component. Let $B=(V_B, E_B)$ be a connected component of $G^k(C)$, such that $s \notin V_B$. Since G^k is connected, B must be connected to $\bar{C}$ and since $s \notin V_B$, $C-V_B$ also determines an (s, t)-cut. This cut, however, is smaller than C and contradicts its minimality. The proof for $\bar{C}$ is similar. □

Lemmas 3.1.2, 3.1.3, and 3.1.4 imply the following corollary.

COROLLARY 3.1.5. *If U_k^k is split in Step k into U_k^{k+1} and U_{k+1}^{k+1} and if either U_k^{k+1} or U_{k+1}^{k+1} is split in the next step, then G^{k+1} is planar.*

LEMMA 3.1.6. *If U_k^k is split in Step k into U_k^{k+1} and U_{k+1}^{k+1} and if U_k^{k+1} (U_{k+1}^{k+1}) is split in Step l, where $l > k$, then G^l is planar.*

Proof. The connected components, resulting from T^{k+1} when U_k^{k+1} (U_{k+1}^{k+1}) is removed, are not affected by a further splitting of other nodes, different from U_k^{k+1} (U_{k+1}^{k+1}). Thus, they are also the connected components, resulting from T^l when U_k^{k+1} (U_{k+1}^{k+1}) is removed. Thus, G^l is the same graph as G^{k+1} of Lemma 3.1.3, (i.e., as G^{k+1} in case that U_k^{k+1} (U_{k+1}^{k+1}) is split in Step $k+1$). Thus, by Corollary 3.1.5, G^l is planar. □

The proof of Theorem 3.1 follows immediately from Lemma 3.1.6. □

4. How to achieve (2.1). Let $s_1, \cdots, s_{n-1}, t_1, \cdots, t_{n-1}$ denote the sources and terminals respectively, which are used in the $n-1$ max-flow problems, computed during the G–H algorithm. Thus we have:

$$s_i, t_i \in V^i, \qquad i = 1, \cdots, n-1. \tag{4.1}$$

G has at least as many vertices and edges as each of the G_i's. Thus the complexity of our algorithm would not exceed $O(n \log n \sum_{i=1}^{n-1} d_{s_i t_i})$.

In the following, we will show that the s_i's and t_i's can be chosen so that (2.1) is satisfied. Moreover, the time involved in choosing them will be negligible.

LEMMA 4.1. *Let G be a connected graph with n vertices, m of which are blue and the other $n-m$ are red. Then there are two blue vertices v_i, v_j such that*

$$d_{v_i v_j} \leqq 2\lfloor n/m \rfloor. \tag{4.2}$$

Proof. Let $v_1, \cdots, v_m$ be the blue vertices, and let $G_i = (V_i, E_i)$, $i = 1, \cdots, m$, be connected subgraphs of G such that $v_i \in V_i$ and $|V_i| = \lfloor n/m \rfloor + 1$ for $i = 1, \cdots, m$. Such G_i's can be easily found if we start a search at v_i and stop it as soon as $\lfloor n/m \rfloor + 1$ vertices are encountered.

$\sum_{i=1}^m |V_i| > m \cdot n/m = n$ and therefore there exist i and j such that $V_i \cap V_j \neq \emptyset$. Let $v \in V_i \cap V_j$; then $d_{v_i v} \leqq \lfloor n/m \rfloor$ and $d_{v_i v} \leqq \lfloor n/m \rfloor$. Thus $d_{v_i v_j} \leqq 2 \cdot \lfloor n/m \rfloor$. □

The bound on this lemma cannot be improved as one can verify by considering a star with a red center and blue leaves.

COROLLARY 4.1.1. *If G^i has m vertices and r nodes, then s_i and t_i can be chosen so that*

$$d_{s_i t_i} \leqq 2\left\lfloor \frac{m+r}{m} \right\rfloor. \tag{4.3}$$

Moreover, using the search which is described in the proof of Lemma 4.1, s_i and t_i can be found in linear time.

Let us consider T^k again. Each of its nodes which contains $\alpha \geqq 2$ vertices is going to be split and subsplit until it breaks into α nodes (which are single vertices) in the final tree. We will refer to this process as a *complete split* of a given node. The whole G–H algorithm is just a complete split of the single node of T^1.

With each node that occurs in any of the trees $T^1, \cdots, T^k$, we shall associate two numbers. The first, m, is the number of vertices that it contains and the second, r, is the number of connected components, resulting by removing it from the tree that it belongs to. We have already noticed (Lemma 3.1.6) that if the node appears in T^k for the first time and in T^l for the last time $(l \geqq k)$ then r remains the same for $T^k, \cdots, T^l$. Thus, r can really be associated with the node as long as it exists, regardless of what tree it belongs to. Note that if this node is going to be split in the lth step, then m and r will be the numbers of the vertices and nodes of G^l, respectively. A node with the associated numbers m and r will be called an (m, r)-node.

We would like to have an upper bound on $\sum_{i=1}^{n-1} d_{s_i t_i}$ which can be interpreted as the sum of the distances between sources and terminals in all the max-flow problems which occur during a complete split of the initial $(n, 0)$-node, namely U_1^1. This motivates the following: Let U be an (m, r)-node; then $f_U(m, r)$ denotes the worst-case sum $\sum d_{s_i t_i}$ taken over all the max-flow problems which are involved in a complete split of U, assuming that s_i and t_i are always chosen so that (4.3) is satisfied.

Let $f(m, r) = \max\{f_U(m, r) | U \text{ is an } (m, r) \text{ node}\}$. In order to obtain an estimate for $f(n, 0)$ we establish and solve a recursion formula for $f(m, r)$.

LEMMA 4.2.

$$\begin{aligned} &f(1, r) = 0, \\ &f(m, r) \leqq 2\left\lfloor \frac{m+r}{m} \right\rfloor + \max\{f(m', r'+1) \\ &\qquad + f(m-m', r-r'+1) | 1 \leqq m' \leqq m-1, 0 \leqq r' \leqq r\}. \end{aligned} \tag{4.4}$$

Proof. Obviously, $f(1, r) = 0$ since a $(1, r)$-node is not going to be split anymore. In view of Lemma 3.3 and Fig. 1, it is easy to see that an (m, r)-node is always split into an $(m', r'+1)$-node and an $(m-m', r-r'+1)$-node, for some $1 \leqq m' \leqq m-1$ and $0 \leqq r' \leqq r$. Thus, (4.4) is now implied by Corollary 4.1.1. □

Let ln denote the natural logarithm.

THEOREM 4.3.

$$f(m, r) \leqq 2(m+r) \ln m. \tag{4.5}$$

Proof. Using induction on m, (4.5) reduces to proving that

$$\begin{aligned} &\max\{(m'+r'+1) \ln m' + (m-m'+r-r'+1) \ln (m-m') | 1 \leqq m' \leqq m-1, 0 \leqq r' \leqq r\} \\ &\qquad \leqq (m+r) \ln m - \left\lfloor \frac{m+r}{m} \right\rfloor. \end{aligned} \tag{4.6}$$

Since $(m'+r'+1) \ln m' + (m-m'+r-r'+1) \ln (m-m')$ is linear in r', it attains a maximal value when $r' = 0$ or $r' = r$. In the first case, the left-hand side of (4.6) reduces to

$$\max\{(m'+1) \ln m' + (m+r-m'+1) \ln (m-m') | 1 \leqq m' \leqq m-1\} \tag{4.7}$$

and in the second, it reduces to

$$\begin{aligned} &\max\{(m'+r+1) \ln m' + (m-m'+1) \ln (m-m') | 1 \leqq m' \leqq m-1\} \\ &\quad = \max\{((m-m')+1) \ln (m-m') \\ &\qquad + (m+r-(m-m')+1) \ln (m-(m-m')) | 1 \leqq m' \leqq m-1\}. \end{aligned} \tag{4.8}$$

Obviously, (4.7) and (4.8) are equal and we will consider only (4.7).

Let $g_1(m') = (m'+1) \ln m' + (m-m'+1) \ln (m-m')$ and let $g_2(m') = r \ln (m-m')$. Then

$$g_1'' = \frac{d^2 g_1}{dm'^2} = \frac{1}{m'} - \frac{1}{m'^2} + \frac{1}{m-m'} - \frac{1}{(m-m')^2}.$$

Since $g_1'' > 0$ in the interval $1 < m' < m-1$, g_1 is convex in this interval. Since it is continuous at 1 and $m-1$, a maximal value is attained when $m' = 1$ or $m' = m-1$, which in this case have the same value.

$g_2' = -r/(m-m') < 0$ for $1 \leqq m' \leqq m-1$ and therefore g_2 attains its maximal value when $m' = 1$.

Thus, (4.7) $= \max\{g_1(m')+g_2(m')|1 \leqq m' \leqq m-1\} = (m+r)\ln(m-1)$. Hence, the left-hand side of (4.6) $\leqq (m+r)/m+(m+r)\ln(m-1)<(m+r)\ln m$.

The last inequality follows from the inequality

$$\ln(m-1)+\frac{1}{m}<\ln m$$

which in turn follows from the basic inequality

$$\frac{1}{m}<\int_{m-1}^{m}\frac{dx}{x}.$$

This completes the proof of Theorem 4.3. □

Since ln n and $log_2\, n$ differ by a multiplicative constant, equation (2.1) is proved.

Acknowledgment. The author thanks the referee for his helpful comments.

REFERENCES

[1] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 506–518.

[2] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math. 9 (1961), pp. 551–570.

[3] A. ITAI AND Y. SHILOACH, *Maximum flow in planar networks*, this Journal, 8 (1979), pp. 135–150.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0002 $01.00/0

ON MULTIPLICATION OF POLYNOMIALS MODULO A POLYNOMIAL*

S. WINOGRAD†

Abstract. The multiplicative complexity of the direct product of algebras A_P of polynomials modulo a polynomial P is studied. In particular, we show that if P and Q are irreducible polynomials then the multiplicative complexity of $A_P \times A_Q$ is 2 deg (P) deg (Q) − k, where k is the number of factors of P in the field extended by a root of Q.

Key words. complexity of computations, multiplicative complexity, product of polynomials

1. Introduction. Let $R(u)=\sum_{i=0}^{n-1} x_i u^i$ and $S(u)=\sum_{i=0}^{n-1} y_i u^i$ be two polynomials with indeterminates as coefficients. Let $P(u)=u^n+\sum_{i=0}^{n-1} a_i u^i$ be a polynomial with coefficients in a field G. (We will assume that G is of characteristic 0, even though the results hold whenever G has enough elements and the field extensions needed are separable.) The coefficients of the polynomial $T(u)=R(u)\cdot S(u) \bmod P(u)$ form a system of bilinear forms, which is designated by T_P. The multiplicative complexity of T_P was studied in [1] and [2]. It was shown there that the minimum number of multiplications needed to compute T_P is $2n-k$ where k is the number of irreducible polynomials which divide P. The results of the investigation of the complexity of T_P led to new algorithms for computing the cyclic convolution and the Discrete Fourier Transform ([3], [4]).

In this paper we will extend the results of [2] to multivariate polynomials. Let $R^1(u,v)=\sum_{j=0}^{k-1}\sum_{i=0}^{n-1} x_{ij} u^i v^j$ and $S^1(u,v)=\sum_{j=0}^{k-1}\sum_{i=0}^{n-1} y_{ij} u^i v^j$ be two polynomials with indeterminates as coefficients. Let P(u) and Q(v) be two polynomials of degree n and k, respectively with coefficients in G. In this paper we will investigate the multiplicative complexity of computing the coefficients of the $T^1(u,v)=R^1(u,v)\cdot S^1(u,v) \bmod P(u)$, Q(v). We will limit the scope of our investigation to the case that P(u) and Q(v) have no repeated roots. It shall be clear from the results that they are not limited to polynomials of two variables, but that they cover the more general multivariate case.

As an example of the class of bilinear forms under consideration, let P(u) be the polynomial $P(u)=u^2+1$ and Q(v) be the polynomial $Q(v)=v^2+v+1$. In this case we want to compute the coefficients of

$$
\begin{gathered}
(x_{0,0}+x_{1,0}u+x_{0,1}v+x_{1,1}uv)\\
(y_{0,0}+y_{1,0}u+y_{0,1}v+y_{1,1}uv)\\
\bmod (u^2+1,\ v^2+v+1).
\end{gathered}
$$

Multiplying the two polynomials and substituting −1 for u^2, and $-v-1$ for v^2 we obtain that the four coefficients to be computed are $t_{0,0}$, $t_{1,0}$, $t_{0,1}$, and $t_{1,1}$ where

$$
\begin{pmatrix} t_{0,0}\\ t_{1,0}\\ t_{0,1}\\ t_{1,1}\end{pmatrix}=
\begin{pmatrix}
x_{0,0} & -x_{1,0} & x_{0,1} & x_{1,1}\\
x_{1,0} & x_{0,0} & -x_{1,1} & -x_{0,1}\\
x_{0,1} & -x_{1,1} & x_{0,0}-x_{0,1} & -x_{1,0}+x_{1,1}\\
x_{1,1} & x_{0,1} & x_{1,0}-x_{1,1} & x_{0,0}-x_{0,1}
\end{pmatrix}
\begin{pmatrix} y_{0,0}\\ y_{1,0}\\ y_{0,1}\\ y_{1,1}\end{pmatrix}.
$$

The structure of this matrix becomes more apparent if we "block" it into 2 × 2 blocks. In

* Received by the editors October 13, 1977, and in revised form April 27, 1979.

† IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

the block form we have to compute

$$\begin{pmatrix} X_0 & -X_1 \\ X_1 & X_0 - X_1 \end{pmatrix}\begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix}$$

which is the coefficient of $(X_0+X_1v)(Y_0+Y_1v) \bmod (v^2+v+1)$, while each of the blocks has the structure of multiplication modulo u^2+1. This structure suggests denoting this system of bilinear forms by $T_{u^2+1} \times T_{v^2+v+1}$. Another reason for the tensor product notation comes from viewing the system of bilinear forms as multiplication of two "general" elements in an algebra. By slight abuse of notation we will denote by T_{u^2+1} the algebra of linear polynomials with multiplication modulo u^2+1, and by T_{v^2+v+1} the algebra of linear polynomials with multiplication v^2+v+1. The algebra of the four bilinear forms under consideration is $T_{u^2+1} \times T_{v^2+v+1}$.

The example given above illustrates the subject of this paper. That is, to determine the complexity of $T_P \times T_Q$ where P and Q are polynomials such that neither of them has a repeated root. The naïve way of computing $T_P \times T_Q$ is to start with (non commutative) algorithms A and B computing T_P and T_Q respectively, and use them to build the algorithm $A \times B$ computing $T_P \times T_Q$. (This type of construction was used in [3], [4], and [5]). It was shown in [2] that $A \times B$ does not necessarily use the minimum number of multiplications even though A and B do. The specific example given in [2] shows that while the minimum number of multiplications to compute T_{u^2+1} over the rationals is 3, the multiplicative complexity of $T_{u^2+1} \times T_{v^2+1}$ is 6 (and not 9).

In the next section we will show that if P and Q are irreducible polynomials (in G) then the multiplicative complexity (over G) of $T_P \times T_Q$ is 2 deg(P)deg(Q)-m where m is the number of factors of P in the field G extended by a root of Q. More specifically, we will show that if β is a root of Q and if in $G(\beta)$ $P = \prod_{i=1}^{m} P_i$ then $T_P \times T_Q$ is isomorphic to the direct sum $T_{V_1} + T_{V_2} + \cdots + T_{V_m}$ where V_i is an irreducible polynomial of degree $\deg(P_i)\deg(Q)$, that means that using only additions and scalar multiplication the problem of computing $T_P \times T_Q$ can be transformed to computing m independent problems $T_{V_1}, T_{V_2} \cdots, T_{V_m}$.

If P is a polynomial without repeated roots, we can write $P = \prod_{i=1}^{k} P_i$ where the P_i's are (distinct) irreducible polynomials. By the Chinese remainder theorem, T_P is isomorphic to $T_{P_1} + T_{P_2} + \cdots + T_{P_k}$. Thus the result mentioned above also yields the complexity of $T_P \times T_Q$ for any two polynomials without repeated roots; and even more generally that of $T_{P_1} \times T_{P_2} \cdots \times T_{P_s}$, whenever none of the polynomials P_i have repeated roots.

By taking $P_i = u^{n_i}-1$ we thus obtain the multiplicative complexity of $n_1 \times n_2 \times \cdots \times n_s$ s-dimensional cyclic convolution, and in a way analogous to [4] new algorithms for multi-dimensional discrete Fourier transform.

2. Results. Let A be a system of bilinear forms $z_k = \sum_{j=1}^{s} \sum_{i=1}^{r} a_{ijk}\ x_i\ y_j$, $k = 1, 2, \cdots .t$. Let $R_{r\times r}$, $S_{s\times s}$, and $T_{t\times t}$ be nonsingular matrices. Substitution $x_i = \sum_{l=1}^{r} R_{i,l}\, x_i^1$, $y_j = \sum_{l=1}^{s} S_{j,l}\, y_l^1$ into A, and computing $z_k^1 = \sum_{l=1}^{t} T_{k,l}\, z_l$ we obtain a new system of bilinear forms A′ defined by $z_k^1 = \sum_{j=1}^{s} \sum_{i=1}^{r} a_{ijk}^1 x_i^1 y_j^1$, $k = 1, 2, \cdots, t$. Clearly every algorithm for computing A can be transformed into an algorithm for computing A′ (and vice versa) using only additions and scalar multiplication.

Definition. Two systems of bilinear forms A and A′ are called equivalent if and only if A′ is obtainable from A by the construction above. In other words, A and A′ are equivalent if and only if the tensor (a'_{ijk}) is obtainable from (a_{ijk}) by change of bases.

Let F be a finite extension of the field G and let $e_1, e_2, \cdots, e_n$ be a basis of F. Let

$\sum_{i=1}^{n} z_i e_i = (\sum_1^n x_i e_i)\ (\sum_{i=1}^{n} y_i e_i)$, then the z_i's form a system of bilinear forms which is denoted by $T_{F,\{e_i\}}$. If $e_1^1, e_2^1, \cdots, e_n^1$ is another basis of F then $T_{F,\{e_i^1\}}$, is equivalent to $T_{F,\{e_i\}}$. That means that up to equivalence we can assign a system of bilinear forms T_F to every finite extension F of G.

This assignment of a system of bilinear forms can be extended to any finite dimensional algebra. Let A be an algebra and let $e_1, e_2, \cdots, e_n$ be a basis, then if $\sum_{i=1}^{n} z_i e_i = (\sum_{i=1}^{n} x_i e_i)\ (\sum_{i=1}^{n} y_i e_i)$ the z_i's are a system of bilinear forms. Again, a change of basis yields an equivalent system, and we can therefore denote the system of bilinear forms by TA (without reference to the basis). We will continue to denote the system of bilinear forms of the algebra of polynomials modulo P by T_P.

If P and Q are polynomials, and A_P, A_Q are the algebras of polynomials modulo P and Q respectively, then T_P+T_Q is the system of bilinear forms associated with the algebra A_P+A_Q the direct sum of A_P and A_Q. That is, T_P+T_Q is the system of bilinear forms consisting of the systems T_P and T_Q on a disjoint set of indeterminates. The system $T_P \times T_Q$ is the one associated with the algebra $A_P \times A_Q$ the direct product of A_P and A_Q.

The main theorem of the paper is:

THEOREM. *Let* P *and* Q *be irreducible polynomials over a field* G, *and let β be a root of* Q, *and over* $G(\beta)$ *let* $P=\prod_{i=1}^{m} P_i$ *where the* P_i*'s are irreducible. Let* α_i *be a root of* P_i. *Then* $T_P \times T_Q$ *is equivalent to* $T_{G(\beta,\alpha_1)}+T_{G(\beta,\alpha_2)}+\cdots+T_{G(\beta,\alpha_m)}$, *and the minimum number of multiplications needed to compute* $T_P \times T_Q$ *is* 2deg(P)deg(Q)-m. *Moreover, every minimal algorithm computes each of the* $T_{G(\beta,\alpha_i)}$*'s separately.*

Proof. The proof of this theorem can be obtained directly from the main result of [2]. However, we will give a separate (and longer) proof which also shows how to derive the algorithms.

Let p = deg (P) and q = deg (Q), then T_P is $A(x)\mathbf{y}$ where $A(x)$ is a $p \times p$ matrix whose entries are linear forms of $\{x_0, x_1. \cdots, x_{p-1}\}$ and $\mathbf{y}$ is the (column) vector $(y_0, y_1, \cdots, y_{p-1})^T$. Similarly, T_Q is $B(x^1)\mathbf{y}^1$ where $B(x^1)$ is a $q \times q$ matrix whose entries are linear forms of $\{x_0^1, x_1^1, \cdots, x_{q-1}^1\}$ and $\mathbf{y}^1$ is the (column) vector $(y_0^1, y_1^1, \cdots, y_{q-1}^1)^T$. The system of bilinear forms $T_p \times T_q = C(x_i^{(j)})\mathbf{y}^{(k)}$ is obtained from $A(x)\mathbf{y}$ and $B(x^1)\mathbf{y}^1$ by substituting the matrix $B(x^{(i)})$ for every occurrence of x_i in $A(x)$ and the vector $\mathbf{y}^{(j)}=(y_0^{(j)}, y_1^{(j)}, \cdots, y_{q-1}^{(j)})^T$ for y_j in $\mathbf{y}$. (The indeterminates $\{x_j^{(i)}\}$ are disjoint, and so are $\{y_j^{(i)}\}$). As this construction is the key to the proof, the reader may want to go back to the example of the introduction. The problem of computing the coefficients of $(x_{0,0}+x_{1,0}u+x_{0,1}v+x_{1,1}uv)\ (y_{0,0}+y_{1,0}u+y_{0,1}v+y_{1,1}uv)$ mod (u^2+1, v^2+v+1) can be viewed as multiplying $(X_0+X_1v)(Y_0+Y_1v)$ mod v^2+v+1, where $X_0=x_{0,0}+x_{1,0}u$; $X_1=x_{0,0}+x_{1,1}u$; $Y_0=y_{0,0}+y_{1,0}u$; $Y_1=y_{0,1}+y_{1,1}u$; and X_0, X_1, Y_0, Y_1 are viewed as elements of T_{u^2+1}.

If we denote the pq bilinear forms of $T_P \times T_Q$ by $z_{i,j}$ $0 \leqq i < p$, $0 \leqq j < q$, we can give an alternative description of $T_P \times T_Q$ in terms of T_P and T_Q. A "general" element in $G(\beta)$ (where β is a root of Q) is $\mathbf{x}^{(i)}=\sum_{j=0}^{q-1} x_j^{(i)}\beta^j$. Let the lth row of $A(x)y$ be $\sum_{i,j=0}^{p-1} a_{i,j,l}\ x_i y_j$, then z_{lk} is the coefficient of β^k in $\sum_{i,j=0}^{p-1} a_{ij,l}\ \mathbf{x}^{(i)}\mathbf{y}^{(j)}$. This follows from the observation that the k^{th} row of $B(x^{(i)})y^{(j)}$ is the coefficient of β^k of $\mathbf{x}^{(i)}\mathbf{y}^{(j)}$. Therefore $T_P \times T_Q$ can be viewed as $A(\mathbf{x})\mathbf{y}$ where $A(\mathbf{x})$ is obtained from $A(x)$ by replacing x_i by $\mathbf{x}^{(i)}$, and $\mathbf{y}$ is obtained from y by replacing y_i by $\mathbf{y}^{(i)}$. The advantage of $A(\mathbf{x})$ is that we can use the field $G(\beta)$ as the field of constants, and can describe, for any $g \in G(\beta)$ the components of $g \cdot \mathbf{x}^{(i)}$ in terms of linear forms over G of the $x_j^{(i)}$'s.

Since $P=\prod_{i=1}^{m} P_i$ in $G(\beta)$, the Chinese remainder theorem states that $A(\mathbf{x})\mathbf{y}$ is equivalent (over $G(\beta)$) to $\bar{T}_{P_1}+\bar{T}_{P_2}+\cdots+\bar{T}_{P_m}$, where $\bar{T}_{P_i}$ is T_{P_i} with "general" elements of $G(\beta)$ replacing the indeterminates. Reversing the process and substituting

$B(x^{(i)})$ for $\bar{x}^{(i)}$ in $\bar{T}_{P_j}$ we obtain that $T_P \times T_Q$ is equivalent to $T_{G(\beta,\alpha_1)} + T_{G(\beta,\alpha_2)} + \cdots + T_{G(\beta,\alpha_m)}$.

Let V_j be a polynomial whose root generates $G(\beta, \alpha_j)$, then $T_{G(\beta,\alpha_j)}$ is equivalent to T_{V_j}, and consequently $T_P \times T_Q$ is equivalent to $T_{V_1} + T_{V_2} + \cdots + T_{V_m}$.

It was shown in [2] that the minimum number of multiplications needed to compute $T_{V_1} + T_{V_2} + \cdots + T_{V_m}$ is $\sum_{i=1}^{m} (2 \deg (V_i)\text{-}1) = \sum_{i=1}^{m} (2 \deg (P_i) \deg (Q)\text{-}1) = 2 \deg (P) \deg (Q)\text{-}m$. It was further shown in [2] that every minimal algorithm necessarily computes each of the T_{V_i}'s separately. This proves the theorem.

Since $T_P \times T_Q$ is equivalent to $T_Q \times T_P$ we obtain the following corollary:

COROLLARY. *Let* P *and* Q *be irreducible polynomials in* G, *let* α *be a root of* P *and* β a root of Q, *then the number of factors of* P *in* $G(\beta)$ *is the same as the number of roots of* Q *in* $G(\alpha)$.

The corollary can be strengthened. Let α be a root of P, then $Q = \prod_{i=1}^{m} Q_i$ in $G(\alpha)$. Let β_i be a root of Q_i $i = 1, 2, \cdots, m$, then by the result of [2] we obtain:

COROLLARY. *The set of fields* $\{G(\beta, \alpha_i)\}_{i=1}^{m}$ *is equivalent to* $\{G(\alpha, \beta_i)\}_{i=1}^{m}$.

In applications to multidimensional cyclic convolution and multidimensional discrete Fourier transform we use the field of rational numbers $\mathbb{Q}$ for G, and the polynomials P and Q are cyclotomic polynomials. As usual we will denote the minimal polynomial whose root is the h^{th} root of unity by Φ_h.

COROLLARY. *Let* p *and* q *be prime numbers; then*

1. $T_{\Phi_{p^n}} \times T_{\Phi_{p^m}}$ $(m \geqq n)$ *is equivalent to direct sum* $T_{\Phi_{p^m}} + T_{\Phi_{p^m}} + \cdots + T_{\Phi_{p^m}}$ $((p-1)p^{n-1}$ *times*), *and therefore its multiplicative complexity is* $2p^{m+n} - 4p^{m+n-1} - p^n + p^{n-1}$.

2. $T_{\Phi_{p^n}} \times T_{\Phi_{q^n}}$ *is equivalent to* $T_{\Phi_{p^n q^m}}$ (p, q *distinct primes*).

Proof. Let α_h be the h^{th} root of unity, then $\mathbb{Q}(\alpha_{p^m})$ splits Φ_{p^n} $(m \geqq n)$ into $(p-1)p^{n-1}$ linear factors. This proves the first part of the corollary. To prove the second half, assume that Φ_{p^n} can be factored into $P_1 \cdot P_2 \cdot \ldots \cdot P_k$ in $\mathbb{Q}(\alpha_{q^m})$, and without loss of generality assume that α_{p^n} is a root of P_1. Since p and q are relatively prime there exist integers r and s such that $rp^n + sq^m = 1 \bmod p^n q^m$. Using the fact that $\alpha_{ab}^{a} = \alpha_b$ we obtain that the field $\mathbb{Q}(\alpha_{q^m}, \alpha_{p^n})$ includes the element $\alpha_{q^m}^{r} \alpha_{p^n}^{s} = \alpha_{p^n q^m}^{rp^n} \alpha_{p^n q^m}^{sq^m} = (\alpha_{p^n q^m})^{rp^n + sq^m} = \alpha_{p^n q^m}$. So the dimension of $\mathbb{Q}(\alpha_{q^m}, \alpha_{p^n})$ is at least $(p-1)p^{n-1}(q-1)q^{m-1}$. On the other hand the dimension of $\mathbb{Q}(\alpha_{q^m}, \alpha_{p^n})$ is deg. (Φ_{q^m}) deg $(P_1) \leqq$ deg (Φ_{q^m}) deg $(\Phi_{q^n}) = (p-1)p^n(q-1)q^m$. Therefore equality has to hold, $P_1 = \Phi_{p^n}$, and $\mathbb{Q}(\alpha_{q^m}, \alpha_{p^n}) = \mathbb{Q}(\alpha_{p^n q^m})$.

Using this corollary we can analyze the multiplicative complexity of multiplication of group algebras for commutative groups. We will illustrate by analyzing the multiplicative complexity of a $p^n \times q^m$ two-dimensional cyclic convolution. By the Chinese remainder theorem $T_{u^{p^n}-1}$ (where p is a prime) is equivalent to $T_{\Phi_{p^n}} + T_{\Phi_{p^{n-1}}} + \cdots + T_{p^0}$. If we use $r \cdot T_P$ to denote the *r*-fold direct sum $T_P + T_P + \cdots + T_p$, we obtain:

COROLLARY. *Let* p *and* q *be primes, then:*

1. $T_{u^{p^n}-1} \times T_{v^{p^m}-1}$ $(m \geqq n)$ *is equivalent to* $p^n \cdot (\sum_{i=n+1}^{m} T_{\Phi_{p^i}}) + \sum_{i=1}^{n} (p^i + p^{i-1}) \cdot T_{\Phi_{p^i}} + T_{u-1} \times T_{v-1}$, *and therefore its multiplicative complexity is* $2p^{m+n} - (m-n)p^n - (p^{n+1} + p^n - 2)/(p-1)$.

2. $T_{u^{p^n}-1} \times T_{v^{q^m}-1}$ *is equivalent to* $\sum_{j=0}^{m} \sum_{i=0}^{n} T_{\Phi_{p^i q^j}}$, (*where* T_{Φ_1} *denotes* T_{u-1}), *and therefore its multiplicative complexity is* $2p^n q^m - (m+1)(n+1)$.

Proof. $T_{u^{p_n}-1}$ is equivalent to $\sum_{i=0}^{n} T_{\Phi_p}$, and therefore $T_{u^{p^m}-1} \times T_{u^{p^n}-1}$ is equivalent to

$$\left(\sum_{j=0}^{m} T_{\Phi_{p^j}}\right) \times \left(\sum_{i=0}^{n} T_{\Phi_{p^i}}\right) = \left(\sum_{j=n+1}^{m} T_{\Phi_{p^j}}\right) \times \left(\sum_{i=0}^{n} T_{\Phi_{p^i}}\right) + \left(\sum_{j=0}^{n} T_{\Phi_{p^j}}\right) \times \left(\sum_{i=0}^{n} T_{\Phi_{p^j}}\right).$$

By the previous corollary it is equivalent to $p^n \cdot \sum_{j=n+1}^{m} T_{\Phi_{p^j}} + (\sum_{j=0}^{n} T_{\Phi_{p^j}}) \times (\sum_{i=0}^{n} T_{\Phi_{p^l}}) = p^n \cdot (\sum_{j=n+1}^{m} T_{\Phi_{p^j}}) + T_{\Phi_{p^n}} \times T_{\Phi_{p^n}} + 2 \cdot T_{\Phi_{p^n}} \times (\sum_{j=0}^{n-1} T_{\Phi_{p^i}}) + (\sum_{j=0}^{n-1} T_{\Phi_{p^j}}) \times (\sum_{i=0}^{n-1} T_{\Phi_{p^t}})$.

$$p^n \cdot \left(\sum_{j=n+1}^{m} T_{\Phi_{p^j}}\right) + (p-1)p^{n-1} \cdot T_{\Phi_{p^n}} + 2p^{n-1} \cdot T_{\Phi_{p^n}} + \left(\sum_{j=0}^{n-1} T_{\Phi_{p^j}}\right) \times \left(\sum_{i=0}^{n-1} T_{\Phi_{p^i}}\right)$$

$$= p^n \cdot \left(\sum_{j=n+1}^{m} T_{\Phi_{p^i}}\right) + (p^n + p^{n-1}) \cdot T_{\Phi_{p^n}} + \left(\sum_{j=0}^{n-1} T_{\Phi_{p^j}}\right) \times \left(\sum_{i=0}^{n-1} T_{\Phi_{p^i}}\right).$$

Continuing this way we obtain that $T_{u^{p^m}-1} \times T_{v^{p^n}-1}$ is equivalent to $p^n \cdot \sum_{j=n+1}^{m} T_{\Phi_{p^j}} + \sum_{l=1}^{n} (p^i + p^{i-1}) \cdot T_{\Phi_{p^i}} + T_{u-1} + T_{v-1}$. Therefore, the multiplicative complexity is

$$p^n \sum_{j=n+1}^{m} (2(p-1)p^{j-1} - 1) + \sum_{i=1}^{n} (p^i + p^{i-1})(2(p-1)p^{i-1} - 1) + 1$$

$$= 2(p-1)p^{2n} \sum_{j=0}^{m-n-1} p^j - (m-n)p^n + 2(p^2-1) \sum_{i=1}^{n} p^{2(i-1)} - (p+1) \sum_{i=1}^{n} p^{i-1} + 1$$

$$= 2p^{2n}(p^{m-n} - 1) - (m-n)p^n + 2(p^{2n-1}) - \frac{(p+1)(p^n-1)}{p-1} + 1$$

$$= 2p^{m+n} - (m-n)p^n - \frac{p^{n+1} + p^n - 2}{p-1}.$$

This proves the first half of the corollary.

To prove the second half we observe that $T_{u^{p^n}-1} \times T_{v^{q^m}-1}$ is equivalent to $(\sum_{i=0}^{n} T_{\Phi_{p^i}}) \times (\sum_{j=0}^{m} T_{\Phi_{q^j}}) = \sum_{i=1}^{m} \sum_{j=1}^{m} T_{\Phi_{p^i}} \times T_{\Phi_{q^j}} + \sum_{i=1}^{n} T_{\Phi_{p^i}} \times T_{v-1} + \sum_{j=1}^{n} T_{\Phi_{q^j}} \times T_{u-1} + T_{u-1} \times T_{v-1}$. By the previous corollary it is equivalent to $\sum_{i=1}^{n} \sum_{j=1}^{m} T_{\Phi_{p^i q^j}} + \sum_{i=1}^{n} T_{\Phi_{p^i}} + \sum_{j=1}^{m} T_{\Phi_{p^j}} + T_{u-1} \times T_{v-1}$. Because the degree of $T_{\Phi_{p^i q^i}}$ is $(p-1)(q-1)p^{i-1}q^{j-1}$ $(i, j \geqq 1)$ we obtain that the multiplicative complexity is

$$\sum_{i=1}^{n} \sum_{j=1}^{m} (2(p-1)(q-1)p^{i-1}q^{j-1} - 1) + \sum_{i=1}^{n} (2(p-1)p^{i-1} - 1) + \sum_{j=1}^{m} (2(q-1)q^{j-1} - 1) + 1$$

$$= 2p^n q^m - (m+1)(n+1).$$

REFERENCES

[1] C. M. FIDUCCIA AND Y. ZALCSTEIN, *Algebras having linear multiplicative complexities*, J. Assoc. Comput. Mach. 24 (1977), pp. 311–331.

[2] S. WINOGRAD, *Some bilinear forms whose multiplicative complexity depends on the fields of constants*, Mathematical System Theory, 10 (1976/77), pp. 169–180.

[3] R. C. AGARWAL AND J. W. COOLEY, *New Algorithms for Digital Convolution*. IEEE Trans. on Acoustics, Speech and Signal Processing, to appear.

[4] S. WINOGRAD, *On Computing the Discrete Fourier Transform*, Proc. Nat. Acad. Sci., U.S.A., 73 (1976) pp. 1005–1006.

[5] V. STRASSEN, *Gaussian Elimination is Not Optimal*, Numer. Math., 13 (1969), pp. 354–356.

© 1980 Society for Industrial and Applied Mathematics
0097-5397/80/0902-0003 $01.00/0

ON THE EVALUATION OF POWERS AND MONOMIALS*

NICHOLAS PIPPENGER†

Abstract. Let $y_1, \cdots, y_p$ be monomials over the indeterminates $x_1, \cdots, x_q$. For every $y = (y_1, \cdots, y_p)$ there is some minimum number $L(y)$ of multiplications sufficient to compute $y_1, \cdots, y_p$ from $x_1, \cdots, x_q$ and the identity 1. Let $L(p, q, N)$ denote the maximum of $L(y)$ over all y for which the exponent of any indeterminate in any monomial is at most N. We show that if $p = (N+1)^{o(q)}$ and $q = (N+1)^{o(p)}$, then $L(p, q, N) = \min\{p, q\} \log N + H/\log H + o(H/\log H)$, where $H = pq \log(N+1)$ and all logarithms have base 2.

Key words. addition chain, computational complexity, monomial, power

1. Introduction. The result described in the abstract generalizes a number of previous results and solves a number of open problems. In 1937, Scholz [7] raised the problem of determining $L(1, 1, N)$ (computing one power of one indeterminate) and observed that

$$\log N \leqq L(1, 1, N) \leqq 2 \log N.$$

In 1939, Brauer [2] obtained the asymptotic formula

$$L(1, 1, N) \sim \log N,$$

and in 1960, Erdös [3] improved this to

$$L(1, 1, N) = \log N + \frac{\log(N+1)}{\log\log(N+1)} + o\left(\frac{\log(N+1)}{\log\log(N+1)}\right).$$

In 1963, Bellman [1] raised the problem of determining $L(1, q, N)$ (computing one monomial in several indeterminates), and in 1964, Straus [8] showed that

$$L(1, q, N) \sim \log N$$

for each fixed q.

In 1969, Knuth [4] (Section 4.6.3, Exercise 32) raised the problem of determining $L(p, 1, N)$ (computing several powers of one indeterminate), and in 1976, Yao [9] showed that

$$L(p, 1, N) \sim \log N$$

for each fixed p.

In a preliminary version of this paper [5], the author raised the problem of determining $L(p, q, N)$ and showed that if $p = 2^{o(q)}$ and $q = 2^{o(p)}$, then

$$L(p, q, 1) \sim pq/\log(pq).$$

In this paper we shall prove the following

THEOREM.

$$L(p, q, N) = v \log N + \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(w),$$

where $v = \min\{p, q\}$, $H = pq \log(N+1)$, *and* $w = \max\{p, q\}$. *The expression* $U(\cdots)$ *denotes a factor of the form* $\exp O(\cdots)$; *if the quantity represented by the ellipsis tends to* 0, $U(\cdots)$ *is equivalent to* $1 + O(\cdots)$.

* Received by the editors November 1, 1978, and in revised form April 30, 1979.

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

Since $p=(N+1)^{o(q)}$ and $q=(N+1)^{o(p)}$ together imply (in fact, are equivalent to) $w=o(H/\log H)$, this theorem implies the result described in the abstract, as well as all the other asymptotic formulae cited above. The proof of the theorem is in two parts: a lower bound and an upper bound. The lower bound, presented in § 2, owes several ideas to the paper [3] of Erdös cited above. The upper bound, presented in § 3, would be the more difficult part of the proof if we had to start from scratch. In another paper[6], however, the author has given a result (also growing out of the preliminary version [5]) which allows the upper bound to be deduced as a corollary.

1.1. Reformulation of the problem. It is both traditional and convenient to reformulate the problem at hand in additive rather than multiplicative notation.

Let $q\geqq 1$ be a integer. A sequence

$$f=(f_1,\cdots,f_q)$$

of nonnegative integers will be called a *(q-dimensional) vector*, and $f_1,\cdots,f_q$ will be called its *components*. The vector

$$x_0=(0,\cdots,0)$$

will be called the *zero* vector, and the vectors

$$x_1=(1,\cdots,0),$$
$$\cdots$$
$$x_q=(0,\cdots,1)$$

will be called *unit* vectors. If

$$f=(f_1,\cdots,f_q)$$

and

$$g=(g_1,\cdots,g_q),$$

the vector

$$f+g=(f_1+g_1,\cdots,f_q+g_q)$$

will be called the *sum* of f and g.

Let $p\geqq 1$ be an integer. A sequence

$$y=(y_1,\cdots,y_p)$$

of (q-dimensional) vectors will be called a *(p-by-q) matrix*, and $y_1,\cdots,y_p$ will be called its *rows*.

Let $l\geqq 1$ be an integer. A sequence

$$z=(z_1,\cdots,z_l)$$

of vectors will be called a *chain*, and $z_1,\cdots,z_l$ will be called its *rows*, if each vector z_k $(1\leqq k\leqq l)$ is (1) the zero vector, (2) one of the unit vectors, or (3) the sum of two of the vectors $z_1,\cdots,z_{k-1}$ that precede it in the sequence (these two vectors need not be distinct). The zero and unit vectors will be called *basic* vectors; the others will be called *auxiliary* vectors. The number of basic vectors will be denoted by m; the number of auxiliary vectors will be denoted by n and called the *length* of the chain.

Let $N\geqq 1$ be an integer. We shall say that a vector is $(N+1)$-*ary* if all its components are in the set $\{0,1,\cdots,N\}$, and that a matrix is $(N+1)$-*ary* if all its vectors are $(N+1)$-ary.

We shall say that a chain z *computes* a matrix y if each vector $y_i (1 \leqq i \leqq p)$ appears as one of the vectors $z_k (1 \leqq k \leqq l)$. If y is a matrix, $L(y)$ will denote the minimum possible length of a chain computing y, and $L(p, q, N)$ will denote the maximum of $L(y)$ over all p-by-q $(N+1)$-ary matrices y.

2. The lower bound. In this section we shall prove the lower bound

$$L(p, q, N) \geqq v \log N + \frac{H}{\log H} U\left(\frac{\log \log H}{\log H}\right) + O(w).$$

2.1. The easy case. Consider first the case

$$v \log N \leqq \frac{H \log \log H}{(\log H)^2}.$$

In this case the first term, $v \log N$, is absorbed by the U-factor of the second term,

$$\frac{H}{\log H} U\left(\frac{\log \log H}{\log H}\right) = \frac{H}{\log H} + O\left(\frac{H \log \log H}{(\log H)^2}\right).$$

Thus it will suffice to show

$$L(p, q, N) \geqq \frac{H}{\log H} U\left(\frac{\log \log H}{\log H}\right) + O(w).$$

If

$$w \geqq \frac{H}{\log H},$$

the desired bound is trivial; hence we shall assume

$$w \leqq \frac{H}{\log H}.$$

If

$$L(p, q, N) \geqq \frac{H}{\log H},$$

we are done; hence we shall assume

$$L(p, q, N) \leqq \frac{H}{\log H}.$$

It follows that we may also assume

$$\begin{aligned} l &= m + n \\ &\leqq q + 1 + L(p, q, N) \\ &= O\left(\frac{H}{\log H}\right). \end{aligned}$$

Let us consider a chain and assign to each vector in it a number called its *depth*. The basic vectors are assigned the depth 0. For $d = 1, 2, \cdots$, if a vector is the sum of two preceding vectors that both have depth at most $d-1$, but is not the sum of two preceding vectors that both have depth at most $d-2$, then it is assigned the depth d. By induction, this assigns depths uniquely to all the vectors in the chain.

Let us impose upon the set of all vectors a definite total order, which will be called the *standard order.*

We shall say that a chain is *standard* if its rows are all distinct, rows of lower depth precede those of higher depth, and rows of equal depth appear in the standard order.

LEMMA 2.1–1. *If a matrix is computed by a chain z, then it is also computed by a standard chain z' of no greater length.*

Proof. Given a chain z, consider the set of all vectors appearing in z. Remove from this set all the basic vectors and arrange them in the standard order to form a chain. Then remove from the set all the vectors that are the sum of two vectors currently in the chain, arrange them in the standard order, and append them to the end of the chain. Repeat this process until no more vectors can be removed. When the process terminates, the set must be empty, for if it contains any vectors, at the very least the one that appears earliest in z can be removed. The process thus yields a chain z' which is standard by construction, which contains every vector that appears in z (so that, in particular, it computes every matrix computed by z), and which contains no other vectors (so that, in particular, it has no greater length than z). □

By virtue of this lemma, we may henceforth restrict our attention to standard chains, and all chains will be assumed to be standard even if this is not explicitly mentioned.

We shall say that a matrix is *standard* if its rows are distinct and appear in the standard order. Henceforth we shall restrict our attention to standard matrices, and all matrices will be assumed to be standard even if this is not explicitly mentioned.

LEMMA 2.1–2. *There are at least*

$$2^H U(w \log H)$$

matrices.

Proof. There are $(N+1)^q$ rows that can appear in a matrix, and thus

$$\binom{(N+1)^q}{p}$$

ways to choose p distinct rows to form a matrix. Using the bound

$$\binom{A}{B} = A(A-1) \cdots (A-B+1)/B(B-1) \cdots 1 \geqq (A/B)^B$$

we obtain

$$\begin{aligned}\binom{(N+1)^q}{p} &\geqq (N+1)^{pq}/p^p \\ &= 2^H U(p \log p) \\ &= 2^H U(w \log H)\end{aligned}$$

matrices. □.

LEMMA 2.1–3. *For some value of $n \leqq L(p, q, N)$, there are at least*

$$2^H U(w \log H)$$

chains.

Proof. Each matrix is computed by some chain of length at most $L(p, q, N)$. Each of these chains computes at most

$$\begin{aligned}\binom{l}{p} &\leqq l^p \\ &= U(p \log l) \\ &= U(w \log H)\end{aligned}$$

matrices, so there are at least

$$2^H U(w \log H)/U(w \log H) = 2^H U(w \log H)$$

chains of length at most $L(p, q, N)$. Each chain has one of at most

$$L(p, q, N) = U(\log H)$$

possible lengths, so for some length $n \leqq L(p, q, N)$, there are at least

$$2^H U(w \log H)/U(\log H) = 2^H U(w \log H)$$

chains. □

With each chain z we shall associate an object, which will be called a *code*, constructed as follows. Each basic vector in z is x_j for some j such that $0 \leqq j \leqq q$. Let $\mathcal{M}$ be the subset of $\{0, 1, \cdots, q\}$ that contains the m values of j corresponding to the basic vectors in z. Each auxiliary vector z_k in z is $z_{a_k} + z_{b_k}$ for some a_k and b_k such that $1 \leqq a_k \leqq b_k \leqq k-1$. Let $\mathcal{N}$ be a subset of $\{1, \cdots, l\} \times \{1, \cdots, l\}$ that contains n ordered pairs (a_k, b_k), one corresponding to each auxiliary vector in z. The ordered pair $(\mathcal{M}, \mathcal{N})$ will be the code associated with z.

LEMMA 2.1–4. *A chain is uniquely determined by its code.*

Proof. Let $(\mathcal{M}, \mathcal{N})$ be a code. From $\mathcal{M}$, determine the set of basic vectors; arrange these in the standard order to form a chain. Remove from $\mathcal{N}$ the pairs (a, b) for which b is less than or equal to the number of vectors currently in the chain. For each such pair, compute the vector $z_a + z_b$; arrange these vectors in the standard order and append them to the end of the chain. Repeat this process until no more pairs can be removed. Clearly only the resulting chain can have the code $(\mathcal{M}, \mathcal{N})$. □

LEMMA 2.1–5. *For any value of $n \leqq L(p, q, N)$, there are at most*

$$(H^2/n)^n U(n) U(w)$$

chains.

Proof. For any m and n, there are at most

$$\binom{q+1}{m}\binom{l^2}{n}$$

codes, since the two factors bound the number of ways of choosing $\mathcal{M}$ and $\mathcal{N}$, respectively. Using the bounds

$$\binom{A}{B} \leqq 2^A$$

and

$$\binom{A}{B} \leqq A^B/B! \leqq (Ae/B)^B$$

(where $e = 2.718 \cdots$ is the base of natural logarithms), we obtain

$$\begin{aligned}\binom{q+1}{m}\binom{l^2}{n} &\leqq 2^{q+1}(l^2 e/n)^n \\ &= (l^2/n)^n U(n) U(q) \\ &= (l^2/n)^n U(n) U(w).\end{aligned}$$

There are

$$\begin{aligned}q+1 &= U(\log q) \\ &= U(\log w)\end{aligned}$$

possible values of m, and for each value of $n \leqq L(p, q, N)$,

$$\begin{aligned}l &= O(H) \\ &\leqq HU(1).\end{aligned}$$

Thus, for any value of $n \leqq L(p, q, N)$, there are at most

$$U(\log w)(H^2 U(1)^2/n)^n U(n) U(w) = (H^2/n)^n U(n) U(w)$$

codes.

Each chain is associated with some code, and at most one chain is associated with each code. Thus the bound just derived applies to chains as well as codes. □

We can now complete the proof. By Lemmas 2.1–3 and –5, there is a value of $n \leqq L(p, q, N)$ such that

$$(H^2/n)^n U(n) U(w) \geqq 2^H U(w \log H)$$

or, by taking logarithms,

$$2n \log H - n \log n + O(n) \geqq H + O(w \log H).$$

Ignoring the $n \log n$ term for the moment, this implies

$$2n \log H + O(n) \geqq H + O(w \log H)$$

or

$$(2n \log H) U\left(\frac{1}{\log H}\right) \geqq HU\left(\frac{w \log H}{H}\right).$$

This yields

$$n \geqq \frac{H}{2 \log H} U\left(\frac{1}{\log H}\right) U\left(\frac{w \log H}{H}\right)$$

or, by taking logarithms,

$$\log n \geqq \log H + O(\log \log H) + O\left(\frac{w \log H}{H}\right).$$

Multiplication by n yields

$$\begin{aligned}n \log n &\geqq n \log H + O(n \log \log H) + O\left(\frac{nw \log H}{H}\right) \\ &= n \log H + O\left(\frac{H \log \log H}{\log H}\right) + O(w).\end{aligned}$$

With this bound on the $n \log n$ term, the original inequality implies

$$n \log H + O(n) \geqq H + O\left(\frac{H \log \log H}{\log H}\right) + O(w \log H)$$

or

$$(n \log H) U\left(\frac{1}{\log H}\right) \geqq HU\left(\frac{\log \log H}{\log H}\right) + O(w \log H).$$

Thus

$$\begin{aligned} L(p, q, N) &\geqq n \\ &\geqq \frac{H}{\log H} U\left(\frac{\log \log H}{\log H}\right) + O(w), \end{aligned}$$

which is the desired lower bound.

2.2. The hard case. Consider now the case

$$v \log N \geqq \frac{H \log \log H}{(\log H)^2}.$$

Since $H = vw \log (N+1)$, we have

$$w \leqq \frac{(\log H)^2}{\log \log H}.$$

If

$$L(p, q, N) \geqq v \log N + \frac{H}{\log H}$$

we are done; hence we shall assume

$$L(p, q, N) \leqq v \log N + \frac{H}{\log H}.$$

It follows that we may also assume

$$\begin{aligned} l &= m + n \\ &\leqq q + 1 + L(p, q, N) \\ &= O(H). \end{aligned}$$

For any vector f and any $1 \leqq j \leqq q$, let us define

$$D(f, j) = -\sum_{1 \leqq i < j} f_i + f_j - \sum_{j < i \leqq q} f_i.$$

Thus $D(f, j)$ measures the extent to which the jth component of f exceeds all the other components combined.

We shall say that a vector f is a *j-vector* if

$$D(f, j) \geqq 1.$$

Clearly, a vector can be j-vector for at most one value of j.

Let z be a chain and let $z_k (m+1 \leqq k \leqq l)$ be an auxiliary j-vector in z. We shall say that z_j is *j-immediate* if it is equal to 2 times a preceding j-vector. Let

$$h = \lceil (\log H)^2 \rceil .$$

We shall say that z_k is *j-short* if it is not j-immediate but is the sum of two preceding j-vectors, between which fewer than h j-vectors intervene. Finally, we shall say that z_k is *j-long* if it is neither j-immediate nor j-short.

Let n_j, r_j, s_j, and t_j denote the numbers of j-vectors, j-immediate vectors, j-short vectors, and j-long vectors in z. Clearly,

$$n_j = r_j + s_j + t_j.$$

Let

$$\phi = (1+5^{1/2})/2 = 1.618 \cdots$$

be the golden ratio. Then

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

Let

$$\psi = h^{1/h} \leqq 3^{1/3} = 1.442 \cdots .$$

Then for $h \geqq 2$,

$$\psi^{-h-2} + \psi^{-1} \leqq 1.$$

For $h=2$, this is trivial to check. For $h \geqq 3$, it follows from

$$\psi^{-h-2} = h^{-1} \exp(-2h^{-1} \ln h),$$
$$\psi^{-1} = \exp(-h^{-1} \ln h),$$
$$\exp x \leqq 1/(1-x),$$

and

$$\ln h \geqq 1.$$

LEMMA 2.2–1. *For any chain z, any vector z_k in z, and any $1 \leqq j \leqq q$,*

$$D(z_k, j) \leqq 2^{r_j} \phi^{s_j + t_j}$$

and

$$D(z_k, j) \leqq 2^{r_j + s_j} \psi^{t_j}.$$

Proof. We shall proceed by induction on $n_j = r_j + s_j + t_j$. If $n_j = 0$, there are no auxiliary j-vectors. But if z_k is a basic vector or not a j-vector,

$$D(z_k, j) \leqq 1,$$

and the assertions of the lemma are trivial. Suppose then that $n_j \geqq 1$ and that z_k is an auxiliary j-vector. It follows that it must be j-immediate, j-short, or j-long.

If z_k is j-immediate, there exists $1 \leqq b \leqq k-1$ such that $z_k = 2z_b$. The vector z_b appears in a chain with at most $r_j + s_j + t_j - 1$ j-vectors, of which at most $r_j - 1$ are j-immediate. By inductive hypothesis,

$$D(z_b, j) \leqq 2^{r_j - 1} \phi^{s_j + t_j}$$

(since $2 \geqq \phi$), and so

$$\begin{aligned} D(z_k, j) &= 2D(z_b, j) \\ &\leqq 2^{r_i}\phi^{s_j+t_j}. \end{aligned}$$

If on the other hand z_k is not j-immediate, there exist $1 \leqq a < b \leqq k-1$ such that $z_k = z_a + z_b$. The vectors z_a and z_b both appear in a chain with at most $r_j + s_j + t_j - 1$ j-vectors, of which at most r_j are j-immediate. By inductive hypothesis,

$$D(z_a, j) \leqq 2^{r_i}\phi^{s_j+t_j-1}$$

and

$$D(z_b, j) \leqq 2^{r_i}\phi^{s_j+t_j-1}.$$

If z_b is not a j-vector,

$$D(z_b, j) \leqq 0$$

and

$$\begin{aligned} D(z_k, j) &= D(z_a, j) + D(z_b, j) \\ &\leqq D(z_a, j) \\ &\leqq 2^{r_i}\phi^{s_j+t_j-1} \\ &\leqq 2^{r_i}\phi^{s_j+t_j}. \end{aligned}$$

If on the other hand z_b is a j-vector, then z_a appears in a chain with at most $r_j + s_j + t_j - 2$ j-vectors, of which at most r_j are j-immediate. By inductive hypothesis,

$$D(z_a, j) \leqq 2^{r_j+t_j-2}$$

and so

$$\begin{aligned} D(z_k,j) &= D(z_a, j) + D(z_b, j) \\ &\leqq 2^{r_i}\phi^{s_j+t_j-2} + 2^{r_i}\phi^{s_j+t_j-1} \\ &= 2^{r_i}\phi^{s_j+t_j}. \end{aligned}$$

This proves the first assertion of the lemma.

If z_k is not j-long, there exist $1 \leqq a \leqq b \leqq k-1$ such that $z_k = z_a + z_b$. The vectors z_a and z_b both appear in a chain with at most $r_j + s_j + t_j - 1$ j-vectors, of which at most $r_j + s_j - 1$ are not j-long. By inductive hypothesis,

$$D(z_a, j) \leqq 2^{r_j+s_j-1}\psi^{t_j}$$

and

$$D(z_b, j) \leqq 2^{r_j+s_j-1}\psi^{t_j}$$

(since $2 \geqq \psi$), and so

$$\begin{aligned} D(z_k, j) &= D(z_a, j) + D(z_b, j) \\ &\leqq 2\,2^{r_j+s_j-1}\psi^{t_j} \\ &= 2^{r_j+s_j}\psi^{t_j}. \end{aligned}$$

If on the other hand z_k is j-long, there exist $1 \leqq a < b \leqq k-1$ such that $z_k = z_a + z_b$, and at least h j-vectors intervene between z_a and z_b. The vectors z_a and z_b both appear in a chain with at most $r_j+s_j+t_j-1$ j-vectors, of which at most r_j+s_j are not j-long. By inductive hypothesis,

$$D(z_a, j) \leqq 2^{r_j+s_j}\psi^{t_j-1}$$

and

$$D(z_b, j) \leqq 2^{r_j+s_j}\psi^{t_j-1}.$$

If z_b is not a j-vector,

$$D(z_b, j) \leqq 0$$

and

$$\begin{aligned} D(z_k, j) &= D(z_a, j) + D(z_b, j) \\ &\leqq D(z_a, j) \\ &\leqq 2^{r_j+s_j}\psi^{t_j-1} \\ &\leqq 2^{r_j+s_j}\psi^{t_j}. \end{aligned}$$

If on the other hand z_b is a j-vector, then z_a appears in a chain with at most $r_j+s_j+t_j-h-2$ j-vectors, of which at most r_j+s_j are not j-long. By inductive hypothesis,

$$D(z_a, j) \leqq 2^{r_j+s_j}\psi^{t_j-h-2}$$

and so

$$\begin{aligned} D(z_k, j) &= D(z_a, j) + D(z_b, j) \\ &\leqq 2^{r_j+s_j}\psi^{t_j-h-2} + 2^{r_j+s_j}\psi^{t_j-1} \\ &\leqq 2^{r_j+s_j}\psi^{t_j}. \end{aligned}$$

This proves the second assertion of the lemma. □

Let z be a standard chain and let z_k $(m+1 \leqq k \leqq l)$ be an auxiliary vector in z. We shall say that z_k is *immediate* if it is j-immediate for some $1 \leqq j \leqq q$. We shall say that z_k is *short* if it is j-short for some $1 \leqq j \leqq q$. Finally, we shall say that z_k is *long* if it is neither immediate nor short.

Let r, s, and t denote the numbers of immediate, short, and long vectors in z. Clearly,

$$n = r + s + t.$$

We shall say that a chain is *special* if, for each $1 \leqq u \leqq v$, it contains a vector z_k such that

$$D(z_k, u) \geqq N/2.$$

LEMMA 2.2–2. *For any special chain of length at most $L(p, q, N)$,*

$$s + t = O\left(\frac{H}{\log H}\right)$$

and

$$r + s \geqq v \log N + O\left(\frac{H \log\log H}{(\log H)^2}\right).$$

Proof. If z is a special chain, it must contain, for each $1 \leqq u \leqq v$, a vector z_k such that

$$D(z_k, u) \geqq N/2.$$

Applying the preceding lemma to this vector, we obtain

$$2^{r_u} \phi^{s_u + t_u} \geqq N/2,$$

or, by taking logarithms,

$$r_u + (s_u + t_u) \log \phi \geqq \log N - 1.$$

Summing this over $1 \leqq u \leqq v$ and using

$$\sum_{1 \leqq u \leqq v} r_u \leqq r,$$

$$\sum_{1 \leqq u \leqq v} s_u \leqq s,$$

$$\sum_{1 \leqq u \leqq v} t_u \leqq t,$$

we obtain

$$\begin{aligned} r + (s+t) \log \phi &\geqq v \log N - v \\ &= v \log N + O\left(\frac{(\log H)^2}{\log \log H}\right). \end{aligned}$$

Subtracting this from

$$\begin{aligned} r + s + t &= n \\ &\leqq L(p, q, N) \\ &\leqq v \log N + \frac{H}{\log H} \end{aligned}$$

yields

$$(s+t)(1 - \log \phi) \leqq \frac{H}{\log H} + O\left(\frac{(\log H)^2}{\log \log H}\right).$$

Since $1 - \log \phi > 0$, this proves the first assertion of the lemma.

Again applying the preceding lemma to z_k, we obtain

$$2^{r_u + s_u} \psi^{t_u} \geqq N/2,$$

or, by taking logarithms,

$$r_u + s_u + t_u \log \psi \geqq \log N - 1.$$

Summing over $1 \leqq u \leqq v$, we obtain

$$\begin{aligned} r + s + t \log \psi &\geqq v \log N - v \\ &= v \log N + O\left(\frac{(\log H)^2}{\log \log H}\right). \end{aligned}$$

Since

$$\begin{aligned} t &\leqq l \\ &= O(H) \end{aligned}$$

and

$$\begin{aligned} \log \psi &= h^{-1} \log h \\ &= O\left(\frac{\log \log H}{(\log H)^2}\right), \end{aligned}$$

this yields

$$r+s \geqq v \log N + O\left(\frac{H \log \log \mathrm{H}}{(\log H)^2}\right),$$

which proves the second assertion of the lemma. □

We shall say that a matrix is *special* if, for every $1 \leqq u \leqq v$, it contains a vector z_k such that

$$D(z_k, u) \geqq N/2.$$

LEMMA 2.2–3. *There are at least*

$$2^H U\left(\frac{(\log H)^4}{\log \log H}\right)$$

special matrices.

Proof. Let

$$K = \left\lfloor \frac{N+1}{4q} \right\rfloor.$$

For any $1 \leqq j \leqq q$, there are at least $(K+1)^q$ vectors f such that

$$D(f, j) \geqq N/2,$$

since if

$$\begin{aligned} &0 \leqq f_i \leqq K \quad \text{for } 1 \leqq i \leqq j, \\ &N-K \leqq f_j \leqq N, \\ &0 \leqq f_i \leqq K \quad \text{for } j < i \leqq q, \end{aligned}$$

then there are $K+1$ possible values for each component and

$$\begin{aligned} D(f, j) &\geqq N - qK \\ &= N - q\left\lfloor \frac{N+1}{4q} \right\rfloor \\ &\geqq N/2. \end{aligned}$$

It follows that there are at least

$$(K+1)^{qv}\binom{(N+1)^q}{p-v} \geqq \binom{(K+1)^q}{p}$$

special matrices. Estimating this binomial coefficient as before, we obtain

$$\begin{aligned}\binom{(K+1)^q}{p} &\geqq \left(\frac{(K+1)^q}{p}\right)^p\\ &= (K+1)^{pq}U(w \log w)\\ &\geqq \left(\frac{N+1}{4q}\right)^{pq} U(w \log w)\\ &= (N+1)^{pq}U(w^2 \log w)\\ &= 2^H U\left(\frac{(\log H)^4}{\log \log H}\right)\end{aligned}$$

special matrices. □

LEMMA 2.2–4. *For some value of t, there are at least*

$$2^H U\left(\frac{(\log H)^4}{\log \log H}\right)$$

special chains of length at most L(p, q, N).

Proof. Each special matrix is computed by a special chain of length at most $L(p, q, N)$. Each of these chains computes at most

$$\begin{aligned}\binom{l}{p} &\leqq l^p\\ &= U(p \log l)\\ &= U\left(\frac{(\log H)^3}{\log \log H}\right)\end{aligned}$$

matrices, so there are at least

$$2HU\left(\frac{(\log H)^4}{\log \log H}\right) \Big/ U\left(\frac{(\log H)^3}{\log \log H}\right) = 2^H U\left(\frac{(\log H)^4}{\log \log H}\right)$$

special chains of length at most $L(p, q, N)$. Each chain has one of at most

$$\begin{aligned}L(p, q, N) &= O(H)\\ &= U(\log H)\end{aligned}$$

possible values of t, so for some value of t there are at least

$$2^H U\left(\frac{(\log H)^4}{\log \log H}\right) \Big/ U(\log H) = 2^H U\left(\frac{(\log H)^4}{\log \log H}\right)$$

special chains. □

With each special chain z we shall associate an object, which will be called a *special code*, constructed as follows. Let $\mathcal{M}$ be the set defined above. Each immediate vector z_k in z is $2z_{b_k}$ for some b_k such that $1 \leqq b_k \leqq k-1$. Let $\mathcal{R}$ be a subset of $\{1, \cdots, l\}$ that contains r elements b_k, one corresponding to each immediate vector in z. Each short vector z_k in z is a j-vector for some $1 \leqq j \leqq q$ and is $z_{a_k} + z_{b_k}$ for some a_k and b_k such that $1 \leqq a_k < b_k \leqq k-1$, z_{a_k} and z_{b_k} are both j-vectors, and the number Δ_k of j-vectors intervening between z_{a_k} and z_{b_k} satisfies $0 \leqq \Delta_k \leqq h-1$. Let $\mathcal{S}$ be a subset of $\{0, \cdots, h-1\} \times \{1, \cdots, l\}$ that contains s ordered pairs (Δ_k, b_k), one corresponding to each short vector in z. Each long vector z_k in z is $z_{a_k}+z_{b_k}$ for some a_k and b_k such that $1 \leqq a_k < b_k \leqq k-1$. Let $\mathscr{T}$ be a subset of $\{1, \cdots, l\} \times \{1, \cdots, l\}$ that contains t ordered pairs (a_k, b_k), one corresponding to each long vector in z. The ordered quadruple $(\mathscr{M}, \mathscr{R}, \mathscr{S}, \mathscr{T})$ will be the special code associated with z.

LEMMA 2.2–5. *A special chain is uniquely determined by its special code.*

Proof. Let $(\mathscr{M}, \mathscr{R}, \mathscr{S}, \mathscr{T})$ be a special code. From $\mathscr{M}$, determine the set of basic vectors, arrange these in the standard order to form a chain. Remove from $\mathscr{R}$ all elements b, from $\mathscr{S}$ all pairs (Δ, b), and from $\mathscr{T}$ all pairs (a, b) for which b is less than or equal to the number of vectors currently in the chain. For each b removed from $\mathscr{R}$, compute $2z_b$. For each (Δ, b) removed from $\mathscr{S}$, determine j such that z_b is a j-vector, determine a such that z_a is a j-vector and exactly Δ j-vectors intervene between z_a and z_b, and compute z_a+z_b. For each (a, b) removed $\mathscr{T}$, compute z_a+z_b. Arrange the computed vectors in the standard order and append them to the end of the chain. Repeat this process until no more elements or pairs can be removed. Clearly, only the resulting chain can have the special code $(\mathscr{M}, \mathscr{R}, \mathscr{S}, \mathscr{T})$. □

LEMMA 2.2–6. *For any value of t, there are at most*

$$(H^2/t)^t U(t) U\left(\frac{H \log\log H}{\log H}\right)$$

special chains of length at most $L(p, q, N)$.

Proofs. For any m, r, s, and t, there are at most

$$\binom{q+1}{m}\binom{l}{r}\binom{hl}{s}\binom{l^2}{t}$$

special codes, since the four factors bound the number of ways of choosing $\mathscr{M}$, $\mathscr{R}$, $\mathscr{S}$, and $\mathscr{T}$, respectively. Since

$$q+1 = O\left(\frac{(\log H)^2}{\log\log H}\right),$$

then

$$\binom{q+1}{m} \leqq 2^{q+1} = U\left(\frac{(\log H)^2}{\log\log H}\right).$$

Since

$$l = O(H)$$

and

$$m+s+t = O\left(\frac{H}{\log H}\right),$$

$$\begin{aligned}\binom{l}{r} &= \binom{l}{l-r}\\ &= \binom{l}{m+s+t}\\ &\leqq \left(\frac{le}{m+s+t}\right)^{m+s+t} = U\left(\frac{H \log\log H}{\log H}\right).\end{aligned}$$

Since

$$hl = O(H(\log H)^2)$$

and

$$s = O\left(\frac{H}{\log H}\right),$$

$$\binom{hl}{s} \leqq (hle/s)^s = U\left(\frac{H \log \log H}{\log H}\right).$$

Since

$$l = O(H),$$

$$\binom{l^2}{t} \leqq (l^2 e/t)^t = (H^2/t)^t U(t).$$

There are

$$\begin{aligned} q+1 &= O\left(\frac{(\log H)^2}{\log \log H}\right) \\ &= U(\log \log H) \end{aligned}$$

possible values of m, and at most

$$\begin{aligned} L(p, q, N)^2 &= O(H^2) \\ &= U(\log H) \end{aligned}$$

possible combinations of values of r and s. Thus, for any value of t, there are at most

$$(H^2/t)^t U(t) U\left(\frac{H \log \log H}{\log H}\right)$$

special codes.

Each special chain is associated with some special code, and at most one special chain is associated with each special code. Thus the bound just derived applies to special chains as well as special codes. □

We can now complete the proof. By Lemmas 2.2–4 and –6, there is a value of t such that

$$(H^2/t)^t U(t) U\left(\frac{H \log \log H}{\log H}\right) \geqq 2^H U\left(\frac{(\log H)^4}{\log \log H}\right),$$

since these quantities bound the number of special chains of length at most $L(p, q, N)$. Taking logarithms, we obtain

$$2t \log H - t \log t + O(t) \geqq H + O\left(\frac{H \log \log H}{\log H}\right).$$

Ignoring the $t \log t$ term for the moment, this implies

$$2t \log H + O(t) \geqq H + O\left(\frac{H \log \log H}{\log H}\right)$$

or

$$(2t\log H)U\left(\frac{1}{\log H}\right) \geqq HU\left(\frac{\log\log H}{\log H}\right).$$

This yields

$$t \geqq \frac{H}{2\log H}U\left(\frac{\log\log H}{\log H}\right)$$

or, by taking logarithms,

$$\log t \geqq \log H + O(\log\log H).$$

Multiplication by t yields

$$t\log t \geqq t\log H + O(t\log\log H).$$

With this bound on the $t\log t$ term, the original inequality implies

$$t\log H + O(t\log\log H) \geqq H + O\left(\frac{H\log\log H}{\log H}\right)$$

or

$$(t\log H)U\left(\frac{\log\log H}{\log H}\right) \geqq HU\left(\frac{\log\log H}{\log H}\right).$$

Thus

$$\begin{aligned} t &\geqq \frac{H}{\log H}U\left(\frac{\log\log H}{\log H}\right). \\ &= \frac{H}{\log H} + O\left(\frac{H\log\log H}{(\log H)^2}\right) \end{aligned}$$

Since for special chains

$$r + s \geqq v\log N + O\left(\frac{H\log\log H}{(\log H)^2}\right),$$

we obtain

$$\begin{aligned} L(p, q, N) &\geqq r + s + t \\ &\geqq v\log N + \frac{H}{\log H} + O\left(\frac{H\log\log H}{(\log H)^2}\right) \\ &= v\log N + \frac{H}{\log H}U\left(\frac{\log\log H}{\log H}\right), \end{aligned}$$

which is the desired lower bound.

3. The upper bound. We shall prove

$$L(p, q, N) \leqq v\log N + \frac{H}{\log H}U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(w).$$

We shall begin with the preliminary upper bound

$$L(p, q, N) \leqq \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(v \log N) + O(w),$$

which we shall deduce from a theorem on graphs.

Let y be a p-by-q $(N+1)$-ary matrix. Let $C(y)$ denote the minimum possible number of edges in a directed graph in which

(1) there are p distinguished vertices called *inputs* and q other distinguished vertices called *outputs*;

(2) there is no directed path from an input to another input, from an output to another output, or from an output to an input; and

(3) for all $1 \leqq i \leqq p$ and $1 \leqq j \leqq q$, the number of directed paths from the ith input to the jth output is equal the jth component of the ith row of y.

Let $C(p, q, N)$ denote the maximum of $C(y)$ over all p-by-q $(N+1)$-ary matrices y. In [6] it was shown that

$$C(p, q, N) \leq \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(v \log N) + O(w).$$

Thus the preliminary upper bound will follow if we prove $L(y) \leqq C(y)$, which implies $L(p, q, N) \leqq C(p, q, N)$.

Consider a graph with at most $C(y)$ edges that meets the conditions enumerated above. We may assume that this graph has no cycles, since the deletion of all edges involved in cycles would not affect the number of paths from an input to an output unless that number were originally infinite. From this graph we can obtain another in which the degree (the number of edges directed from) each vertex is 0, 1 or 2, which has at most $C(y)$ vertices with degree 1 or 2, and which also meets the conditions enumerated above; this is done by replacing each vertex with degree $d \geqq 3$ by $d-1$ vertices with degree 2. We can then associate with each vertex a vector which, for $1 \leqq j \leqq q$, has as its jth component the number of paths from the vertex to the jth output. It is easy to verify that these vectors can be arranged to form a chain of length at most $C(y)$ that computes y. Thus $L(y) \leqq C(y)$, which completes the proof of the preliminary upper bound.

3.1. The easy case. Consider first the case

$$v \log N \leqq \frac{H \log\log H}{(\log H)^2}.$$

In this case

$$v \log N = \frac{H}{\log H} O\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right).$$

and the desired lower bound follows from

$$L(p, q, N) \leqq \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(v \log N) + O(w),$$

which has already been proved.

At this point we have proved the case $N = 1$, since in this case $v \log N = 0$. In particular,

$$L(c, d, 1) \leqq \frac{cd}{\log (cd)} U\left(\left(\frac{\log\log (cd)}{\log (cd)}\right)^{1/2}\right) + O(c) + O(d).$$

3.2. The hard case. Consider now the case

$$v \log N \geqq \frac{H \log \log H}{(\log H)^2},$$

which, as before, implies

$$w \leqq \frac{(\log H)^2}{\log \log H}.$$

Let y be a p-by-q $(N+1)$-ary matrix. For $1 \leqq i \leqq p$ and $1 \leqq j \leqq q$, let $e_{i,j}$ denote the jth component of the ith row of y, so that

$$y_i = \sum_{i \leqq j \leqq q} e_{i,j} x_j.$$

Let

$$s = \left\lceil \left(\frac{q \log (N+1)}{p} \right)^{1/2} \right\rceil,$$

$$t = \left\lceil \left(\frac{p \log (N+1)}{q} \right)^{1/2} \right\rceil.$$

Then

$$st \geqq \log (N+1),$$

so

$$2^{st} - 1 \geqq N.$$

On the other hand

$$\begin{aligned} st &\leqq \left\{ \left(\frac{q \log (N+1)}{p} \right)^{1/2} + 1 \right\} \left\{ \left(\frac{p \log (N+1)}{q} \right)^{1/2} + 1 \right\} \\ &= \log (N+1) + \left(\frac{q \log (N+1)}{p} \right)^{1/2} + \left(\frac{p \log (N+1)}{q} \right)^{1/2} + 1, \end{aligned}$$

so

$$\begin{aligned} pqst &\leqq H + (p+q) H^{1/2} + pq \\ &= H + O\left(\frac{H^{1/2} (\log H)^2}{\log \log H} \right). \end{aligned}$$

Similarly,

$$\begin{aligned} vst &= v \log N + O(H^{1/2}), \\ ps &= O(H^{1/2}), \\ qt &= O(H^{1/2}). \end{aligned}$$

We shall consider two cases, according to whether $p \geqq q$ or $p < q$.

If $p \geqq q$, we shall compute $y_1, \cdots, y_p$ from $x_1, \cdots, x_q$ in three steps as follows.

(1) For $1 \leqq j \leqq q$ and $1 \leqq b \leqq t$, compute

$$x'_{t(j-1)+b} = 2^{s(b-1)} x_j.$$

This defines x'_g for $1 \leqq g \leqq qt$. For $1 \leqq i \leqq p$ and $1 \leqq j \leqq q$, write $e_{i,j}$ as a t-digit number in base 2^s.

$$e_{i,j} = \sum_{1 \leqq b \leqq t} e'_{i,t(j-1)+b} 2^{s(b-1)},$$

where $0 \leqq e'_{i,t(j-1)+b} \leqq 2^s - 1$. This is possible since $0 \leqq e_{i,j} \leqq N \leqq 2^{st} - 1$; it defines $e'_{i,g}$ for $1 \leqq i \leqq p$ and $1 \leqq g \leqq qt$. Now write each $e'_{i,g}$ as an s-digit number in base 2:

$$e'_{i,g} = \sum_{1 \leqq a \leqq s} e''_{s(i-1)+a,g} 2^{a-1},$$

where $0 \leqq e''_{s(i-1)+a,g} \leqq 1$. This is possible since $0 \leqq e'_{i,g} \leqq 2^s - 1$; it defines $e''_{f,g}$ for $1 \leqq f \leqq ps$ and $1 \leqq g \leqq qt$.

(2) For $1 \leqq f \leqq ps$, compute

$$y'_f = \sum_{1 \leqq g \leqq qt} e''_{f,g} x'_g.$$

(3) For $1 \leqq i \leqq p$, compute

$$y_i = \sum_{1 \leqq a \leqq s} 2^{a-1} y'_{s(i-1)+a}.$$

It is easy to verify that this computes y_i correctly:

$$\begin{aligned}
y_i &= \sum_{1 \leqq a \leqq s} 2^{a-1} y'_{s(i-1)+a} \\
&= \sum_{i \leqq a \leqq s} 2^{a-1} \sum_{1 \leqq g \leqq qt} e''_{s(i-1)+a,g} x'_g \\
&= \sum_{1 \leqq a \leqq s} 2^{a-1} \sum_{1 \leqq b \leqq t} \sum_{1 \leqq j \leqq q} e''_{s(i-1)+a,t(j-1)+b} 2^{s(b-1)} x_j \\
&= \sum_{1 \leqq b \leqq t} \sum_{1 \leqq j \leqq q} e'_{i,t(j-1)+b} 2^{s(b-1)} x_j \\
&= \sum_{1 \leqq j \leqq q} e_{i,j} x_j.
\end{aligned}$$

Let us now count the number of additions required to perform these steps. Consider $x'_{t(j-1)+b}$. For $b = 1$, it is x_j; for $2 \leqq b \leqq t$, it can be computed from $x'_{t(j-1)+b-1}$ using s additions:

$$x'_{t(j-1)+b} = 2^s x'_{t(j-1)+b-1}.$$

Thus step (1) requires at most

$$\begin{aligned}
q(t-1)s &\leqq vst \\
&= v \log N + O(H^{1/2})
\end{aligned}$$

additions. Since $0 \leqq e''_{f,g} \leqq 1$ for $1 \leqq f \leqq ps$ and $1 \leqq g \leqq qt$, step (2) requires at most

$$\begin{aligned}
L(ps, qt, 1) &\leqq \frac{pqst}{\log(pqst)} U\left(\left(\frac{\log\log(pqst)}{\log(pqst)}\right)^{1/2}\right) + O(ps) + O(qt) \\
&= \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(H^{1/2})
\end{aligned}$$

additions, by taking $c = ps$ and $d = qt$ in the case $N = 1$. Finally, consider the sum

$$y''_{s(i-1)+r} = \sum_{r \leqq a \leqq s} 2^{a-r} y'_{s(i-1)+a}$$

For $r = s$, it is y'_{si}; for $1 \leqq r \leqq s-1$, it can be computed from $y''_{s(i-1)+r+1}$ using two additions:

$$y''_{s(i-1)+r} = y'_{s(i-1)+r} + 2y''_{s(i-1)+r+1}.$$

Since $y_i = y''_{s(i-1)+1}$, step (3) requires at most

$$\begin{aligned} 2p(s-1) &\leqq 2ps \\ &= O(H^{1/2}) \end{aligned}$$

additions. Summing these contributions completes the proof of the upper bound for $p \geqq q$.

If $p < q$, we shall compute $y_1, \cdots, y_p$ from $x_1, \cdots, x_q$ in three steps as follows.

(1) For $1 \leqq j \leqq q$ and $1 \leqq b \leqq t$, compute

$$x'_{t(j-1)+b} = 2^{b-1} x_j.$$

This defines x'_g for $1 \leqq g \leqq qt$. For $1 \leqq i \leqq p$ and $1 \leqq j \leqq q$, write $e_{i,j}$ as an s-digit number in base 2^t:

$$e_{i,j} = \sum_{1 \leqq a \leqq s} e'_{s(i-1)+a,j} 2^{t(a-1)},$$

where $0 \leqq e'_{s(i-1)+a,j} \leqq 2^t - 1$. This is possible since $0 \leqq e_{i,j} \leqq N \leqq 2^{st} - 1$; it defines $e'_{f,j}$ for $1 \leqq f \leqq ps$ and $1 \leqq j \leqq q$. Now write each $e'_{f,j}$ as a t-digit number in base 2:

$$e'_{f,j} = \sum_{1 \leqq b \leqq t} e''_{f,t(j-1)+b} 2^{b-1},$$

where $0 \leqq e''_{f,t(j-1)+b} \leqq 1$. This is possible since $0 \leqq e'_{f,j} \leqq 2^t - 1$; it defines $e''_{f,g}$ for $1 \leqq f \leqq ps$ and $1 \leqq g \leqq qt$.

(2) For $1 \leqq f \leqq ps$, compute

$$y'_f = \sum_{1 \leqq g \leqq qt} e''_{f,g} x'_g.$$

(3) For $1 \leqq i \leqq p$, compute

$$y_i = \sum_{1 \leqq a \leqq s} 2^{t(a-1)} y'_{s(i-1)+a}.$$

It is again easy to verify that this computes y_i correctly:

$$\begin{aligned} y_i &= \sum_{1 \leqq a \leqq s} 2^{t(a-1)} y'_{s(i-1)+a} \\ &= \sum_{1 \leqq a \leqq s} 2^{t(a-1)} \sum_{1 \leqq g \leqq qt} e''_{s(i-1)+a,g} x'_g \\ &= \sum_{1 \leqq a \leqq s} 2^{t(a-1)} \sum_{1 \leqq b \leqq t} \sum_{1 \leqq j \leqq q} e''_{s(i-1)+a,t(j-1)+b} 2^{b-1} x_j \\ &= \sum_{1 \leqq a \leqq s} 2^{t(a-1)} \sum_{1 \leqq j \leqq q} e'_{s(i-1)+a,j} x_j \\ &= \sum_{1 \leqq j \leqq q} e_{i,j} x_j. \end{aligned}$$

Let us again count the number of additions required to perform these steps. Consider $x'_{t(j-1)+b}$. For $b=1$, it is x_j, for $2 \leqq b \leqq t$, it can be computed from $x'_{t(j-1)+b-1}$ using one addition:

$$x'_{t(j-1)+b} = 2x'_{t(j-1)+b-1}.$$

Thus step (1) requires at most

$$\begin{aligned} q(t-1) &\leqq qt \\ &= O(H^{1/2}) \end{aligned}$$

additions. Since $0 \leqq e''_{f,g} \leqq 1$ for $1 \leqq f \leqq qs$ and $1 \leqq g \leqq qt$, step (2) requires at most

$$L(ps, qs, 1) \leqq \frac{H}{\log H} U\left(\left(\frac{\log\log H}{\log H}\right)^{1/2}\right) + O(H^{1/2})$$

additions, as in the case $p \geqq q$. Finally, consider the sum

$$y''_{s(i-1)+r} = \sum_{r \leqq a \leqq s} 2^{t(a-r)} y'_{s(i-1)+a}.$$

For $r=s$, it is y'_{si}; for $1 \leqq r \leqq s-1$, it can be computed from $y''_{s(i-1)+r+1}$ using $t+1$ additions:

$$y''_{s(i-1)+r} = y'_{s(i-1)+r} + 2^t y''_{s(i-1)+r+1}.$$

Since $y_i = y''_{s(i-1)+1}$, step (3) requires at most

$$\begin{aligned} p(s-1)(t+1) &\leqq vst + ps \\ &= v \log N + O(H^{1/2}) \end{aligned}$$

additions. Summing these contributions completes the proof of the upper bound.

REFERENCES

[1] R. E. BELLMAN, *Addition chains of vectors*, Amer. Math. Monthly, 70 (1963), p. 765.

[2] A. BRAUER, *On addition chains*, Bull. Amer. Math. Soc., 45 (1939), pp. 736–739.

[3] P. ERDÖS, *Remarks on number theory, III: On addition chains*, Acta Arith., 6 (1960), pp. 77–81.

[4] D. E. KNUTH, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[5] N. PIPPENGER, *On the evaluation of powers and related problems*, Proc. 17th Ann. IEEE Symp. on Found. of Computer Sci. (Houston, TX), 25-27 Oct. 1976, pp. 258-263.

[6] ———, *The minimum number of edges in graphs with prescribed paths*, Math. Systems Theory, to appear.

[7] A. SCHOLZ, *Jahresbericht der Deutschen Mathematiker-Vereinigung (II)*, 47 (1937), pp. 41–42.

[8] E. G. STRAUS, *Addition chains of vectors*, Amer. Math. Monthly, 71 (1964), pp. 806–808.

[9] A. C.-C. YAO, *On the evaluation of powers*, this Journal, 5 (1976), pp. 100–103.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

EFFICIENT SYNTHESIS AND IMPLEMENTATION OF LARGE DISCRETE FOURIER TRANSFORMATIONS*

SALVATORE D. MORGERA†

Abstract. A systematic technique is presented for synthesizing and efficiently performing large discrete Fourier transformations (DFT's) in the range from 60 to 5000 points. The technique is termed the mutual prime factor cyclic algorithm (MPFCA). The mutual prime factor portion of the algorithm is attributed originally to L. H. Thomas, with generalization supplied by I. J. Good; the cyclic aspect of the algorithm has recently been formalized by S. Winograd. Three methods are described for implementing the MPFCA; computational complexity (multiplications and additions) is estimated for each method and compared with the fast Fourier transform (FFT). For special purpose hardware, the MPFCA is *at least* twice as efficient as the FFT. A major result of importance is the realization that the three considerably different implementation methods presented lead to rather *similar* multiplication complexities for large size DFT's; furthermore, the resulting multiplication complexity is considerably higher than that achieved for small size DFT's. It is felt that further substantial improvements for large size DFT's built up using mutually prime factors will require more general theoretical results in addition to long, tedious hours spent with computer based formula manipulation systems.

Key words. discrete Fourier transformation (DFT), fast Fourier transformation (FFT), fast spectral analysis, finite impulse response (FIR), cyclic convolution, mutually prime factors, cyclic group, primitive root, cyclotomic polynomials

1. Introduction. Because of its widespread interdisciplinary use, untold numbers of diverse workers have concerned themselves with the efficient implementation of the DFT. The FFT algorithm was first described by Cooley and Tukey [1] in 1965. Their algorithm was quite general in that the number of points, N, could be composite and *not* necessarily a power of 2; however, only when N *is* a power of 2 (or 4) do this greatest saving and the commonly quoted proportionality of multiplications and additions to $N \log_2 N$ result.

Thomas describes an algorithm [2] which he used as far back as 1948 for performing calculations of Fourier series. His algorithm is different vis-à-vis the Cooley–Tukey algorithm for the following major reasons: (1) the factors of N *must* be mutually prime, and (2) no intervening phase shifts ("twiddle factors") are employed in multidimensional transformations. Good showed that if N is composite, with mutually prime factors (i.e., $N = N_1 \cdot N_2 \cdot \cdots \cdot N_L$, where g.c.d. $(N_k, N_l) = 1$; $k, l = 1, 2, \cdots, L$; $k \neq l$), a one-dimensional Fourier analysis of N points can be performed by doing L-dimensional Fourier analysis on an L-dimensional, $N_1 \times N_2 \times \cdots \times N_L$, array [3]. This generalization by Good to rectangular arrays in conjunction with the insight offered by Winograd [4], [5], and Agarwal and Cooley [6], [7] in the efficient transformation of the factors themselves leads directly to the MPFCA described here.

Winograd has published a rather general theorem [4] which states that the number of multiplications needed to compute the product of two polynomials, modulo a third polynomial, viz.,

$$R_M \cdot S_M (\text{mod } U_M) \tag{1}$$

is $2M - k$, where M is the polynomial degree, and k is the number of irreducible factors of U_M over the field, G, of interest, e.g., rationals, reals, or complex numbers. Recently,

* Received by the editors July 6, 1977, and in revised form April 13, 1979.

† Systems Engineering Laboratory, Raytheon Company, Submarine Signal Division, Portsmouth, Rhode Island. Now at the Department of Electrical Engineering, Concordia University, Montréal H3G 1M8, Québec, Canada.

Winograd has further shown in a rather terse communication [5] that the DFT can be a special case of (1), and, consequently, when $M = p$ is absolutely prime, it is possible to perform the "difficult part" of the DFT in $[2(p-1)-k]$ multiplications (over the rationals).[1] The result follows from the theorem of Winograd and the fact that the group of nonzero integers with the operation of multiplication modulo p is isomorphic to Z_{p-1} (the group of integers under addition modulo $p-1$). In practice, the key to achieving this efficiency is to exploit the fact that the transformation matrix associated with the difficult part of the DFT can be put into circulant form. For this work, it is important to realize that a DFT of size $M = p^r$, where p is an absolute prime, can also be performed efficiently; the reduced complexity follows again from the theorem of Winograd and the fact that the group of integers relatively prime to M with group multiplication modulo M is isomorphic to $Z_{(p-1)(M/p)}$, for $p \neq 2$. A very important part of this work involves the calculation of this cyclic group of integers in a manner which maintains the overall computational complexity (multiplication *and* additions) at a reasonable level. In general, more than the minimum number of multiplications (as dictated by the theorem of Winograd) is required in order to hold the number of additions down.

In general, when an N-point DFT is decomposed into mutually prime factors (i.e., $N = N_1 \cdot N_2 \cdot \cdots \cdot N_L$, where g.c.d. $(N_k, N_l) = 1$, $k, l = 1, 2, \cdots, L$; $k \neq l$), each of the factors N_l can take on one of the following forms: (1) $N_l = p$, an absolute prime; (2) $N_l = p^r$, a power $r \geqq 2$ of an (odd) absolute prime; and (3) $N_l = 2^t$. In this work, we synthesize DFT sizes which always have a factor of 4, i.e., $t = 2$; the remaining factors are then of the form p or p^r. We further restrict the *distinct* absolute primes which appear in the remaining factors to be either 3 or 5. The advantages of such a systematic approach are threefold: (1) a more than adequate selection of transformation sizes in the important range from 60 to 5000 points results, (2) the number of distinct cyclic transformations (3- and 5-point) that must be programmed is kept to a minimum, and (3) the overall computational complexity is kept to a minimum. The latter point certainly follows for multiplication complexity, one reason being that the factor of 4 leads to a 4-point transformation of multidimensional data which requires *no* multiplications.

We present three methods for implementing the N-point transformation synthesized as above; the methods differ in the manner in which the factors of form p^r are handled. Method I obtains the transforms of size p^r as $p^{(r-1)}r$ cyclic transformations of size p, with intervening twiddle factors. Method II obtains the transforms of size p^r as two transforms of size $p^{(r-1)}$ and a cyclic convolution of size $p^{(r-1)}(p-1)$. For smaller transform sizes ($N \leqq 900$), the multiplication complexity of Method II is significantly less than that achieved using Method I; however, for larger transform sizes, the addition complexity for Method II is about twice that of Method I. The reason for the larger addition complexity stems from the fact that the larger transform sizes implemented using Method II require long cyclic convolutions which have rather large addition complexities. We have used the best algorithms presently available [6], [7] for the long cyclic convolutions; it is possible that future improvements to the algorithms in the way of further minimizing the addition complexity at little (or no) expense in multiplication complexity will be made. Whether specific applications and processing environments await these future improvements is up to the user; e.g., multiplication is considerably more expensive than addition for a special-purpose machine or a computer which favors integer arithmetic. Method III does not compute the transform of size p^r per se;

[1] By "difficult part" we mean the $(p-1)$-dimensional transformation that results by excluding the first column and first row of the transformation matrix.

the complete N-point transformation is factored into a portion that is implemented with a number of small nested cyclic transformations and another portion that is implemented with both cyclic transformations and cyclic convolutions. Method III allows a reduced addition complexity vis-à-vis Method II but an addition complexity that is still greater than Method I, although in several cases the difference is slight. Definite promise for the partial nesting technique does reveal itself for certain transform sizes.

With regard to terminology, we use the wording cyclic transformation to mean a prime size transformation with point ordering permuted such that the transformation matrix of the difficult part is of circulant form. The wording cyclic convolution takes on the standard meaning found in the literature. We utilize the above distinction although recognizing that a prime size transformation may (also) be cast in the form of a convolution of one periodic sequence with another [8]. The recent book of McClellan and Rader [14] provides an excellent tutorial for understanding some of the manipulations carried out in this work, and the accompanying anthology contains many important references.

2. Synthesizing transforms of large size. Table 1 presents a list of the transform sizes, in the range from 60 to 5000 points, which are readily synthesized from the mutually prime base factors (4, 3, 5) of a 60-point transform. The transform sizes of Table 1 include selections close to the power of 2 sizes with which we are intimately familiar, as well as some potentially useful intermediate size selections.

TABLE 1
Transform sizes synthesized from 60-point transform factors.

Transform size	Mutually prime factors
60	$4 \cdot 3 \cdot 5$
180	$4 \cdot (3)^2 \cdot 5$
300	$4 \cdot 3 \cdot (5)^2$
540	$4 \cdot (3)^3 \cdot 5$
900	$4 \cdot (3)^2 \cdot (5)^2$
1,500	$4 \cdot 3 \cdot (5)^3$
1,620	$4 \cdot (3)^4 \cdot 5$
2,700	$4 \cdot (3)^3 \cdot (5)^2$
4,860	$4 \cdot (3)^5 \cdot 5$

All the transform sizes of Table 1 may be written as

$$N = 4 \cdot (p_1)^r \cdot (p_2)^s, \qquad r, s \geqq 1, \tag{2}$$

where $p_1 = 3$ and $p_2 = 5$ are absolutely prime. The factors $(p_1)^r$ and $(p_2)^s$ are mutually prime. Thus, using a version of the Chinese remainder theorem, the transform of size $N/4$ may be cast in the form of a two-dimensional transformation, wherein a $(p_1)^r$ point transform is performed in which each point is the result of a $(p_2)^s$ point transformation. These transforms require that transforms of size p_1 and p_2 be performed; the implementation details associated with these cyclic transformations are provided in Appendix B. The matrix algebraic manipulations in Appendix B are also easily extended to larger prime sizes. Method I for implementing the transform of size $N/4$ requires a number of small cyclic transforms of sizes p_1 and p_2, twiddle factors associated with transforms of sizes $(p_1)^r$ and $(p_2)^s$, and the procedure (control) for interconnecting the transforms. Method II also requires cyclic transformations of sizes p_1 and p_2, a number of cyclic convolutions, and the procedure for interconnecting transforms and convolutions. Method III utilizes a nesting technique, and although substantially different in principle from the first two methods, it does employ some ingredients from both Methods I and II.

Since the cyclic transformations of sizes $p_1 = 3$ and $p_2 = 5$ are common to all the methods described here, we review their complexity before proceeding further. As dictated by the theorem of Winograd [5], the *multiplication* complexity of prime size (p) transformations is given by $[2(p-1)-k]$ (real×complex, for complex data) multiplications, where k is the number of irreducible factors of the polynomial $U^{(p-1)}-1$ over the field of rationals. The factorization of $U^{(p-1)}-1$ over the field of rational numbers is known in terms of the cyclotomic polynomials. Useful rules for the generation of the cyclotomic polynomials are provided in Appendix A. From Appendix A we have,

$$\text{(3)}\qquad \begin{aligned} U^{(p_1-1)}-1 &= U^2-1=(U-1)(U+1),\\ U^{(p_2-1)}-1 &= U^4-1=(U-1)(U+1)(U^2+1). \end{aligned}$$

Therefore, the multiplication complexity is given by,

$$\text{(4)}\qquad \begin{aligned} [2(p_1-1)-k_1] &= 2(\text{real}\times\text{complex}) = 4(\text{real}) \triangleq m_1,\\ [2(p_2-1)-k_2] &= 5(\text{real}\times\text{complex}) = 10(\text{real}) \triangleq m_2 \end{aligned}$$

as verified by the details of Appendix B. Appendix B also provides the respective addition complexities as $a_1 = 6$ (complex) = 12(real), and $a_2 = 16$(complex) = 32(real) additions. In this work, we consider *both* addition and multiplication complexity for each of the methods because both complexities may be equally important for certain software environments and/or hardware implementations.

3. Method I: MPFCA with cyclic transformations only. From a programming standpoint, this method is perhaps the most straightforward and modularly redundant of the methods presented here. The modules are basically the 3- and 5-point cyclic transformations. We proceed simply by performing a transform of size p^r as $p^{(r-1)}r$ cyclic transforms of size p with $(r-1)(p-1)[p^{(r-1)}-1]$ intervening twiddle factors; e.g., a transform of size $(3)^2 = 9$ requires 6 cyclic transforms of size 3 and 4 complex phasors. Thus, the large transform of (2) is synthesized as $(p_1)^r$ transforms of size $(p_2)^s$, each requiring $(p_2)^{(s-1)}s$ cyclic transforms of size p_2, and $(s-1)(p_2-1)[(p_2)^{(s-1)}-1]$ twiddle factors, and $(p_2)^s$ transforms of size $(p_1)^r$, each requiring $(p_1)^{(r-1)}r$ cyclic transforms of size p_1, and $(r-1)(p_1-1)[(p_1)^{(r-1)}-1]$ twiddle factors. All this must be performed four times; although further additions are required by the 4-point transformation, *no* further multiplications are required. It is *very important* to note that no additional twiddle factors are needed to interconnect the transforms of the mutually prime factors: 4, $(p_1)^r$, and $(p_2)^s$.

The transform sizes of Table 1 synthesized in the above manner have multiplication and addition complexities, M_{NI} and A_{NI}, respectively, which may be related to the complexity of the cyclic transformations, viz.,

$$\text{(5a)}\qquad M_{NI} \leqq 4[(p_1)^r(p_2)^{(s-1)}sm_2+(p_2)^s(p_1)^{(r-1)}rm_1]+3T_{NI} \triangleq M^u_{NI}(\text{real}),$$

$$\text{(5b)}\qquad A_{NI} \leqq 4[(p_1)^r(p_2)^{(s-1)}sa_2+(p_2)^s(p_1)^{(r-1)}ra_1]+(p_1)^r(p_2)^s a_0+5T_{NI} \triangleq A^u_{NI}(\text{real}),$$

where

$$\text{(5c)}\qquad T_{NI} \triangleq 4\{(p_1)^r(s-1)(p_2-1)\cdot[(p_2)^{(s-1)}-1]+(p_2)^s(r-1)(p_1-1)[(p_1)^{(r-1)}-1]\}(\text{complex})$$

and a_0 is the number of (real) additions required by a 4-point transformation; i.e., $a_0 = 16$ (real) additions. We see that both M_{NI}^u and A_{NI}^u are *upper bounds*; these bounds may be lowered somewhat, but not necessarily concurrently, at the cost of storage for precomputed quantities. A tradeoff between speed and memory is generally biased in favor of the former because the cost of memory continues to decrease rapidly. Table 2 results from an evaluation of (5) for the transform sizes of Table 1. In evaluating (5), the cyclic transformation complexities of (4) and Appendix B have been employed along with the equivalence of three real multiplies and five real additions per complex multiply.

We see from Table 2 that when the transform sizes are synthesized using the MPFCA with cyclic transformations only, approximately one-half to one-third as many multiplications are required vis-à-vis $M_N \triangleq 2N \log_2 N$ (truncated), and about the same number of additions as $A_N \triangleq 3N \log_2 N$ (truncated). The FFT multiplication and addition complexities are used as comparative guidelines here although it is recognized that M_N and A_N are measures of these complexities only when N is a power of 2. Furthermore, we assert that the M_N and A_N used here are the complexities associated with *commonly available* software; optimized FFT software is not generally used, but it can reduce the (real) multiplication complexity to $3((N/2) \log N - (3N/2) + 2)$, and the (real) addition complexity to $2N \log N + 5 \cdot$ (number of multiplications). The reductions are achieved by not performing multiplications by unity and $\pm j$. The work of Singleton [13] provides an operation count for a specialized algorithm for computing a mixed radix FFT.

TABLE 2

Upper bound on multiplication and addition complexities for MPFCA Method I.

N	M_{NI}^u(real)	A_{NI}^u(real)	M_N(real)	A_N(real)	$\frac{C_{NI}^u}{C_N}$ $(c = 1/2)$
60	200	864	708	1,063	0.50
180	1,080	3,712	2,697	4,045	0.62
300	2,176	7,200	4,937	7,405	0.66
540	5,160	15,296	9,802	14,704	0.74
900	8,928	27,200	17,664	26,497	0.72
1,500	17,912	52,320	31,652	47,478	0.79
1,620	21,240	58,368	34,544	51,816	0.83
2,700	36,384	102,400	61,553	92,329	0.81
4,860	80,520	211,744	119,038	178,557	0.89

To develop a better appreciation for the results of Table 2, we define the upper bound on overall computational complexity as,

$$C_{NI}^u \triangleq M_{NI}^u + cA_{NI}^u, \tag{6}$$

where c denotes the "cost" of addition relative to multiplication. If we assume that additions "cost" half as much as multiplications (roughly halfway between what we expect for general purpose computers and specially constructed digital processing devices) and normalize the resulting quantity by $C_N \triangleq M_N + cA_N$, the last column of Table 2 results. We see that anywhere from 50 to 11 percent of the FFT computational complexity is saved by employing the MPFCA Method I for the transform sizes of Table 2. It is stressed that the figures shown in Table 2 represent an upper bound, with additional savings possible if certain nondata-dependent parameters are precomputed and stored. The algorithm must be worked out in detail for the transform sizes of interest in order to precisely determine these parameters. As an example, the algorithm has been worked out for (among others) the 60-point transformation [9]; optimizing speed at the sacrifice of storage resulted in 120 (real) multiplications, as opposed to the 200 (real) multiplications of Table 2. The number of additions did not increase, and it remained at the value of 864 (real) additions as shown in Table 2. The normalized computational complexity is now 0.44, for a relative savings of 56%.

It is clear from Table 2 that some transform sizes (r, s combinations) result in lower computational complexity vis-à-vis neighboring sizes; e.g., 900 and 2,700 are notably "better" sizes. We now investigate why this may be the case, and in so doing we provide some general characterization of the optimization of the computational complexity for *any* set of primes p_1, p_2. A necessary condition that a pair (r, s) give rise to minimum normalized computational complexity is that the pair be a solution to a linear combination of the two relations obtained by taking the partial derivative of the normalized computational complexity with respect to each variable independently and setting the result equal to zero, i.e., a linear combination of the two equations,

$$\text{(7a)} \qquad \frac{\partial C_N}{\partial r} C^u_{NI} = \frac{\partial C^u_{NI}}{\partial r} C_N,$$

$$\text{(7b)} \qquad \frac{\partial C_N}{\partial s} C^u_{NI} = \frac{\partial C^u_{NI}}{\partial s} C_N.$$

The solution to (7a) is the set of r-values for fixed s which minimize the normalized computational complexity; the solution to (7b) is the set of s-values for fixed r which minimize the normalized computational complexity (the second partial derivatives are positive).

Before forming a linear combination of (7a) and (7b), we solve (7a) using (6) and $C_N = (2+3c)N \log_2 N$. Taking the partial derivative with respect to r of (6) and simplifying results in

$$\text{(8a)} \qquad \frac{\partial C^u_{NI}}{\partial r} = \frac{\partial C_N}{\partial r}\left\{\left[\frac{1}{(2+3c)\log_2(4p_1^r p_2^s e)}\right][ar + bs + k + f_1(r) + f_2(s)]\right\},$$

where

$$\begin{aligned} a &= [(m_1 + ca_1) + (3+5c)(p_1 - 1)]/p_1, \\ b &= (m_2 + ca_2)/p_2, \\ \text{(8b)} \qquad k &= [(m_1 + ca_1) - (3+5c)(p_1 - 1)\ln p_1]/p_1 \ln p_1 + ca_0/4, \\ f_1(r) &= (3+5c)(p_1 - 1)[(p_1)^{(r-1)} - 1]/(p_1)^r \ln p_1, \\ f_2(s) &= (3+5c)(p_2 - 1)(s-1)[(p_2)^{(s-1)} - 1]/(p_2)^s \end{aligned}$$

and the variation of the FFT computationl complexity with respect to r is given by,

$$\text{(8c)} \qquad \frac{\partial C_N}{\partial r} = (2+3c)4(p_1)^r \ln p_1 (p_2)^s \log_2 [4(p_1)^r (p_2)^s e],$$

where e is the base of natural logarithms. The relations $(d/dx)a^u = a^u \ln a(du/dx)$ and $(d/dx)\log_a u = \log_a e/u(du/dx)$ have proved instrumental in obtaining (8). Inserting (8) into (7a) and after much algebraic manipulation, we obtain the optimization conditions for r (s fixed),

$$\text{(9a)} \qquad a_1 r^2 + b_1 r + c_1 = d_1(p_1)^r,$$

where

$$
\text{(9b)}\qquad
\begin{aligned}
a_1 &= (3+5c)(p_1-1)\ln p_1,\\
b_1 &= b_1(s) = (3+5c)(p_1-1)\ln [4(p_2)^s/p_1],\\
c_1 &= c_1(s) = -(3+5c)(p_1-1)[1+s\ln (p_1e)\ln (4p_2)/\ln p_1],\\
d_1 &= d_1(s) = s\{(m_2+ca_2)/p_2-\ln (4p_2)[(m_1+ca_1)+(3+5c)(p_1-1)]/p_1\ln p_1\}\\
&\quad +\{(3+5c)(p_2-1)(s-1)[(p_2)^{(s-1)}-1]/(p_2)^s\}-(3+5c)(p_1-1)/p_1+ca_0/4.
\end{aligned}
$$

The solution to (9) is the intersection of the quadratic and exponential curves; there is only one admissible solution, i.e., one for which $r \geqq 0$. Due to the "symmetric" form of C^u_{NI} and C_N, (7b) follows directly from (9), viz., (7b) becomes,

$$
\text{(9c)}\qquad a_2s^2+b_2s+c_2=d_2(p_2)^s,
$$

where a_2, b_2, c_2 and d_2 are equal to a_1, b_1, c_1 and d_1, respectively, with p_1 replaced by p_2 (and vice versa); m_1 and a_1 replaced by m_2 and a_2, respectively (and vice versa); and s replaced by r.

The linear combination we form is now obtained by subtracting (9a) and (9c),

$$
\text{(10a)}\qquad [a_1r^2-a_2s^2]+[b_1r-b_2s]+[c_1-c_2]=[d_1(p_1)^r-d_2(p_2)^s].
$$

The variation of (10a) has been extensively studied on the computer; the solution is given by,

$$
\text{(10b)}\qquad r=\left[\frac{p_2\ln p_2}{p_1\ln p_1}\right]^{1/2}s.
$$

However, the local region about this extremal point is asymmetrical, with slightly smaller values of r not causing great deviation from the minimum, but larger values of r causing a large jump in normalized computational complexity. The following range of r (in terms of s) is completely satisfactory for obtaining minimum computational complexity,

$$
\text{(10c)}\qquad s\leqq r\leqq \left[\frac{p_2\ln p_2}{p_1\ln p_1}\right]^{1/2}s.
$$

This result is very general and significant. It allows one to select combinations of p_1, p_2, r and s which give rise to minimum normalized computational complexity. Also, we see that the power of the smaller prime should always be lower bounded by the power of the larger prime. For the cases we have studied, we have $s \leqq r \leqq (1.56)s$, which is completely in accord with the observations derived from the data of Table 2.

4. Method II: MPFCA with cyclic transformations and convolutions. This approach to synthesizing the transform sizes of (2) uses a more complicated number theoretic technique for performing the constituent transforms of size p^r. We restrict p to be an *odd* prime. This restriction holds for the transform sizes in (2) since the even number 4 is a mutually prime factor. Rather than breaking down the transform of size p^r into a number of cyclic transformations of size p with intervening twiddle factors as in Method I, we compute the transform points in two steps: (1) those points that are not mutually prime to p^r and (2) those points that are mutually prime to p^r. We note that the latter set of points form a cyclic group [10]

DEFINITION. A group (subgroup) is called *cyclic* if there is some element in the group (subgroup) whose multiples constitute the whole group (subgroup). The particular element is called the *primitive root* α of the group (subgroup).

THEOREM. *The integers not exceeding, and mutually prime to, a fixed number, M, form a group under multiplication modulo M. The group is cyclic if M is equal to $2, 4, p^r$, or $2 \cdot p^r$, where p is an odd prime number.*

COROLLARY. *The group of integers* $\{1, 2, \cdots, p-1\}$, *where p is a prime number, form a cyclic group under multiplication modulo p.*

By way of example, the integers $\{1, 2, 3, 4\}$, with the operation of multiplication modulo $p=5$, form a cyclic group. The primitive root of the group is $\alpha = 2$. By the axioms of group theory, the unique inverse of the primitive root $\alpha^{-1} = \beta$, such that $\alpha \cdot \beta = 1$ (the unique identity element) is also a member of the group. Thus, multiples of β also constitute the whole group. For this example, $\beta = 3$.

The $M = p^r$ transform points are given by,

$$X_m = \sum_{k=0}^{M-1} x_k W_M^{mk}, \qquad m = 0, 1, 2, \cdots, M-1, \tag{11}$$

where $W_M \triangleq e^{-2\pi j/M}$. The M/p transform points which are *not* mutually prime to M are proportional to p, i.e.,

$$X_{lp} = \sum_{k=0}^{M-1} x_k W_M^{lpk}, \qquad l = 0, 1, 2, \cdots, \frac{M}{p} - 1. \tag{12}$$

Recognizing that $W_M^p = W_{M/p}$, (12) becomes

$$\begin{aligned} X_{lp} &= \sum_{k=0}^{M-1} x_k W_{M/p}^{lk} \\ &= \sum_{k=0}^{(M/p)-1} \left(\sum_{k'=0}^{p-1} x_{(k'(M/p)+k)} \right) W_{M/p}^{lk}, \qquad l = 0, 1, 2, \cdots, \frac{M}{p} - 1, \end{aligned} \tag{13}$$

which is precisely the form of an M/p point DFT. The remaining $(M/p)(p-1)$ points which are mutually prime to M are given by,

$$X_{lp+q} = \sum_{k=0}^{M-1} x_k W_M^{(lp+q)k}, \qquad l = 0, 1, 2, \cdots, \frac{M}{p} - 1, \qquad q = 1, 2, \cdots, p-1. \tag{14}$$

The DFT of (14) may be further decomposed into two transformations: one which operates on the data x_k for which $k = k'p$, and one for which $k = k'p + q'$, viz.,

$$\begin{aligned} X_{lp+q} &= \sum_{k'=0}^{(M/p)-1} x_{k'p} W_{M/p}^{(lp+q)k'} + \sum_{k'=0}^{(M/p)-1} \sum_{q'=1}^{p-1} x_{k'p+q'} W_M^{(lp+q)(k'p+q')}, \\ & l = 0, 1, 2, \cdots, \frac{M}{p} - 1, \qquad q = 1, 2, \cdots, p-1. \end{aligned} \tag{15}$$

The first term of (15) is another M/p point DFT. This DFT serves to provide those transform points for which $1 \leqq (lp+q) \leqq (M/p) - 1$. The transform is calculated only for these distinct points; then, these points are also utilized for larger values of $(lp+q)$ by proper modulo M/p application. The second term of (15) is somewhat more complicated and requires that we recall the definition of a cyclic group given earlier. The two sets of $(M/p)(p-1)$ integers—$(lp+q)$ and $(k'p+q')$, $l, k = 0, 1, 2, \cdots, (M/p)-1$; q, $q' = 1, 2, \cdots, p-1$—with the operation of multiplication modulo M, each form a cyclic group; therefore, any one of the integers in either set may be represented as a multiple of the primitive root α or its inverse $\alpha^{-1} = \beta$. We choose to represent $(lp+q) \equiv \alpha^n$ and

$(k'p+q')\equiv\beta^k$; therefore, the second term of (15) is equivalent to,

$$\begin{aligned}\sum_{k'=0}^{(M/p)-1}\sum_{q'=1}^{p-1} x_{k'p+q'}W_M^{(lp+q)(k'p+q')} &\equiv \sum_{k=0}^{(M/p)(p-1)-1} x_{\beta^k}W_M^{\alpha^n\beta^k}\\ &= \sum_{k=0}^{(M/p)(p-1)-1} x_{\alpha^{-k}}W_M^{\alpha^{(n-k)}},\end{aligned}\tag{16}$$

where it is understood that $(\alpha^{-1})^k=\alpha^{-k}$, i.e., α^{-k} is such that $\alpha^{-k}\cdot\alpha^k=1$ modulo M. The result of (16) is of the form of a cyclic convolution with impulse response

$$\{W_M^{\alpha^{(n-k)}}\}.$$

According to the theorem of Winograd [5], the convolution can be done in $[2(M/p)\cdot(p-1)-k]$ (real × complex) multiplications, where k is the number or irreducible factors (over the rationals) of the polynomial $U^{[(M/p)(p-1)]}-1$.

We see, then, that a $M=p^r$ point transformation is performed as *two* $M/p=p^{(r-1)}$ point transformations and a cyclic convolution of size $(M/p)(p-1)=p^{(r-1)}(p-1)$. The transformations cannot be performed cyclically unless $r=2$, i.e., unless M/p is absolutely prime. Thus, in order to achieve maximum efficiency, we must decompose the transformations further. In general, we will have $(r-1)$ stages $(r\geqq 2)$ of decomposition, leading to the computation of a $M=p^r$ point transformation as $2^{(r-1)}$ p-point cyclic transformations and $2^{(i-1)}$ cyclic convolutions of size $p^{(r-i)}(p-1)$, where $i=1,2,\cdots,r-1$. Any one of these cyclic convolutions requires $m_{i,r}\triangleq[2p^{(r-i)}(p-1)-k_{i,r}]$ (real × complex) multiplications, where $k_{i,r}$ is the number of irreducible factors (over the rationals) of the polynomial $U^{p^{(r-i)}(p-1)}-1$. The large transform of (2) is synthesized as a $(p_1)^r$ point transformation in which each point is a $(p_2)^s$ point transformation, both performed as above. The $(p_1)^r$ point transformation requires cyclic convolutions of lengths 6, 18, 54, and 162, for $r=2,3,4,5$; whereas, the $(p_2)^s$ point transformation requires cyclic convolutions of length 20 and 100 for $s=2,3$. As with Method I, everything must be carried out four times. Although no further multiplication complexity is added, some addition complexity is added by the 4-point transformation.

The transform sizes of Table 1 synthesized in the above manner have a multiplication complexity M_{NII}, which may be related to the computational complexity of the cyclic transformations and cyclic convolutions, viz.,

$$\begin{aligned}M_{NII}&\leqq 4\left\{(p_1)^r\left[2^{(s-1)}m_2+\sum_{i=1}^{s-1}2^{(i-1)}m_{i,s}\right]+(p_2)^s\left[2^{(r-1)}m_1+\sum_{i=1}^{r-1}2^{(i-1)}m_{i,r}\right]\right\}\\ &\triangleq M_{NII}^u(\text{real}).\end{aligned}\tag{17}$$

We assert that M_{NII}^u is an upper bound (albeit, tight) on the multiplication complexity for Method II. This upper bound may be lowered slightly (neglecting, for the moment, the important issue of what may happen to the number of additions) at the cost of storage of certain precomputed quantities and by simply reducing nonessential calculations, e.g., by calculating only the distinct points in the first term of (15). In computing (17) for the powers r and s of Table 1, the cyclotomic polynomial factorizations (cf. Appendix A) and subsequent *theoretical* convolution complexities of Table 3 are required. We emphasize the word theoretical because although the theorem of Winograd gives the minimum number of multiplications required to compute such convolutions, as the powers r and s grow, the number of additions becomes very large. Agarwal and Cooley [6], [7] have studied this difficult problem and have found that, in

TABLE 3
Cyclic convolution minimum multiplication complexity.

Convolution length		Polynomial factorization	Theoretical multiplication complexity (real × complex)
		$[k_{i,r}=k_{(r-i)}]$	$[m_{i,r}=m_{(r-i)}]$
Generated	6	$k_1=4$	$m_1=8$
by	18	$k_2=6$	$m_2=30$
Factor	54	$k_3=8$	$m_3=100$
$(p_1)^r$	162	$k_4=10$	$m_4=314$
		$[k_{i,s}=k_{(s-i)}]$	$[m_{i,s}=m_{(s-i)}]$
$(p_2)^s$	20	$k_1=6$	$m_1=34$
	100	$k_2=9$	$m_2=191$

general, *more* than the minimum number of multiplications is required to keep the number of additions to a reasonable level. For example, a $(p_1)^r=(3)^3=27$-point transform requires two 6-point cyclic convolutions and one 18-point cyclic convolution. It is relatively easy to find an algorithm to compute the short 6-point convolution using 8 (real × complex) multiplications, as in Table 3, and a reasonable number of additions. However, the best algorithm found by Agarwal and Cooley for the 18-point cyclic convolution requires 44 (real × complex) multiplications as opposed to the theoretical number of 30 (real × complex) multiplications from Table 3.

The technique of Agarwal and Cooley utilizes a multidimensional approach wherein a long one-dimensional cyclic convolution is written as a multidimensional convolution that is cyclic in all dimensions. Convolution along each dimension is of short length and is implemented by an efficient algorithm using a rectangular transform approach which reduces the number of multiplications *and* additions. The rectangular transform approach may be written in the following manner,

$$\mathbf{m}=A\mathbf{h}\otimes B\mathbf{x}, \qquad \mathbf{y}=C\mathbf{m}, \tag{18}$$

where A, B, and C^T are rectangular matrices of dimension $(M_c \times N_c)$, M_c being the number of points in the transform domain, and N_c being the number of input data points. The symbol $\otimes$ denotes point-by-point multiplication. It is obvious that $M_c \geqq N_c$; thus, the matrices A and B transform $\mathbf{h}$ and $\mathbf{x}$, respectively, to a higher dimension manifold. The necessary and sufficient (n.s.) condition on the A, B, and C matrices in (18) needed to produce the $(N_c \times 1)$-dimensional output data vector $\mathbf{y}$ as the cyclic convolution of the $(N_c \times 1)$-dimensional input data vector $\mathbf{x}$ and impulse response vector $\mathbf{h}$ is that the matrices should exhibit the property,

$$\sum_{k=0}^{M_c-1} C_{n,k}A_{k,p}B_{k,q}=\begin{cases}1, & p+q=n \bmod N_c,\\ 0, & \text{otherwise.}\end{cases} \tag{19}$$

For our application and many others, the impulse response vector $\mathbf{h}$ is fixed; therefore, $A\mathbf{h}$ may be precomputed and the associated operations not counted. Generally, the matrices B and C have integer elements and do not add to the number of multiplications required to compute (18); however, the integer elements may be rather large and as costly as multiplication. The objective of Agarwal and Cooley [6], [7] is to obtain algorithms with as few multiplications as possible in the point-by-point multiplication of $A\mathbf{h}$ and $B\mathbf{x}$ while still keeping the matrices B and C of simple form. An immediate decrease in the complexity of (18) is possible if we realize that the number of additions required in the calculation of $C\mathbf{m}$ is much larger than that required in the calculation of either $A\mathbf{h}$ or $B\mathbf{x}$. In order that advantage may be taken of this situation, it is possible to replace A by A', and C by C' where,

$$A'_{k,p} = C_{-p,k}, \qquad C'_{n,k} = A_{k,-n}. \tag{20}$$

Now, the vector $A'\mathbf{h}$, which involves the largest number of additions, is the precomputed quantity. It is easy to show that the matrices A', B, and C' satisfy the n.s. condition of (19). Efficient algorithms of the form of (18) and (20) have been derived for the short lengths $N_c = 2, 3, 4, 5, 6, 7, 8$, and 9. The "best" long convolution algorithms are then obtained by picking mutually prime factors from this set, ordering them properly, and using the Chinese remainder theorem to map the sequences into the multidimensional arrays.

If there are L mutually prime factors, i.e., if

$$N'_c = N_{c_1} \cdot N_{c_2} \cdots N_{c_L} \tag{21}$$

repeated application of (18) and (20) generalizes to the form,

$$\mathbf{y} = C'_1 C'_2 \cdots C'_L[(A'_L \cdots A'_2 A'_1)\mathbf{h} \otimes (B_L \cdots B_2 B_1)\mathbf{x}], \tag{22}$$

where $\mathbf{y}$ is (now) a $(N'_c \times 1)$-dimensional output data vector. The number of additions required to compute (22) depends on the ordering (21); the ordering which minimizes the number of additions is given by requiring that the quantity $T(N_{c_l}) \leqq T(N_{c_k})$ when $l < k$, where,

$$T(N_{c_l}) = \frac{m_l - N_{c_l}}{a_l}, \qquad l = 1, 2, \cdots, L. \tag{23}$$

The quantities m_l and a_l are the multiplication and addition complexities, respectively, for the cyclic convolution of short length N_{c_l}.

It should now be obvious that if we pick mutually prime factors, N_{c_l}, from the above set, then, of the required sizes of Table 3, we can compute only the length $N'_c = 6$, 18, and 20 cyclic convolutions. For the additionally required sizes, we choose to substitute longer convolutions using the observations that a cyclic convolution (or correlation) of length N_1 may be computed as part of a cyclic convolution of length N_2, where $N_2 \geqq 2N_1 - 1$ [8], [12]. An alternate approach is to imbed a *portion* of the cyclic convolution of size N_1 in a cyclic convolution of size $N_3 < N_2$, and compute the remaining points directly. We have examined this alternate approach for the specific sizes required here and found that the former technique of computing cyclic convolutions of size N_2 is more desirable from a computational complexity standpoint. For the required sizes of 54, 162, and 100, we compute longer convolutions of length 120, 360, and 210. Table 4 shows the selected convolution sizes, the short (ordered) transform factors, and the *practical* multiplication and addition complexities. The overall addition complexity, A_{NII}, is given by

$$\begin{aligned} A_{NII} &\leqq 4\left\{(p_1)^r\left[2^{(s-1)}a_2 + \sum_{i=1}^{s-1} 2^{(i-1)}a_{i,s}\right] + (p_2)^s\left[2^{(r-1)}a_1 + \sum_{i=1}^{r-1} 2^{(i-1)}a_{i,r}\right]\right\} + (p_1)^r(p_2)^s a_0 \\ &\triangleq A^u_{NII}(\text{real}). \end{aligned} \tag{24}$$

Table 5 results from an evaluation of (17) and (24) for the transform sizes of Table 1. The cyclic transformation complexities of (4) and Appendix B, and the cyclic convolution complexities of Table 4 have been employed in the evaluation of (17) and (24).

TABLE 4

Cyclic convolution multiplication and addition complexities using multidimensional rectangular transforms.

Convolution length	Ordered factors	Multiplication complexity (real × complex)	Addition complexity (complex)
		$[m_{i,r} = m_{(r-i)}]$	$[a_{i,r} = a_{(r-i)}]$
6	—	$m_1 = 8$	$a_1 = 34$
18	$2 \cdot 9$	$m_2 = 44$	$a_2 = 178$
(54) 120	$3 \cdot 8 \cdot 5$	$m_3 = 560$	$a_3 = 3{,}096$
(162) 360	$8 \cdot 9 \cdot 5$	$m_4 = 3{,}080$	$a_4 = 15{,}916$
		$[m_{i,s} = m_{(s-i)}]$	$[a_{i,s} = a_{(s-i)}]$
20	$4 \cdot 5$	$m_1 = 50$	$a_1 = 230$
(100) 210	$2 \cdot 3 \cdot 5 \cdot 7$	$m_2 = 1{,}520$	$a_2 = 7{,}566$

TABLE 5

Upper bound on multiplication and addition complexities for MPFCA Method II $C^u_{NII} = M^u_{NII} + cA^u_{NII}$.

N	M^u_{NII}(real)	A^u_{NII}(real)	M_N(real)	A_N(real)	$\frac{C^u_{NII}}{C_N}$ $(c = 1/2)$
60	200	864	708	1,063	0.50
180	840	3,712	2,697	4,045	0.57
300	1,840	8,688	4,937	7,405	0.71
540	3,800	16,416	9,802	14,704	0.70
900	6,720	31,664	17,664	26,497	0.72
1,500	41,760	206,160	31,652	47,478	2.61
1,620	32,600	166,848	34,544	51,816	1.92
2,700	26,560	121,392	61,553	92,329	0.81
4,860	193,800	993,664	119,038	178,557	3.32

We see from Table 5 that when the transform sizes are synthesized using the MPFCA with cyclic transformations and cyclic convolutions, approximately one-half to one-third as many multiplications are required vis-à-vis $M_N \triangleq 2N \log_2 N$ (truncated) for $N \leqq 900$ points. The addition complexity for this same range of transform points is approximately that of the FFT, i.e., approximately $A_N \triangleq 3N \log_2 N$ (truncated). Therefore, Method II has about the same normalized computational complexity for $N \leqq 900$ points. For larger transform sizes, the multiplication complexity is approximately equal to M_N, and the addition complexity can grow to as large as $8A_N$. The large multiplication complexity is caused by the necessity to perform cyclic convolutions of length $N_2 \geqq 2N_1 - 1$, with the extremely large addition complexity arising from the double length convolutions as well as the large addition complexities of Table 4 for the cyclic convolutions of long length. The exception to this is $N = 2{,}700$ which has a normalized computational complexity of 0.81, equal to that for Method I. This transform does not require double length cyclic convolution and uses only cyclic convolutions of lengths for which optimal algorithms have been found.

For large transform sizes ($N > 900$ points) we can see that a better approach is to synthesize the transform size by using an increased number of mutually prime factors, i.e., let $N = p_1^r p_2^s p_3^t$, for example, where the p_i are mutually prime. In this way the required cyclic convolution length will remain small and even double length cyclic convolutions, if required, will not exhibit extremely large addition complexities. For sizes $N \leqq 900$ points Method II is competitive with Method I and does have the distinct advantage of requiring the investigation and software implementation of relatively efficient cyclic convolutions, which are very important in the computation of auto- and cross-correlation functions, design of finite impulse response (FIR) and infinite impulse response (IIR) filters, and the solution of difference equations.

5. Method III: MPFCA with cyclic transformations and nested factors. Yet another method has been examined for implementing the transform sizes of Table 1. This technique attempts to capitalize on a property of a transformation whose size is the product of certain "special" numbers, some of which are absolutely prime. For the sizes 2, 3, 4, 5, 7, 8, 9, and 16, the associated transformation matrix $\tilde{W}_M$ may be decomposed in the following manner [11],

$$\tilde{W}_M = S_M D_M T_M, \tag{25}$$

where T_M is a $(J \times M)$-dimensional incidence (graph theoretic) matrix; (i.e., the element values are 0, +1, and −1 only), S_M is a $(M \times J)$-dimensional incidence matrix, and D_M is a $(J \times J)$-dimensional *diagonal* matrix. It is not difficult to decompose the transformation matrix in the form of (25) for $J \gg M$, e.g., $J = M^2$; what makes the above numbers special is that the decomposition of (25) exists for $J \sim M$. The implication here is that the number of multiplications required to perform a transformation of size $M = 2, 3, 4, 5, 7, 8, 9$, or 16 is $\sim M$. Examples for the sizes 3 and 5 relevant to our work here are given in Appendix B.

If the desired transform size $\hat{M}$ can be written as the product of L *mutually prime* factors $M_L \cdot M_{L-1} \cdots M_1$, then we have,

$$\begin{aligned} \mathbf{X}' &= \tilde{W}_{\hat{M}} \mathbf{x}'' \\ &= (\tilde{W}_{M_L} * \tilde{W}_{M_{L-1}} * \cdots * \tilde{W}_{M_1}) \mathbf{x}'', \end{aligned} \tag{26}$$

where the operation $*$ denotes the Kronecker (direct) matrix product. The vectors $\mathbf{x}''$ and $\mathbf{X}'$ are the $(\hat{M} \times 1)$-dimensional input data and transform vectors, respectively; the primes indicate that the point ordering is *not* "natural" and it is generally different for the two vectors. When each factor of $\hat{M}$ corresponds to one of the special numbers *such that the L factors are mutually prime*, then (25) may be substituted L times in (26) to obtain,

$$\mathbf{X}' = (S_{M_L} * S_{M_{L-1}} * \cdots * S_{M_1})(D_{M_L} * D_{M_{L-1}} * \cdots * D_{M_1})(T_{M_L} * T_{M_{L-1}} * \cdots * T_{M_1}) \mathbf{x}'', \tag{27}$$

where $M_l = 2, 3, 4, 5, 7, 8, 9$, or 16, with g.c.d. $(M_k, M_l) = 1$, $k, l = 1, 2, \cdots, L$ and $k \neq l$. It follows from (27) that the multiplication complexity of the transform of size $\hat{M}$ is given by the product of the individual multiplication complexities, m_{M_l}; i.e.,

$$M_{\hat{M}} = \prod_{l=1}^{L} m_{M_l} \tag{28}$$

independently of the ordering of the factors. We point out that a relationship similar to (28) also holds for long cyclic convolutions performed using the multidimensional rectangular transform approach of (22). If we wish to enjoy the computational savings offered by (28), we are limited to an upper transform size of $\hat{M} = 16 \cdot 7 \cdot 5 \cdot 9 = 5040$, because the factors must be mutually prime.[2] Of course, the number of sizes that can be synthesized is again limited to the product of mutually prime special numbers. We note that there is no theoretical limit to the transform size synthesized in the manner of (2).

[2] Strictly speaking, this is not true. The author has also found an efficient transform of size 11. This allows an upper transform size of $\hat{M} = 16 \cdot 11 \cdot 7 \cdot 5 \cdot 9 = 55{,}440$.

The addition complexity of the transform of size $\hat{M}$ is easily obtained via induction on two factors as,

$$(29)\qquad A_{\hat{M}} = \sum_{l=1}^{L}\left[\left(\prod_{j=1}^{L-l} M_j\right) a_{M_{L-l+1}}\left(\prod_{j=L-l+2}^{L} m_{M_j}\right)\right] \quad \text{(complex)},$$

where it is understood that $\prod_{j=a}^{b}(\cdot) = 1$ for $a > b$.

The result of (29) applies to complex data, where a_{M_l} and m_{M_l} denote the number of (complex) additions and (real × complex) multiplications required to compute a transform of size M_l. The ordering of the factors which minimizes the number of additions is identical to (23).

Methods I and II presented previously derive part of their advantage from the decomposition of (25) for the sizes 3 and 5. It would seem that if the transform sizes of interst were factored such that there were products comprising at least one factor that were mutually prime special numbers, then savings in excess of Methods I and II might derive. For the transform sizes of Table 1, it is natural to choose the (ordered) factors $\hat{M}_1 \triangleq 3 \cdot 4 \cdot 5 = 60$, and $\hat{M}_2 = 4 \cdot 5 \cdot 9 = 180$, and synthesize the larger transform size $N = \hat{M} \cdot (\prod_l M_l)$, as shown in Table 6.

TABLE 6
Transform sizes synthesized from the nested factors $\hat{M}_1, \hat{M}_2$.

Transform size	Factors
60	$\hat{M}_1 = 3 \cdot 4 \cdot 5$
180	$\hat{M}_2 = 4 \cdot 5 \cdot 9$
300	$\hat{M}_1 \cdot 5$
540	$\hat{M}_2 \cdot 3$
900	$\hat{M}_2 \cdot 5$
1,500	$\hat{M}_1 \cdot 5^2$
1,620	$\hat{M}_2 \cdot 3^2$
2,700	$\hat{M}_2 \cdot 3 \cdot 5$
4,860	$\hat{M}_2 \cdot 3^3$

We note from Table 6 that the factors other than $\hat{M}$ are special numbers. However, these other factors are obviously *not* mutually prime to $\hat{M}$. The other factor $(\prod_l M_l)$ is either a single special number ($N = 300$, 540, and 900), of the form $(p_1)^r$ or $(p_2)^s$ ($N = 1{,}500$, 1,620, and 4,860), or a composite of two special numbers ($N = 2{,}700$). Since the factors of the form $(p_1)^r$ and $(p_2)^s$ involve a cyclic convolution no larger than length 20, we choose to perform this transformation using Method II: the composite factor is handled just like a transformation of size $\hat{M}$. A number of complex twiddle factors $T_{NIII} = (\hat{M} - 1)(\prod_l M_l - 1)$ are required to interconnect the transform of size $\hat{M}$ with the remaining factor $(\prod_l M_l)$.

The transform sizes of Table 1 synthesized in the above manner have multiplication and addition complexities, M_{NIII} and A_{NIII}, respectively, which may be related to the associated complexities, $M_{\hat{M}}$ and $A_{\hat{M}}$ of the factor $\hat{M}$, and, $M_{(\prod_l M_l)}$ and $A_{(\prod_l M_l)}$, of the remaining factor $(\prod_l M_l)$, viz.

$$(30a)\qquad M_{NIII} \leqq (\hat{M}) M_{(\prod_l M_l)} + \left(\prod_l M_l\right) M_{\hat{M}} + 3T_{NIII} \triangleq M^u_{NIII}\text{(real)},$$

$$(30b)\qquad A_{NIII} \leqq (\hat{M}) A_{(\prod_l M_l)} + \left(\prod_l M_l\right) A_{\hat{M}} + 5T_{NIII} \triangleq A^u_{NIII}\text{(real)},$$

where

(30c) $$T_{NIII} = (\hat{M}-1)\Big(\prod_l M_l - 1\Big) \quad \text{(complex)}$$

and $\hat{M} = \hat{M}_1$ or $\hat{M}_2$. The addition and multiplication complexities for the transform of size $\hat{M}$ are given by (28) and (29); i.e.,

(31) $$\begin{aligned} \hat{M}_1 = 60: &\quad A_{\hat{M}_1} = 444(\text{complex}) = 888(\text{real}),\\ &\quad M_{\hat{M}_1} = 72(\text{real} \times \text{complex}) = 144(\text{real}),\\ \hat{M}_2 = 180: &\quad A_{\hat{M}_2} = 2{,}028(\text{complex}) = 4{,}056(\text{real}),\\ &\quad M_{\hat{M}_2} = 312(\text{real} \times \text{complex}) = 624(\text{real}). \end{aligned}$$

In calculating the value of (31), the number of multiplications (and associated additions) for 3-, 4-, 5-, and 9-point transformations given by Silverman [11] have been used because these numbers count multiplications by W^0, as required for the nested approach. Table 7 results from an evaluation of (30) using (31) for the transform sizes of Tables 1 and 6. Once again, we employ the equivalence of three real multiplies and five real additions per complex multiply.

We see from Table 7 that, again, at most, one-half to one-third as many multiplications are required vis-à-vis the FFT. In general, the number of multiplications required is less than the number for MPFCA Method II and about the same (with some substantial savings for $N = 1{,}620$ and $2{,}700$) as MPFCA Method I. We have reduced the number of additions considerably over MPFCA Method II because by partial nesting, the remaining factor $(\prod_l M_l)$ can be implemented with cyclic convolutions no larger than length 20. The addition complexity is still greater than that for MPFCA Method I although in several cases the difference is slight. Definite promise in the partial nesting technique reveals itself for $N = 2{,}700$, where *both factors*, $\hat{M}_2$ and $3 \cdot 5$, are partially nested and combined with the appropriate twiddle factors.

TABLE 7

Upper bound on multiplication and addition complexities for MPFCA Method III $C^u_{NIII} = M^u_{NIII} + cA^u_{NIII}$.

N	M^u_{NIII}(real)	A^u_{NIII}(real)	M_N(real)	A_N(real)	$\dfrac{C^u_{NIII}}{C_N}$ $(c = 1/2)$
60	144	888	708	1,063	0.47
180	624	4,056	2,697	4,045	0.56
300	2,028	7,540	4,937	7,405	0.67
540	3,666	16,118	9,802	14,704	0.68
900	7,068	29,620	17,664	26,497	0.70
1,500	15,048	60,720	31,652	47,478	0.81
1,620	14,232	57,360	34,544	51,816	0.70
2,700	23,358	102,530	61,553	92,329	0.69
4,860	55,290	229,982	119,038	178,557	0.81

6. Conclusions. A systematic approach to synthesizing large discrete Fourier transformations has been described that minimizes the number of small distinct cyclic transformations that must be programmed. In fact, the synthesis offered here of transformations in the large range from 60 to 5000 points requires only that cyclic transformations of sizes 3 and 5 be coded. Three methods have been presented and compared for implementing the large transformations: Method I interconnects cyclic transformations exclusively, Method II interconnects cyclic transformations along with a number of cyclic convolutions, and Method III is a partial nesting technique which employs ingredients from both Methods I and II.

Of primary importance is the realization that for large sizes the considerably different implementation methods lead to rather *similar* multiplication complexities. Furthermore, the multiplication complexity appears to behave asymptotically as $N \log_2 N$, in the same manner as that of the FFT. The implication is that the highly efficient multiplication complexity $(2N-k)$, promised by the theorem of Winograd and achievable for *small* transformation sizes, is *not* presently achievable for *large* transformation sizes. This is not to say that present methods do not offer improvement over the FFT complexity for large sizes—they do, as substantiated by the results presented here. Further substantial improvements for large sizes, however, will probably require theoretical breakthroughs in addition to long, tedious hours spent with computer-based formula manipulation systems.

To highlight some of the results, Table 8 presents (for complex input data) a comparison of the *average* computational complexity of the methods presented here relative to the FFT. By average computational complexity, we mean the linear combination of the (upper bound) number of multiplications and additions averaged over the transform sizes synthesized in the range of 60 to 5000 points; the parameter of Table 8 is the linear weighting, i.e., the "cost" of an addition relative to that of a multiplication.

TABLE 8

Upper bound on average computational complexity relative to FFT for selected addition/multiplication costs.

MPFCA implementation method	c		
	1/16	1/2	1
I	0.56	0.73	0.82
II	0.80	1.32	1.58
III	0.43	0.67	0.80

The left-most column of Table 8 ($c = 1/16$) might be indicative of a special purpose machine or one which favors integer arithmetic, whereas the right-most ($c = 1$) is suggestive of a general purpose computer. We see from Table 8 that all three methods favor special purpose hardware, with the saving of average computational complexity as high as 57%. Method II is actually more complex than the FFT for $c = 1/2$ and 1. This is due primarily to the large complexities of transforms larger than 900 points; for $N \leqq 900$, we have that the average computational complexity is 0.64 for $c = 1/2$ and 0.76 for $c = 1$. We stress again that the figures of Table 8 are upper bounds; the complexities may be reduced further at the expense of storage for certain precomputed nondata-dependent quantities. Special purpose hardware implementations optimized in this manner have yielded computational complexities as low as 16 to 20% of the FFT computational complexity for certain transform sizes [9]. In a general purpose computer, both Methods I and III seem to share the advantage, with the saving of average computational complexity as high as 20%.

All three methods are equally difficult to program. Method II does have the advantage that the detailed implementation of efficient cyclic convolutions of long length benefits many other application areas, e.g., the calculation of auto- and cross-correlation functions, design of IIR and FIR digital filters, solution of difference equations, MODEM design, and spatial beamforming. Finally, we mention that if these multidimensional cyclic convolutions are implemented in modulo arithmetic, there is no roundoff error introduced at any stage of the computation. Even if ordinary arithmetic is used, the rectangular transform approach described is likely to have less arithmetical roundoff noise than an FFT approach.

Appendix A. The purpose of this appendix is to introduce the factorization of a certain polynomial. The results presented here are important in that the number of factors is required for the calculation of the minimum multiplication complexity for certain discrete Fourier transformations and cyclic convolutions while the specific form of the factors is necessary for their detailed implementation.

The factorization of the polynomial U^M-1 over the field of rational numbers is given in terms of the cyclotomic polynomials $C_m(U)$, i.e.,

$$U^M-1=\prod_{\substack{m|M\\0<m\leqq M}} C_m(U), \tag{A.1a}$$

$$C_m(U)=\prod_{\substack{n\\ \text{g.c.d.}(n,\,m)=1\\ 1\leqq n<m}} (U-W_m^n). \tag{A.1b}$$

In (A.1) the notation $m|M$ is read as "all m that divide M", and $W_m \triangleq e^{-2\pi j/m}$. The theorem of Winograd uses the number of factors, k, where k is the order of the set $\{m|M;\, 0<m\leqq M\}$, to establish the minimum multiplication complexity. For example, if we wish to compute a cyclic convolution of length $M=6$, we have $k=4$ (1, 2, 3, and 6 divide 6), and $2M-k=8$(real×complex) multiplications are required for complex input data.

The cyclotomic polynomials $C_m(U)$ are easily generated from the following rules when p and q are absolutely prime $(C_1(U)=U-1)$,

$$C_p(U)=U^{(p-1)}+U^{(p-2)}+\cdots+U+1, \tag{A.2a}$$

$$C_{p^r}(U)=C_p(U^{p^{(r-1)}}), \tag{A.2b}$$

$$C_{2p}(U)=C_p(-U) \qquad (p \text{ odd}), \tag{A.2c}$$

$$C_{p\cdot q}(U)=\frac{C_q(U^p)}{C_q(U)}=\frac{C_p(U^q)}{C_p(U)}. \tag{A.2d}$$

It can be seen, therefore, that the cyclotomic polynomials have *simple* coefficients (0, +1, and −1).

Appendix B. The purpose of this appendix is to provide the implementation details associated with cyclic transformations of prime sizes $p_1=3$ and $p_2=5$.

B.1. Cyclic transformation of size $p_1=3$. The 3-point DFT is given by:

$$\begin{aligned}\begin{bmatrix}X_0\\X_1\\X_2\end{bmatrix}&=\begin{bmatrix}1&1&1\\1&W_3&W_3^2\\1&W_3^2&W_3^4\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\end{bmatrix}\\&=\begin{bmatrix}1&1&1\\1&W_3&W_3^2\\1&W_3^2&W_3\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\end{bmatrix},\end{aligned} \tag{B.1}$$

where we have used the fact that $W_3 = e^{-2\pi j/3} = W_3^4$. The difficult part of the DFT is given by:

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} W_3 & W_3^2 \\ W_3^2 & W_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \triangleq \tilde{W}_{2,3} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{B.2}$$

We see that the transformation matrix $\tilde{W}_{2,3}$ is already in circulant form. This matrix may be expanded in the following manner,

$$\tilde{W}_{2,3} = \left(\frac{W_3 + W_3^2}{2}\right)[1] + \left(\frac{W_3 - W_3^2}{2}\right)[1^{\pm}], \tag{B.3a}$$

where

$$[1] \triangleq \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{B.3b}$$

$$[1^{\pm}] \triangleq \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{B.3c}$$

Inserting (B.3) into (B.2) results in,

$$\begin{bmatrix} Y_1 \\ \\ Y_2 \end{bmatrix} = \begin{bmatrix} \left(\frac{W_3 + W_3^2}{2}\right)(x_1 + x_2) + \left(\frac{W_3 - W_3^2}{2}\right)(x_1 - x_2) \\ \left(\frac{W_3 + W_3^2}{2}\right)(x_1 + x_2) - \left(\frac{W_3 - W_3^2}{2}\right)(x_1 - x_2) \end{bmatrix}. \tag{B.4}$$

The complete transform is then given from (B.1) and (B.4) as

$$\begin{aligned} X_0 &= x_0 + (x_1 + x_2), \\ X_1 &= x_0 + Y_1, \\ X_2 &= x_0 + Y_2. \end{aligned} \tag{B.5}$$

The following real and pure imaginary quantities should be precomputed and stored for the calculation of (B.5).

$$\left(\frac{W_3 + W_3^2}{2}\right) = \left(\frac{W_3 + W_3^*}{2}\right) = \cos\left(\frac{2\pi}{3}\right) = -\frac{1}{2}, \tag{B.6a}$$

$$-j\left(\frac{W_3 - W_3^2}{2}\right) = -j\left(\frac{W_3 - W_3^*}{2}\right) = \sin\left(\frac{2\pi}{3}\right) = \frac{\sqrt{3}}{2}. \tag{B.6b}$$

Note that the 3-point transformation requires two (real × complex) multiplications and six (complex) additions. The decomposition (25) for the transformation matrix $\tilde{W}_3$ follows directly from the above, i.e.,

$$\tilde{W}_3 = S_3 D_3 T_3, \tag{B.7a}$$

where the incidence and diagonal matrices are equal to

(B.7b)
$$S_3=\begin{bmatrix}1&0&0\\0&1&-1\\0&1&1\end{bmatrix},$$

(B.7c)
$$T_3=\begin{bmatrix}1&1&1\\0&1&1\\0&1&-1\end{bmatrix},$$

(B.7d)
$$D_3=\begin{bmatrix}1&0&0\\0&\cos\left(\frac{2\pi}{3}\right)&0\\0&0&-j\sin\left(\frac{2\pi}{3}\right)\end{bmatrix}.$$

B.2. Cyclic transformation of size $p_2=5$. The 5-point DFT is given by:

(B.8)
$$\begin{bmatrix}X_0\\X_1\\X_2\\X_3\\X_4\end{bmatrix}=\begin{bmatrix}1&1&1&1&1\\1&W_5&W_5^2&W_5^3&W_5^4\\1&W_5^2&W_5^4&W_5&W_5^3\\1&W_5^3&W_5&W_5^4&W_5^2\\1&W_5^4&W_5^3&W_5^2&W_5\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\\x_3\\x_4\end{bmatrix}$$
$$=\begin{bmatrix}1&1&1&1&1\\1&W_5&W_5^2&(W_5^2)^*&W_5^*\\1&W_5^2&W_5^*&W_5&(W_5^2)^*\\1&(W_5^2)^*&W_5&W_5^*&W_5^2\\1&W_5^*&(W_5^2)^*&W_5^2&W_5\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\\x_3\\x_4\end{bmatrix}$$

where we have used the facts that $W_5^4=W_5^*$, and $W_5^3=(W_5^2)^*$.

The difficult part of the DFT is easily extracted, i.e.,

(B.9)
$$\begin{bmatrix}Y_1\\Y_2\\Y_3\\Y_4\end{bmatrix}=\begin{bmatrix}W_5&W_5^2&(W_5^2)^*&W_5^*\\W_5^2&W_5^*&W_5&(W_5^2)^*\\(W_5^2)^*&W_5&W_5^*&W_5^2\\W_5^*&(W_5^2)^*&W_5^2&W_5\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\end{bmatrix}\triangleq\tilde{W}_{4,5}\begin{bmatrix}x_1\\x_2\\x_3\\x_4\end{bmatrix}.$$

The permutation of the cyclic group of integers $\{1,2,3,4\}$ which puts the transformation matrix $\tilde{W}_{4,5}$ in circulant form may be determined from the primitive root $\alpha=2$. We have $\alpha^0=1$, $\alpha^1=2$, $\alpha^2=4$, $\alpha^3=3 \pmod 5$; thus, the desired permutation is $\{1,2,4,3\}$. Performing the permutation, we allow the transformation matrix $\tilde{W}_{4,5}$ of (B.9) to be partitioned in the following manner,

(B.10a)
$$\begin{bmatrix}Y_1\\Y_2\\Y_4\\Y_3\end{bmatrix}=\left[\begin{array}{c|c}\tilde{W}_{2,5}&\tilde{W}_{2,5}^*\\\hline\tilde{W}_{2,5}^*&\tilde{W}_{2,5}\end{array}\right]\begin{bmatrix}x_1\\x_2\\x_4\\x_3\end{bmatrix}$$

where

$$\tilde{W}_{2,5} \triangleq \begin{bmatrix} W_5 & W_5^2 \\ W_5^2 & W_5^* \end{bmatrix}. \tag{B.10b}$$

Partitioning the transform vector and data vector in the same fashion, we may write (B.10) as,

$$\begin{aligned} \mathbf{Y}_{1,2} &= \tilde{W}_{2,5}\mathbf{x}_{1,2} + \tilde{W}^*_{2,5}\,\mathbf{x}_{4,3}, \\ \mathbf{Y}_{4,3} &= \tilde{W}^*_{2,5}\,\mathbf{x}_{1,2} + \tilde{W}_{2,5}\mathbf{x}_{4,3}. \end{aligned} \tag{B.11}$$

In order to take advantage of the symmetry relationships inherent in (B.11), we must work separately with the real and imaginary components. The transformation matrix $\tilde{W}_{2,5}$ may be written as

$$\tilde{W}_{2,5} = \tilde{W}_{2,5\mathrm{R}} + j\tilde{W}_{2,5\mathrm{I}}, \tag{B.12a}$$

where

$$\begin{aligned} \tilde{W}_{2,5\mathrm{R}} = \mathrm{Re}\,\{\tilde{W}_{2,5}\} &= \begin{bmatrix} \cos\left(\frac{2\pi}{5}\right) & \cos\left(\frac{4\pi}{5}\right) \\ \cos\left(\frac{4\pi}{5}\right) & \cos\left(\frac{2\pi}{5}\right) \end{bmatrix}, \\ \tilde{W}_{2,5\mathrm{I}} = \mathrm{Im}\,\{\tilde{W}_{2,5}\} &= \begin{bmatrix} -\sin\left(\frac{2\pi}{5}\right) & -\sin\left(\frac{4\pi}{5}\right) \\ -\sin\left(\frac{4\pi}{5}\right) & \sin\left(\frac{2\pi}{5}\right) \end{bmatrix}. \end{aligned} \tag{B.12b}$$

Inserting (B.12) into (B.11), we have,

$$\mathbf{Y}_{1,2} = \tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2} + \mathbf{x}_{4,3}) + j\tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2} - \mathbf{x}_{4,3}), \tag{B.13a}$$

$$\mathbf{Y}_{4,3} = \tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2} + \mathbf{x}_{4,3}) - j\tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2} - \mathbf{x}_{4,3}). \tag{B.13b}$$

From (B.13), we see that there are only *two* linear transforms that must be computed: $\tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2} + \mathbf{x}_{4,3})$ and $\tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2} - \mathbf{x}_{4,3})$; we concentrate on these individually.

The circulant matrix $\tilde{W}_{2,5\mathrm{R}}$ may be expanded in the following manner,

$$\tilde{W}_{2,5\mathrm{R}} = \begin{bmatrix} w_{\mathrm{R}1} & w_{\mathrm{R}2} \\ w_{\mathrm{R}2} & w_{\mathrm{R}1} \end{bmatrix} = \left(\frac{w_{\mathrm{R}1} + w_{\mathrm{R}2}}{2}\right)[1] + \left(\frac{w_{\mathrm{R}1} - w_{\mathrm{R}2}}{2}\right)[1^{\pm}], \tag{B.14}$$

where the matrices [1] and $[1^{\pm}]$ are given by (B.3a) and (B.3b), respectively. Use of (B.14) permits the first term transformation of (B.13) to take the form

$$\tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2} + \mathbf{x}_{4,3}) = \begin{bmatrix} \left(\frac{w_{\mathrm{R}1} + w_{\mathrm{R}2}}{2}\right)(x_1 + x_2 + x_3 + x_4) + \left(\frac{w_{\mathrm{R}1} - w_{\mathrm{R}2}}{2}\right)(x_1 + x_4 - x_2 - x_3) \\ \left(\frac{w_{\mathrm{R}1} + w_{\mathrm{R}2}}{2}\right)(x_1 + x_2 + x_3 + x_4) - \left(\frac{w_{\mathrm{R}1} - w_{\mathrm{R}2}}{2}\right)(x_1 + x_4 - x_2 - x_3) \end{bmatrix}. \tag{B.15}$$

The symmetric (but not circulant) matrix $\tilde{W}_{2,5\mathrm{I}}$ is exapanded in the same fashion, but with an additional term, i.e.,

$$\tilde{W}_{2,5\mathrm{I}} = \begin{bmatrix} w_{\mathrm{I}1} & w_{\mathrm{I}2} \\ w_{\mathrm{I}2} & -w_{\mathrm{I}1} \end{bmatrix} = \left(\frac{w_{\mathrm{I}1} + w_{\mathrm{I}2}}{2}\right)[1] + \left(\frac{w_{\mathrm{I}1} - w_{\mathrm{I}2}}{2}\right)[1^{\pm}] - 2(w_{\mathrm{I}1})\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}. \tag{B.16}$$

Use of (B.16) permits the second term transformation of (B.13) to take the form

$$\text{(B.17)}\quad \tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2}-\mathbf{x}_{4,3})=\begin{bmatrix}\left(\frac{w_{\mathrm{I}1}+w_{\mathrm{I}2}}{2}\right)(x_1-x_4+x_2-x_3)+\left(\frac{w_{\mathrm{I}1}-w_{\mathrm{I}2}}{2}\right)(x_1-x_4-x_2+x_3)\\ \left(\frac{w_{\mathrm{I}1}+w_{\mathrm{I}2}}{2}\right)(x_1-x_4+x_2-x_3)-\left(\frac{w_{\mathrm{I}1}-w_{\mathrm{I}2}}{2}\right)(x_1-x_4-x_2+x_3)\\ \qquad\qquad\qquad\qquad\qquad\qquad -(2w_{\mathrm{I}1})(x_2-x_3)\end{bmatrix}.$$

The complete 5-point transform is then given from (B.8) and (B.13) simply by

$$\begin{aligned}&X_0=x_0+(x_1+x_2+x_3+x_4),\\ \text{(B.18)}\qquad &\mathbf{X}_{1,2}=x_0+\mathbf{Y}_{1,2}=x_0+\tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2}+\mathbf{x}_{4,3})+j\tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2}-\mathbf{x}_{4,3}),\\ &\mathbf{X}_{4,3}=x_0+\mathbf{Y}_{4,3}=x_0+\tilde{W}_{2,5\mathrm{R}}(\mathbf{x}_{1,2}+\mathbf{x}_{4,3})-j\tilde{W}_{2,5\mathrm{I}}(\mathbf{x}_{1,2}-\mathbf{x}_{4,3}).\end{aligned}$$

The following quantities should be precomputed and stored for the calculation of (B.18).

$$\text{(B.19a)}\quad \left(\frac{w_{\mathrm{R}1}+w_{\mathrm{R}2}}{2}\right)=\frac{1}{2}\left[\cos\left(\frac{2\pi}{5}\right)+\cos\left(\frac{4\pi}{5}\right)\right]=\frac{1}{2}\left[\cos\left(\frac{2\pi}{5}\right)+2\cos^2\left(\frac{2\pi}{5}\right)-1\right],$$

$$\text{(B.19b)}\quad \left(\frac{w_{\mathrm{R}1}-w_{\mathrm{R}2}}{2}\right)=\frac{1}{2}\left[\cos\left(\frac{2\pi}{5}\right)-\cos\left(\frac{4\pi}{5}\right)\right]=\frac{1}{2}\left[\cos\left(\frac{2\pi}{5}\right)-2\cos^2\left(\frac{2\pi}{5}\right)+1\right],$$

$$\text{(B.19c)}\quad \left(\frac{w_{\mathrm{I}1}+w_{\mathrm{I}2}}{2}\right)=\frac{-1}{2}\left[\sin\left(\frac{2\pi}{5}\right)+\sin\left(\frac{4\pi}{5}\right)\right]=-\frac{1}{2}\left[\sin\left(\frac{2\pi}{5}\right)+2\sin\left(\frac{2\pi}{5}\right)\cos\left(\frac{2\pi}{5}\right)\right],$$

$$\text{(B.19d)}\quad \left(\frac{w_{\mathrm{I}1}-w_{\mathrm{I}2}}{2}\right)=\frac{-1}{2}\left[\sin\left(\frac{2\pi}{5}\right)-\sin\left(\frac{4\pi}{5}\right)\right]=-\frac{1}{2}\left[\sin\left(\frac{2\pi}{5}\right)-2\sin\left(\frac{2\pi}{5}\right)\cos\left(\frac{2\pi}{5}\right)\right],$$

$$\text{(B.19e)}\qquad (2w_{\mathrm{I}1})=-2\sin\left(\frac{2\pi}{5}\right).$$

As before, it is easy to decompose the transformation matrix W_5 in the form of (B.7a) by including the additional "multiplication" by W^0. Note that the 5-point transformation requires five (real × complex) multiplications and 16 (complex) additions.

Acknowledgment. The author wishes to acknowledge many useful discussions with both Ralph W. Lamp of Raytheon Company and Dr. M. Vidyasagar of Condordia University, who have extraordinarily intuitive insight into the properties of multi-radix discrete Fourier transformations and functional optimization, respectively. Also, the comments of Dr. Eric Dubois of Université du Québec à Montréal are most gratefully acknowledged.

REFERENCES

[1] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput., 19 (1965), pp. 297–301.

[2] L. H. Thomas, *Using a computer to solve problems in physics*, Application of Digital Computers, Ginn, Boston, MA, 1963.

[3] I. J. Good, *The interaction algorithm and practical Fourier analysis*, J. Roy. Statist. Soc. Ser. B, 20 (1958), pp. 361–372; Addendum, 22 (1960), pp. 372–375.

[4] S. WINOGRAD, *Some bilinear forms whose multiplication complexity depends on the field of constants*, IBM T. J. Watson Research Center Rep. RC 5669, October 1975.

[5] ———, *On computing the discrete Fourier transform*, Proc. Nat. Acad. Sci. U.S.A. (73), 4 (1976), pp. 1005–1006.

[6] R. C. AGARWAL AND J. W. COOLEY, *New algorithms for digital convolution*, Conference on Acoustics, Speech, and Signal Processing Record, Hartford, CT, May 1977, pp. 360–362.

[7] ———, *New algorithms for digital convolution*, IBM Thomas J. Watson Research Center Rep. RC 6446, March 1977.

[8] C. M. RADER, *Discrete Fourier transformation when the number of data samples is prime*, Proc. IEEE, 56 (1968), pp. 1107–1108.

[9] S. D. MORGERA, *A mutual prime factor cyclic algorithm for discrete Fourier transformation*, Raytheon SSD Memorandum, SDM: 77/01, 5 April 1977.

[10] T. NAGELL, *Introduction to Number Theory*, John Wiley, New York, 1951.

[11] H. F. SILVERMAN, *An introduction to programming the Winograd Fourier transform algorithm (WFTA)*, IEEE Trans. Acoust., Speech, Signal Process, ASSP-25 (1977), pp. 152–165.

[12] C. M. RADER, Personal Communication, October 1978.

[13] R. C. SINGLETON, *An algorithm for computing the mixed radix fast Fourier transform*, IEEE Trans. Audio Electroacoust. AU-17 (1969), pp. 93–103.

[14] J. MCCLELLAN AND C. RADER, *Number Theory in Digital Signal Processing*, Prentice-Hall, New Jersey, 1978.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

PROBABILISTIC ALGORITHMS IN FINITE FIELDS*

MICHAEL O. RABIN†

Abstract. We present probabilistic algorithms for the problems of finding an irreducible polynomial of degree n over a finite field, finding roots of a polynomial, and factoring a polynomial into its irreducible factors over a finite field. All of these problems are of importance in algebraic coding theory, algebraic symbol manipulation, and number theory. These algorithms have a very transparent, easy to program structure. For finite fields of large characteristic p, so that exhaustive search through Z_p is not feasible, our algorithms are of lower order in the degrees of the polynomial and fields in question, than previously published algorithms.

Key words. Computations in finite fields, root-finding, factorization of polynomials, probabilistic algorithms

In this paper we utilize the method of probabilistic algorithms to solve some important computational problems pertaining to finite fields. The questions we deal with are the following. Given a prime p and an integer n, how do we actually perform the arithmetical operations of $E = GF(p^n)$. Given a polynomial $f(x)$ of degree m with coefficients in E, we wish to find a root $\alpha \in E$ of $f(x) = 0$, if such a root does exist. This is the *root-finding* problem. Finally, given a polynomial $f(x) \in E[x]$, we want to find the factorization $f = f_1 \cdot f_2 \cdot \cdots \cdot f_k$ of f into its irreducible factors $f_i(x) \in E[x]$. This is the *factorization* problem.

All of the above problems are of great significance in algebraic coding theory, see [2], in algebraic symbol manipulation, and in computational number theory.

Algorithms for the latter two problems are given in Berlekamp's book [2], and more completely in the important paper [3] which culminates his own work on the subject and also incorporates important ideas of Collins, Knuth, Welch, Zassenhaus, and others.

Berlekamp solves the root-finding problem for $f \in GF(p^n)$, $\deg(f) = m$, by reducing it to the factorization problem of another polynomial $F(x) \in Z_p[x]$ ($Z_p = GF(p)$, is the field of residues mod p), where $\deg(F) = mn$. The problem of factoring $F(x) \in Z_p[x]$ is solved by reducing it to finding the roots in Z_p of another polynomial $G(x) \in Z_p[x]$. Thus everything is reduced to root-finding in Z_p. For root-finding in a large Z_p, a case in which search is not feasible, Berlekamp proposes a probabilistic algorithm involving a random choice of $d \in Z_p$. The article [3] does not contain a proof for the validity of this algorithm.

Our starting point is to solve directly the problem of root-finding in $GF(p^n) = E$ for polynomials $f \in E[x]$, by a probabilistic algorithm which generalizes to arbitrary finite fields Berlekamp's algorithm for Z_p. The validity of the algorithm is based on Theorem 4 which has a surprisingly simple proof.

We now base factorization of a polynomial $f(x) \in Z_p[x]$ on root-finding for the same f. Namely, if $f(x)$ has irreducible factors of degree m, $h_i(x) \in Z_p[x]$, $1 \leqq i \leqq k$, then the product $D(x) = \Pi h_i(x)$ of these factors can be readily found by computations in $Z_p[x]$. The roots of $D(x)$ are in $GF(p^m)$ and the above root-finding algorithm allows us to directly find such a root $\alpha \in GF(p^m)$. The minimal polynomial $h(x) \in Z_p[x]$ of α, which is of degree m, can be found by one of two methods given in § 3. Now, α is also a root of some $h_i(x)$ of degree m, so that $h(x) = h_i(x)$, and we have found one irreducible factor of $f(x)$. An iteration of this process finds all the irreducible factors. The same

* Received by the editors December 15, 1978.

† Department of Mathematics, Hebrew University of Jerusalem, Jerusalem, Israel, and Massachusetts Institute of Technology, Cambridge, Massachusetts. This work was done while the author was visiting MIT Cambridge, Massachusetts in 1976.

algorithm works for factorization of polynomials $f(x) \in E[x]$, where E is any finite field, by use of roots of the polynomial $f(x)$ itself.

In terms of the number of Z_p-operations (additions and multiplications mod p, of numbers $0 \leqq a, b < p$) used, our algorithms are of complexity proportional to $\log p$. Thus they are feasible even for fields $GF(p^n)$ where p is so large that exhaustive search through Z_p is not possible.

Leaving out the factor $\log p$ and factors of order $\log n \cdot \log \log n$, the algorithms presented here have the following complexities. A root of $f(x) \in GF(p^n)$, $\deg f = m$, can be found in $O(n^2 m)$ Z_p-operations. A polynomial $f(x) \in Z_p[x]$, $\deg (f) = n$, can be factored in $O(n^3)$ operations.

If the arithmetical operations of the field $E = GF(p^n)$ are wired into circuitry so that an E-operation can be viewed as a unit, then the above root-finding algorithm uses $O(nm)$ operation. Under the same assumption for the fields $GF(p^i)$, $i \leqq n$, the factorization of $f(x)$ uses $O(n^2)$ operations.

The root-finding and factorization algorithms for the case of large p, given in [3] are of higher order in n. Root-finding for $f(x) \in GF(p^m)$, $\deg (f) = n$, uses $O((n \cdot m)^3 \cdot m)$ Z_p-operations. Factorization of $f \in Z_p[x]$, $\deg (f) = n$, uses $O(n^4)$ Z_p-operations.

If p is small so that it is practicable to find a solution in Z_p of $f(x) = 0$ by search, then a more careful comparison between the algorithms given here and the nonprobabilistic algorithms presented in [3] is necessary. The latter algorithm for factorization will run in time $O(n^3)$ but there is an $O(p)$ factor. Our algorithm will run in $O(n^3)$ (in the nonpreprocessed case) with a factor of $O(\log p)$. Thus for very small p, exact comparisons will depend on the numerical constants involved. However, the algorithms given here are sufficiently fast in all cases to justify their use even for small values of p.

The probabilistic nature of our algorithms does not detract from their practical applicability. The basic probabilistic step is a random choice of an element $\delta \in E$ which is then used in an attempt to split a polynomial $f(x)$ into two factors. We prove that for any fixed finite field E and any fixed $f(x)$, the probability of success by such a random choice is at least half. Thus the expected number of such steps leading to success is at most two. Furthermore, in an algorithm involving many such steps, the probability of a run of bad random choices leading to a significant deviation from the expected total number of steps is very small.

1. Arithmetic of $GF(p^n)$. Let p be a prime, n an integer and $q = p^n$. As customary, denote by $GF(q) = E$ the unique finite field of q elements. In particular $GF(p) = Z_p$ is the field of residues mod p. We want to actually compute with elements of E. For $Z_p = \langle\{0, 1, \cdots, p-1\}, +, \cdot\rangle$, the operations are simply addition and multiplication mod p. If

$$(1) \qquad g(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_0 \in Z_p[x],$$

is an *irreducible* polynomial of degree n, then

$$GF(p^n) \approx Z_p[x]/(g(x))$$

where (g) is the ideal generated by g. Given such a $g(x)$, E can be represented as the set of n-tuples of elements of Z_p. Let $\beta = (b_{n-1}, \cdots, b_0)$, $\gamma = (c_{n-1}, \cdots, c_0)$. Addition is componentwise. To multiply, form

$$d(x) = (b_{n-1}x^{n-1} + \cdots + b_0)(c_{n-1}x^{n-1} + \cdots + c_0)$$

and find the residue $\delta(x) = d_{n-1}x^{n-1} + \cdots + d_0$ of $d(x)$ when divided by $g(x)$. Then $\beta \cdot \gamma = (d_{n-1}, \cdots, d_0)$.

Thus we need a method for finding an irreducible polynomial (1). To *test* for irreducibility we use the following.

LEMMA 1. *Let $l_1, \cdots . l_k$ be all the prime divisors of n and denote $n/l_i = m_i$. A polynomial $g(x) \in Z_p[x]$ of degree n is irreducible in $Z_p[x]$ if and only if*

$$g(x)|(x^{p^n} - x), \tag{2}$$

$$(g(x), x^{p^{m_i}} - x) = 1, \qquad 1 \leqq i \leqq k, \tag{3}$$

where (a, b) denotes the greatest common divisor of a and b.

Proof. Assume that $g(x)$ is irreducible, then every root α of $g(x) = 0$ lies in $E = GF(p^n)$. Hence $\alpha^{p^n} - \alpha = 0$, and $(x - \alpha)|(x^{p^n} - x)$. Since $g(x)$ has no multiple roots, (2) follows.

Since $g(x)$ is irreducible of degree n, it has no roots in any field $GF(p^m)$, $m < n$. This direcly implies (3).

Assume conversely that (2) and (3) hold. From (2) it follows that all roots of $g(x) = 0$ are in $E = GF(p^n)$.

Assume that g has an irreducible factor $g_1(x)$ of degree $m < n$. The roots of $g_1(x)$ lie in $GF(p^m)$ which is generated over Z_p by any one of these roots. Hence $GF(p^m) \subseteq E$ and $m|n$. Consequently $m|m_i$ for one of the maximal divisors m_i of n, and all roots of $g_1(x)$ lie in $GF(p^{m_i})$. But then $(g(x), x^{p^{m_i}} - x)$ is divisible by $g_1(x)$ contradicting (3). Thus $g(x)$ must be irreducible.

In computing the number of operations required to test a given polynomial for primality we count, here and elsewhere in this article, in terms of arithmetical operations of Z_p. To obtain a bit-operations count, we should multiply our results by $B(p)$—the number of bit operations required to multiply or divide two numbers of $\log p$ bits. As is well known, $B(p)$ can be taken to be $O(\log p \cdot \log \log p)$.

In order to shorten subsequent formulas we introduce the following
Notation:

$$L(n) = \log.n \cdot \log \log n.$$

The computation of $(g(x), x^{p^n} - x)$ is executed by computing x^{p^n} modulo $g(x)$. As is well known, x^{p^n} can be calculated by at most $2 \cdot \log p^n$ multiplications mod $g(x)$. Since we compute mod $g(x)$ we never deal with polynomials of degree greater than $2n$.

It is shown in [4] that multiplying two n-degree polynomials with coefficients in any finite field can be done by $O(n \log n \log \log n) = O(n\, L(n))$ field operations. Consequently division and finding remainder can be done in $O(nL(n))$ operations, see [1, p. 288]. Thus the basic step of computing $r(x) \cdot s(x) \bmod g(x)$, where $\deg(r), \deg(s) \leqq n - 1$, uses $O(nL(n))$ operations. The computation of x^{p^n} uses $O(n^2 L(n) \log p)$ operations.

To test (3) we need $k \leqq \log n$ computations of the above type so that the total number of operations is $O(n^2 \log nL(n) \log p)$.

The search for an irreducible polynomial of degree n is based on the following result which is a weaker form, sufficient for our purposes, of Theorem 3.3.6 [2]. We give a proof not utilizing generating functions.

LEMMA 2. *Denote by $m(n)$ the number of different monic polynomials in $Z_p[x]$ of degree n which are irreducible. Then*

$$\frac{p^n - p^{n/2} \log n}{n} \leqq m(n) \leqq \frac{p^n}{n} \tag{4}$$

$$(5)\qquad \frac{1}{2n} \leqq \frac{m(n)}{p^n} \sim \frac{1}{n}.$$

Note that p^n is the number of all monic polynomials of degree n.

Proof. Let $g_1(x), \cdots, g_l(x)$, $l = m(n)$, be all the pairwise different irreducible monic polynomials of degree n. Any element $\alpha \in E = GF(p^n)$ which is of degree n over Z_p satisfies exactly one equation $g_i(x) = 0$ and each such equation has exactly n such roots. If $H \subseteq E$ is the set of elements of degree n over Z_p, then $c(H)/n = m(n)$.

An element $\alpha \in E$ is in H if it is *not* in any proper maximal subfield $GF(p^{m_i}) \subset E$, where m_i is a maximal divisor of n (see the notation in Lemma 1). The cardinality of such a subfield is at most $p^{n/2}$ and the number of these maximal subfields is smaller than $\log n$. Thus $p^n - p^{n/2} \log n \leqq c(H)$ from which (4) and (5) follow.

In [2] Berlekamp remarks that Theorem 3.3.6 means that a randomly chosen polynomial of degree n will be irreducible with probability nearly $1/n$, without suggesting to base an algorithm on this fact. In the general spirit of the present paper, we solve the problem of finding an irreducible polynomial by randomization.

The algorithm for finding an irreducible polynomial proceeds as follows. Choose a polynomial (1) *randomly* and test for irreducibility; continue until an irreducible polynomial of degree n is found. Lemma 2 ensures that the expected number of polynomials to be tried before an irreducible one is found is n. Thus the expected number of operations (in Z_p) for finding an irreducible polynomial of degree n is $O(n^3 \log nL(n) \cdot \log p)$.

The root-finding algorithm for $GF(q)$ assumes that the arithmetic of this field is given, so that the question of finding an irreducible polynomial actually does not arise. In the factorization of a polynomial of degree n we may need computations in fields $GF(p^{n_i})$, $1 \leqq i \leqq l$, such that $\sum n_i \leqq n$. The count of *all* operations, including the precomputation of the $g_{n_i}(x)$, will use the following.

LEMMA 3. *Let n_i, $1 \leqq i \leqq l$, satisfy $\sum n_i \leqq n$. The expected number of operations used for finding irreducible polynomials $h_i(x)$, $\deg(h_i) = n_i$, $1 \leqq i \leqq l$, is $O(n^3 \log nL(n) \log p)$.*

Proof.

$$\begin{aligned}\sum n_i^3 \log n_i L(n_i) \log p &\leqq n^2 \log nL(n) \log p \sum n_i \\ &\leqq n^3 \log nL(n) \log p.\end{aligned}$$

2. Root-finding in $GF(p^n)$. *Let $E = GF(q)$ be a fixed finite field, and $f(x) \in E[x]$* be a polynomial of degree m. We wish to find one (or all) of the roots $a \in E$ of $f(x) = 0$. We give a probabilistic algorithm for this problem, which is a generalization of the algorithm given in Berlekamp [3] for prime fields Z_p, to arbitrary finite fields E. Our proof for the validity of the general algorithm of course applies also to the special case of Z_p, which is given essentially without proof in [3].

Assume for the time being that $q = p^n$ is odd. We shall indicate later how to treat the important case $q = 2^n$.

Form the g.c.d.

$$f_1(x) = (f(x), x^{q-1} - 1).$$

If $f_1(x) = 1$ then $f(x)$ has no roots in E. In general

$$f_1(x) = (x - \alpha_1) \cdots (x - \alpha_k), \qquad k \leqq m,$$

where the α_i are all the pairwise different roots in E of $f(x) = 0$.

Now

(6) $$x^{q-1}-1=(x^d-1)(x^d+1), \qquad d=\frac{q-1}{2}.$$

The next natural step is to try $(f_1(x), x^d-1)$. If some of the α_i satisfy $\alpha_i^d-1=0$ while others satisfy $\alpha_j^d+1=0$, then this g.c.d. will be a true divisor of $f_1(x)$, and we will have further advanced towards the goal of finding a linear factor $x-\alpha$, i.e. a root, of $f(x)$. In general we are not guaranteed that the g.c.d. will be different from 1 or $f_1(x)$. However, this advantageous situation can be created by randomization.

Call $\alpha, \beta \in E$, $\alpha \neq 0$, $\beta \neq 0$, of *different type* if $\alpha^d \neq \beta^d$, where $d=(q-1)/2$.

THEOREM 4. *Let* $\alpha_1, \alpha_2 \in E$, $\alpha_1 \neq \alpha_2$.

(7) $$\frac{q-1}{2}=c(\{\delta \mid \delta \in E, \alpha_1+\delta \textit{ and } \alpha_2+\delta \textit{ are of different type}\}).$$

Proof. The elements $\alpha_1+\delta$ and $\alpha_2+\delta$ are of different type if and only if neither is zero and

$$\left(\frac{\alpha_1+\delta}{\alpha_2+\delta}\right)^d \neq 1; \quad \text{hence} \quad \left(\frac{\alpha_1+\delta}{\alpha_2+\delta}\right)^d=-1.$$

The equation $x^d=-1$ has exactly $d=(q-1)/2$ solutions in E. Consider the 1-1 mapping $\phi(\delta)=(\alpha_1+\delta)/(\alpha_2+\delta)$. As δ ranges over $E-\{-\alpha_2\}$, $\phi(\delta)$ ranges over $E-\{1\}$. Thus for exactly $(q-1)/2$ values of δ, $\phi(\delta)^d=-1$. This implies (7).

COROLLARY 5. *Consider for* $\delta \in E$ *the g.c.d.* $f_\delta(x)=(f_1(x), (x+\delta)^d-1)$. *We have*

(8) $$\tfrac{1}{2} \leqq Pr(\delta \mid 0<\deg f_\delta(x)<\deg f_1).$$

Proof. The common roots of $f_1(x)$ and $(x+\delta)^d-1$ are those $\alpha_i(f_1(\alpha_i)=0)$ for which $(\alpha_i+\delta)^d-1=0$. By Theorem 4, with probability $\frac{1}{2}$, $\alpha_1+\delta$ has this property while $\alpha_2+\delta$ does not, or vice-versa. This entails (8). Actually the probability is nearly $1-\frac{1}{2^k}$, where $\deg f_1=k$, but we cannot prove this.

Root-finding algorithm. Given $f(x)$ of degree m, compute $f_1(x)$. Choose $\delta \in E$ randomly and compute $f_\delta(x)$. If $0<\deg f_\delta<\deg f_1$ then let $f_2(x)=f_\delta(x)$ or $f_2(x)=f_1/f_\delta$, according as to whether $\deg f_\delta \leqq \frac{1}{2}\deg f_1$ or not. If $f_\delta=1$ or $f_\delta=f_1$ choose another δ and repeat the previous step. By Corollary 5, the expected number of choices of $\delta \in E$ until we find $f_2(x)$ is less than 2.

Since the degree is at least halved in each step, after at most $\log m$ steps we find a linear factor $x-\alpha_i$ of $f(x)$, i.e. a root.

The number of (field $-E$) arithmetical operations required for finding $f_1(x)$ and $f_2(x)$ is $O(n \cdot mL(m)\log p)$, where $E=GF(p^n)$. Since $\deg f_2 \leqq \frac{1}{2}m$, it follows that the number of operations for finding $f_3(x)$ is at most half the number of operations for finding f_2; and similarly for f_4 etc. Thus the total number of E-operations used for finding a root of $f(x)$ is still just $O(n \cdot mL(m)\log p)$.

In terms of operations in Z_p, each E-operation requires $O(nL(n))$ operations with residues modulo p. Thus the total (expected) number of Z_p-operations for root-finding is

(9) $$O(n^2 \cdot mL(m)L(n)\log p).$$

3. Factorization of polynomials. Let $f(x) \in Z_p[x]$ be a polynomial of degree n which we want to factor into its irreducible factors. We may assume that $f'(x)$ (the derivative) is not zero. For otherwise $f(x)=(g(x))^{p^k}$ where $g'(x) \neq 0$ and this g is readily found. For example, $x^{2p}+ax^p+b=(x^2+ax+b)^p$. By calculating $(f(x), f'(x))=h(x)$, and f/h, we have reduced the problem to factoring a polynomial with no repeated factors. Calculate

$$g_m(x)=(f(x), x^{p^m}-x), \qquad 1 \leqq m \leqq n.$$

Since $GF(p^m)$ consists exactly of all the elements of degrees i, $i|m$, over Z_p, we have that $g_m(x)$ is the product of all irreducible factors $h(x)|f(x)$ of degrees $i|m$.

Choose the $g_m \neq 1$ of lowest index m. If $\deg(g_m)=l$, then

$$g_m(x)=h_1(x)\cdots h_k(x), \qquad k \cdot m = l,$$

and each $h_i(x)$ is irreducible of degree m. All roots of $g_m(x)$ are in $GF(p^m)$. Find a root α of $g_m(x)=0$. This root is a root of a unique $h_i(x)$.

To find this $h_i(x)$ form the powers

$$1, \alpha, \cdots, \alpha^m. \tag{10}$$

These elements of $GF(p^m)$ are m-component vectors with coordinates in Z_p. Solve the system of linear equations

$$b_0+b_1\alpha+\cdots+b_{m-1}\alpha^{m-1}+\alpha^m=0, \tag{11}$$

where the b_i, $0 \leqq i \leqq m-1$, are the unknowns and the coordinates of the α^i are the coefficients. Now, $h_i(x)=x^m+b_{m-1}x^{m-1}+\cdots+b_0$.

Another way for computing $h_i(x)$ was suggested by M. Ben-Or. Note that $h_i(x)$ is irreducible of degree m. Since $\phi(\xi)=\xi^p$ is an automorphism of $GF(p^m)$ over the field Z_p, the conjugates of α are

$$\alpha_0=\alpha, \quad \alpha_1=\alpha^p, \cdots, \alpha_{m-1}=\alpha^{p^{m-1}}. \tag{12}$$

The polynomial $h_i(x)$ is now obtained by the calculation in $GF(p^m)$ of

$$h_i(x)=(x-\alpha_0)(x-\alpha_1)\cdots(x-\alpha_{m-1}). \tag{13}$$

Using either one of the above methods, one irreducible factor of $g_m(x)$ (and of $f(x)$) is found. Next we find a root β of $g_m(x)/h_i(x)$ and another factor $h_j(x)$ of $g_m(x)$, and so on.

Proceeding to factor the other $g_i(x)$, we choose $g_r(x) \neq 1$ with the lowest index $m<r$. If $m \nmid r$ then $g_r(x)$ is the product of irreducible factors of degree r. If $m|r$ then $g_m|g_r$, and g_r/g_m is the product of such factors. Factor $g_r(x)$ or g_r/g_m into its irreducible factors of degree r by one of the above methods.

In general, let $m_1<m_2<\cdots<m_t \leqq n$ be the indices for which $g_{m_i} \neq 1$. After $i-1$ steps we found $D_1(x), \cdots, D_{i-1}(x)$, where $D_j(x)$ is the product of all irreducible factors of degree m_j of $f(x)$, and each $D_j(x)$ is factored. (Note that $D_j(x) \equiv 1$ is possible despite $g_{m_j} \neq 1$. For example, $f(x)$ may have irreducible factors of degrees 2 and 3, but no irreducible factors of degree 6. In this case $D_2(x) \neq 1$, $D_3(x) \neq 1$, $D_6(x) \equiv 1$, and $g_6(x)=D_2(x)D_3(x)$.) Now,

$$D_i(x)=g_{m_i}(x) \Big/ \prod_{\substack{m_j|m_i \\ m_j<m_i}} D_j(x). \tag{14}$$

If $D_i(x) \neq 1$ and $m_i<\deg D_i(x)$, then factor it by the above method. If $m_i=\deg D_i(x)$ then $D_i(x)$ is already irreducible of degree m_i, and $f(x)$ has exactly one irreducible factor of this degree.

4. Counting operations. Let us now count the number of Z_p-operations required to factor a polynomial $f(x) \in Z_p[x]$ of degree n. The cost of getting rid of multiple factors of

$f(x)$ and of discovering the factors $D_i(x)$ defined in § 3 is majorized by the cost of factoring the $D_i(x)$, so that we confine ourselves to estimating the latter cost.

We have $f(x)=D_1(x)\cdots D_t(x)$, where $\deg D_i=d_i$.

Each $D_i(x)=h_{i1}(x)\cdot\cdots\cdot h_{ik_i}(x)$, where $\deg h_{ij}=m_i$, and h_{ij} is irreducible. The algorithm of § 3 seeks k_i roots $\beta_1,\cdots,\beta_{k_i}$ of $D_i(x)=0$, one for each factor $h_{ij}(x)$, so that $h_{ij}(\beta_j)=0$. Using the operation count (9) for root-finding, where $n=m_i$ because $\beta_i\in GF(p^{m_i})$, $1\leqq j\leqq k_i$), and $\deg D_i=d_i$, we get $O(m_i^2d_iL(d_i)L(m_i)\log p)$ for finding one root, say β_1. We then find $h_{i1}(x)$ by (11) or (13). Next we find a root of $D_i(x)/h_{i1}(x)$, so that we are sure that the root belongs to a $h_{ij}\neq h_{i1}$. Overestimating by not using the fact that $\deg(D_i/h_{i1})=d_i-m_i$ etc., we get $O(k_im_i^2d_iL(d_i)L(m_i)\log p)$ for total number of Z_p-operations to find the relevant roots of $D_i(x)$. Since $k_im_i=d_i$ and $m_i\leqq d_i$ we get

$$O(d_i^3L(d_i)^2\log p) \tag{15}$$

as a bound on these operations for $D_i(x)$. Since $n=\sum d_i$ we obtain by summation from (15), in the manner of deriving Lemma 3,

$$O(n^3L(n)^2\log p) \tag{16}$$

as a bound on cost of finding all the necessary roots of all the $D_i(x)$.

The first method for finding the $h_{ij}(x)$, once a root for each $h_{ij}(x)$ is given, employs $O(m_i^2L(m_i))$ Z_p-operations to calculate the sequence (10) of powers of the given root. The solution in Z_p of the system (11) of m linear equations in m unknowns uses $O(m_i^3)$ operations which majorizes the previous term. Summing over all the $h_{ij}(x)$ and overestimating we get $O(n^3)$ Z_p-operations for finding all the $h_{ij}(x)$, $1\leqq i\leqq t$, $1\leqq j\leqq k_i$.

We now estimate the operations used in Ben-Or's method for computing the $h_{ij}(x)$ from the roots. Using the notation of (12) and (13), so that the root is α and $\deg(h_i(x))=m_i$, we use $O(m_i\log p)$ $GF(p^{m_i})$-multiplications to perform the m_i raisings to exponent p. Counting Z_p-operations, we get

$$O(m_i^2L(m_i)\log p) \tag{17}$$

operations for computing the sequence (12).

The formation of the product (13) is a computation of the polynomial $h(x)$ from its given roots $\alpha_0,\alpha_1,\cdots,\alpha_{m-1}$. Using the result of [1, p. 299], and taking into account that in a finite field we require $O(m\,L(m))$ (instead of $O(m\log m)$ operations to multiply two polynomials of degree m, we get that

$$O((m_iL(m_i))^2\log m_i) \tag{18}$$

operations of Z_p are used to form each h_{ij}. Since $D_i(x)$ has k_i factor $h_{ij}(x)$, $1\leqq j\leqq k_i$, and $\deg D_i=m_ik_i$, we get from (17), (18) the upper estimate

$$O((nL(n))^2(\log n+\log p)) \tag{19}$$

for the Z_p-operations used in Ben-Or's method to find all the irreducible factors $h_{ij}(x)$, $1\leqq i\leqq t$, $1\leqq j\leqq k_i$, of $f(x)$, once a root of each factor was computed.

5. Summary of results and extensions. The root-finding method of § 2 is not applicable to polynomials $f(x)\in GF(2^n)[x]$. However, a small modification does work. Instead of $x^{q-1}-1$ we use the polynomial

$$Tr(x)=x+x^2+\cdots+x^{2^{n-1}}.$$

For $\alpha\in GF(2^n)=E$ we have $T(\alpha)^2=T(\alpha)$ so that every α is a root of $T(x)=0$ or of $T(x)=1$. Also $T(\alpha+\beta)=T(\alpha)+T(\beta)$.

THEOREM 6. *If $\alpha_1 \neq \alpha_2$, $\alpha_1, \alpha_2 \in E$, then*

$$2^{n-1} = c(\{\delta | T(\delta\alpha_1) \neq T(\delta\alpha_2)\}).$$

Proof. $T(\delta\alpha_1) \neq T(\delta\alpha_2)$ iff $T(\delta(\alpha_1+\alpha_2)) \neq 0$ i.e. $=1$. Now $\alpha_1+\alpha_2 \neq 0$ so that $\beta = \delta(\alpha_1+\alpha_2)$ runs with δ through all $\beta \in E$. In particular, for appropriate values of δ, all the 2^{n-1} roots of $T(x)=1$ are obtained. This proves the theorem.

Based on Theorem 6, we have a probabilistic root-finding algorithm for polynomials $f \in E[x]$ which is completely analogous than the algorithm in § 2.

The factorization algorithms for polynomials $f(x) \in Z_p[x]$ given in § 3 immediately generalizes to polynomials with coefficients in a general finite field $E = GF(q)$. The operations-count are the same, with E-operations replacing Z_p-operations.

We summarize our results as follows.

1. *Finding irreducible polynomials.* The expected number of steps for finding an irreducible polynomial $g(x) \in Z_p[x]$, of degree n is $O(n^3 \log n\, L(n) \log p)$. Any such polynomial enables us to compute in $GF(p^n)$.

2. *Root-finding.* The expected number of Z_p-operations used to find a root in $E = GF(p^n)$ of a polynomial $f(x) \in E[x]$ of degree m is $O(n^2 mL(m)L(n) \log p)$.

If the arithmetic of $GF(p^n)$ is directly wired into circuitry so that an O-arithmetical operation is counted as one operation, then the number of operations for root-finding is $O(n \cdot mL(m) \log p)$.

3. *Factorization into irreducible factors.* The total number of Z_p-operations for factoring a polynomial $f \in Z_p[x]$ of degree n is

$$O(n^3 \log nL(n) \log p) + O(n^3 L(n)^2 \log p) + O(n^3).$$

Here are included the computations of the necessary irreducible polynomials $g_i(x)$ needed for the arithmetics of the relevant fields $GF(p^m)$. The last term represents the operations used to solve linear equations under the first method.

If we assume that the arithmetics of all fields $GF(p^m)$, $m \leqq n$, are performed by wired circuitry then it is preferable to use the second method for computing the factors from the roots, based on (12) and (13). From (16) and (19) it follows, since each $GF(p^m)$ operation is counted as one operation, that the number of operations used for factoring a polynomial of degree n into irreducible factors is

$$O(n^2 L(n) \log p) + O(nL(n)(\log n + \log p)).$$

The first term majorizes the second term, but we display the latter as well since it reflects the structure of the algorithm.

REFERENCES

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

[2] E. R. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[3] ———, *Factoring polynomials over large finite fields*, Math. Comput., 24 (1970), pp. 713–735.

[4] A. SCHONHAGE, *Schnelle Multiplikation von Polynomen uber Körpern der Charakteristic 2*, Acta Informatica, 7 (1977), pp. 395–398.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0006 $01.00/0

A THEORETICAL ANALYSIS OF VARIOUS HEURISTICS FOR THE GRAPH ISOMORPHISM PROBLEM*

D. G. CORNEIL† AND D. G. KIRKPATRICK‡

Abstract. The graph isomorphism problem has received considerable attention due to the many practical applications of the problem and its unresolved complexity status. To deal with practical instances of the problem, a great deal of effort has gone into the development of seemingly quite effective heuristic algorithms Typically, these algorithms exploit various vertex properties which are invariant under isomorphism. Empirically, these heuristics have been analyzed extensively; however, very little theoretical analysis has been done on their intrinsic value.

In this paper we show that most commonly used vertex invariants are theoretically ineffective in the sense that any pair of graphs may be uniquely represented by a pair of graphs where the vertex invariant fails to give any information whatsoever about isomorphism or nonisomorphism. As a by-product of these results, new restricted families of graphs are shown to be isomorphism complete (i.e., the isomorphism problem on these graphs is polynomial-time equivalent to the general isomorphism problem).

Key words. graph isomorphism, heuristic algorithms, isomorphism complete problems, regular high girth graphs, automorphism partition

1. Introduction. Given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, a one-to-one mapping σ of V_1 onto V_2 is called an isomorphism iff $(x, y) \in E_1 \Leftrightarrow (\sigma x, \sigma y) \in E_2 \forall x, y \in V_1$. If an isomorphism exists between two graphs then the graphs are *isomorphic* (denoted $G_1 \cong G_2$) and the problem of determining whether two given graphs are isomorphic is termed the *graph isomorphism problem.*

In the past decade the graph isomorphism problem has received a great deal of attention in both the practical and theoretical computing literature. The development of computer algorithms for the graph isomorphism problem has been stimulated by such diverse applications as chemical identification [43], scene analysis [2] and construction and enumeration of combinatorial configurations [18]. Although these algorithms do not guarantee a solution in a reasonable amount of time, they seem to work well in many practical situations. For surveys and annotated bibliographies of these algorithms and other recent results on graph isomorphism see [39], [16], [11].

The theoretical attention stems from the persistent difficulty in characterizing the computational complexity of the isomorphism problem. Recent developments have shown that many combinatorial problems for which there are no known polynomial time algorithms are either *NP*-complete or *NP*-hard[1] [13], [1], [27]. In order to discuss the complexity of the graph isomorphism problem, it is convenient to formulate it as a language acceptance problem; namely, the given graphs G_1 and G_2 are encoded as a string (over some alphabet) which is to be accepted if and only if the two encoded graphs are isomorphic. This formulation of the graph isomorphism problem, while certainly in *NP*, has not been shown to be in *P* or *NP*-complete. Since the problem is not known to be *NP*-complete, it is of interest to ask whether the problem belongs to co-*NP* (i.e., does the complement problem of accepting a string if and only if the two graphs are

* Received by the editors May 5, 1978 and in revised form April 2, 1979.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7. This research was conducted while the author was on leave at the University of British Columbia.

‡ Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada V6T 1W5. This research was conducted while the author was at Simon Fraser University.

[1] We use the term *NP-hard* to denote those problems for which the existence of a polynomial algorithm would imply that $P = NP$.

nonisomorphic belong to *NP*?). As pointed out by Cook and Reckhow [14] and Pratt [38] an affirmative answer to this for any problem which is not known to be *NP*-complete would lend support to the conjecture that the problem is, in fact, not *NP*-complete.

Since it is possible that the graph isomorphism problem is neither in *P* nor *NP*-complete it is of interest to demonstrate polynomial-time equivalent problems. Such problems will be called *isomorphism complete* (see § 4 for a survey of known isomorphism complete problems). If any isomorphism complete problem were shown to be in *P* or to be *NP*-complete then all other isomorphism complete problems also would be in *P* or *NP*-complete. In particular, it is worthwhile to examine isomorphism problems on restricted families of graphs and to determine whether these problems are isomorphism complete. If the isomorphism problem on a restricted family of graphs is isomorphism complete, then this family of graphs will be referred to as an *isomorphism complete* family. This approach of finding restricted yet equivalent problems parallels similar efforts in the study of *NP*-complete problems [21]. Ideally, this approach would yield a restricted family of graphs for which the complexity of the isomorphism problem can be analyzed more easily. Any attempt at establishing lower bounds on the complexity of the general isomorphism problem could then concentrate on such a family. Furthermore, it is conceivable, that a polynomial isomorphism algorithm which utilizes properties of the restricted family of graphs could be developed. This approach might also help identify restricted families of graphs which are not isomorphism complete and thus hold more promise of admitting a polynomial time isomorphism test.

Most isomorphism complete restricted families of graphs do not seem to lend any insight into the difficulty of the graph isomorphism problem in the sense that there is little reason to assume that current practical algorithms would work any better or worse on these restricted graphs than on general graphs. In § 8 we will introduce new families of isomorphism complete restricted graphs for which current practical algorithms can be shown to fail.

In an attempt to solve practical instances of seemingly very difficult problems (such as isomorphism, *NP*-complete, or *NP*-hard problems), heuristic algorithms have been developed. In many cases, these heuristic algorithms work very well; in fact as pointed out by Karp [27], "one of the great mysteries in the field of combinatorial algorithms is the baffling success of many heuristic algorithms". This mystery arises in part from the current inability to analyze accurately many of these heuristic algorithms. We can identify two main types of heuristic algorithms, namely those which work in polynomial time but only provide an approximation to the correct answer and those which do solve the problem but do not guarantee a polynomial time bound. In the first case some valuable theoretical analyses have been done. For some problems (e.g., the bin packing problem), polynomial algorithms which guarantee a good approximation do exist; however, in other cases (e.g., the traveling salesman problem and graph coloring), problems of attaining certain good approximations are known to be *NP*-complete (see [20] and [28] for surveys of this work). Unfortunately, this approach is designed for optimization problems and does not apply directly to the graph isomorphism problem.

For heuristics of the second type the analyses have been less successful. One very common technique [29], [37] is to examine, either empirically or theoretically, an algorithm's behavior on input data drawn randomly from a given distribution (usually uniform). This approach is very misleading for the graph isomorphism problem since it has been shown empirically that a refinement algorithm (similar to the one presented in § 3) usually solves isomorphism problems on random graphs in $O(n^2)$ operations, where n is the number of nodes in each graph [15]. This empirical observation has been supported by the proof of Babai and Erdös [4] that a similar simple linear time algorithm will obtain a canonical ordering for almost all graphs. However, for many isomorphism problems dealing with graphs derived from combinatorial configurations, all known heuristic algorithms require a prohibitive amount of time and some interesting problems of moderate size remain unsolved. Another common technique is to test an algorithm on a small set of carefully chosen examples. Although this approach may establish isolated weaknesses of the particular algorithm, one cannot draw any conclusions about the overall value of the algorithm. To this end, Mathon [34] has compiled a list of pairs of graphs for which the graph isomorphism problem is seemingly very difficult.

In order to motivate the type of theoretical analysis which we will employ for heuristic graph isomorphism algorithms, we will briefly examine the typical approaches used in these algorithms (see § 3). First we present the terminology and definitions employed in the rest of the paper.

2. Terminology and definitions. The definitions and terminology used in this paper are compatible with those of Harary [24]. Throughout the paper, $G(V, E)$ refers to a graph with vertex set V or V_G (of cardinality n) and edge set E or E_G (of cardinality m). P_n, C_n and K_n respectively denote the path, cycle and complete graph on n vertices. $K_{p,q}$ denotes the complete bipartite graph with cell sizes p and q. Unless otherwise stated, all graphs are assumed to be undirected and loopless.

The *complement* $\bar{G}(V, F)$ of the graph $G(V, E)$ has $F=\{(x, y)|x\neq y \wedge (x, y)\notin E\}$. We now define the operations of union and composition applied to two given graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. In the *union*, $G_1\cup G_2$, the vertex set is $V_1\cup V_2$ and the edge set is $E_1\cup E_2$. The *composition* $G_1[G_2]$ has vertex set $V_1\times V_2$ with (x_1, x_2) adjacent to (y_1, y_2) whenever $[(x_1, y_1)\in E_1]$ or $[x_1=y_1$ and $(x_2, y_2)\in E_2]$.

The *distance* between any two points x, y in a graph G, $d_G(x, y)$ is the number of edges in a shortest chain between them. If there is no chain between x and y then $d_G(x, y)=\infty$. A graph G is *compact* if $d_G(x, y)=1$ or 2 for any two vertices x, y and $d_{\bar{G}}(x, y)=1$ or 2 for any two vertices x, y. $\gamma(G)$, the *girth* of a cyclic graph, is the length of the smallest cycle in G. If G is acyclic, then $\gamma(G)=\infty$.

For any vertex x, $\Gamma(x)$ denotes $\{y|(x, y)\in E\}$. The *degree* of vertex x, $\deg(x)=|\Gamma(x)|$; the maximum degree of all vertices in G is denoted by $\Delta(G)$. If all vertices have the same degree k then the graph is *regular of degree k* or *k-regular.* A graph G is called *strongly regular* if it is regular of degree k, $0<k<n-1$, and if any two adjacent (nonadjacent) vertices are adjacent to exactly λ (respectively μ) other vertices. The four integers (n, k, λ, μ) are the parameters of a strongly regular graph.

An *embedding* of a graph $G_1(V_1, E_1)$ into a graph $G_2(V_2, E_2)$ is an injection ϕ: $V_1\rightarrow V_2$ for which $(x, y)\in E_1 \Rightarrow (\phi_x, \phi_y)\in E_2$. The embedding is said to be *induced* if $(x, y)\in E_1\Leftrightarrow (\phi_x, \phi_y)\in E_2$. The vertex $\phi_x\in V_2$ is said to be *covered* (*under* ϕ) by the vertex $x\in V_1$.

Given two graphs $H(W, F)$ and $G(V, E)$ where $|W|=h<|V|$, we assign to each node $x\in V$ a vector $E_H(x)=(e_1, e_2, \cdots, e_h)$ where $e_i=$ the number of induced embeddings of H in G such that node x is covered by node i of H. Graph G is *H-regular* if $E_H(x)=E_H(y)$ $\forall x, y\in V$. Furthermore G is *c-subgraph regular* if it is H-regular for all $H(W, F)$ with $|W|\leqq c$. Clearly G is c-subgraph regular iff $\bar{G}$ is c-subgraph regular since $E_H(x)$ for $x\in \bar{G}=E_{\bar{H}}(x)$ for $x\in G$.

A one-to-one mapping σ of V onto V is an *automorphism* of G iff $(x, y)\in E\Leftrightarrow (\sigma x, \sigma y)\in E$ $\forall x, y\in V$. If there exists an automorphism mapping x onto y then x and y are *similar*, denoted $x\sim y$. A graph is *transitive* if $x\sim y$ $\forall x, y\in V$, it is *rigid* if $x\nsim y$

$\forall x, y \in V, x \neq y$. The *automorphism partition of G* (denoted AP(G)) is the partition of V induced by the equivalence relation $\sim$.

A *balanced incomplete block design* (BIBD) with parameters (v, b, r, k, λ) is an incidence system with v distinct objects and b blocks where each object belongs to r blocks, each block contains k objects and every pair of objects appears together in exactly λ blocks. A BIBD *graph* may be formed from a BIBD by representing the blocks and objects by nodes and the incidence relation by an undirected edge.

3. Heuristic graph isomorphism algorithms. Most heuristic graph isomorphism algorithms are based on *graph invariants* (or *g-invariants*), namely properties or parameters of a graph which must be preserved under isomorphism. Formally, an integer valued function I on graphs is a g-invariant if $G_1 \cong G_2 \Rightarrow I(G_1) = I(G_2)$. A g-invariant I is *complete* if $G_1 \cong G_2 \Leftrightarrow I(G_1) = I(G_2)$, (otherwise it is *incomplete*), that is it constitutes both a necessary and sufficient condition for isomorphism. Clearly, we can relax our definition to allow functions I which take values in α^*, where α is any finite alphabet. Thus the conjunction of a set of invariants can be viewed as a single invariant.

In practice, most heuristics used for the graph isomorphism problem are derived from heuristics for the automorphism partitioning problem. Such a derivation is given theoretical justification by the fact that the automorphism partitioning problem is isomorphism complete (this was proved by Karp; see [39] for a proof). Heuristics for the automorphism partitioning problem are usually designed to exploit various properties of vertices that must be preserved under automorphism. Formally, a *vertex invariant i* is a function which labels the vertices of an arbitrary graph with integers so that similar vertices are assigned the same label. If we denote $i(G)$ by i_G then $i_G\colon V_G \to \mathbb{N}$ and $\forall x, y \in V_G, x \sim y \Rightarrow i_G(x) = i_G(y)$. As with graph invariants, it is sufficient that the range of $i(G)$ be α^*, where α is any finite alphabet. Examples of simple v-invariants include the degree or the number of triangles containing the specified vertex. More complex v-invariants may be formed by the conjunction of several simpler v-invariants. Further examples of commonly used v-invariants are presented in § 5.

The sense in which g-invariants are derived from v-invariants is made precise as follows: if i is a v-invariant and $I(G)$ denotes the multiset $\{i_G(x) | x \in V_G\}$ then I is called the *derived graph invariant* (or *derived g-invariant*) of i. The eigenvalue spectrum of a graph's adjacency matrix is an example of a nonderived g-invariant. Throughout this paper, we will be concerned primarily with derived g-invariants.

Let i be some complete v-invariant. An algorithm evaluates i if, given an arbitrary graph G and a vertex $x \in V_G$, it computes $i_G(x)$. An algorithm verifies i if, given an arbitrary graph G, vertex $x \in G_G$ and integer k, it decides whether $i_G(x) = k$. Motivation for the study of complete v-invariants stems from the fact that if any complete v-invariant can be evaluated (respectively, verified) in time bounded by some polynomial in the size of G, then the automorphism partitioning problem, and hence the graph isomorphism problem, would belong to P (respectively, co-NP). No such complete v-invariants are known. In fact, many v-invariants are known to be *incomplete* in the sense that for such a v-invariant i, there exists a graph G and dissimilar vertices $x, y \in V_G$ for which $i_G(x) = i_G(y)$. Such a graph is said to demonstrate the incompleteness of i.

Many v-invariants (and their derived g-invariants) although known to be incomplete, are nevertheless of practical value. For example, if few graphs demonstrating the incompleteness are known, one is tempted to employ the v-invariant in the hope that one of these graphs will not arise in practice. Furthermore, although an incomplete v-invariant cannot guarantee producing the automorphism partition, it may produce some nontrivial partition of V; the automorphism partition will be a refinement (possibly trivial) of this partition. Any nontrivial partition of V will reduce from $n!$ the number of $V \to V$ mappings which could be an automorphism. Once V has been partitioned, it is standard practice to employ a refinement procedure in the hope of producing a further refinement of V. If the resulting partition is fine enough, it is feasible to use a backtracking algorithm to determine the automorphism partition. This refinement operation is used extensively in practical algorithms and warrants a more detailed examination. The following pseudo *PL/I* algorithm outlines a typical refinement procedure.

Refinement procedure. Given an initial partitioning $V^1, V^2, \cdots . V^l (1 \leqq i < n)$

```
$i' \leftarrow 0$
do while $i' \neq i$
    $i' \leftarrow i$
    to each $x \in V$ assign the list $L_x = (a_1, a_2, \cdots, a_i)$ where
        $a_j = |\{y \mid y \in V^j \wedge (x, y) \in E\}| \ (1 \leqq j \leqq i)$
    do $j = 1$ to $i$
        lexicographically order the lists corresponding to
        the vertices in cell $V^j$. Assume there are $l$
        different lists
        $i' \leftarrow i' + l - 1$
        refine cell $V^j$ into $l$ new cells
    end
end
```

Hopcroft [25] has shown that an equivalent procedure has a time bound of $O(m \log n)$. (For a formal description and analysis of the Hopcroft algorithm see [23].)

As an example of this procedure consider the graph in Fig. 1 with the initial partition being V itself. After h iterations of the main loop in the refinement procedure the partition is $(2)(3, n)(4, n-1), \cdots, (h+1, n-h+2)(h+2, h+3, \cdots, n-h+1)(1)$.

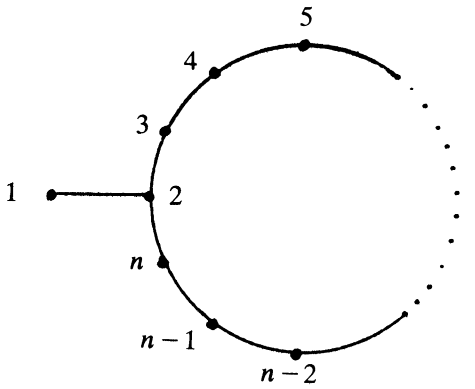


FIG. 1

If the input partition to the refinement procedure is V itself, then as the example indicates, the procedure initially calculates the degree partition and endeavors to refine it. If the graph is regular, then the refinement procedure will not be able to make any refinement unless the input partition is nontrivial. On the other hand, if the graph is a tree, then the refinement procedure will always refine V into the automorphism partition [15].

One further tactic which is always used in automorphism partitioning (and graph isomorphism) algorithms is to apply the particular v-invariant to both the given graph G

and its compliment $\bar{G}$. This acknowledges the fact that the automorphism partition of G is identical to the automorphism partition of $\bar{G}$ and yet an invariant i may fail to distinguish between two dissimilar vertices in G and succeed in $\bar{G}$. On the other hand, if the v-invariant i satisfies $i_G(x)=i_G(y)$, and $i_{\bar{G}}(x)=i_{\bar{G}}(y)$, $\forall x, y \in V_G$, where G is nontransitive and regular then not only is i incomplete, it provides absolutely no information about the automorphism partition of at least one nontransitive graph G. If a given v-invariant is strongly incomplete in this sense for a broad class of graphs we may be justifiably pessimistic about its use as a heuristic for automorphism partitioning. This leads to our notion of v-invariants and derived g-invariants being universally incomplete.

For a given v-invariant i we define S_i to be $\{G|G$ is nontransitive and regular, and $i_G(x)=i_G(y)$ and $i_{\bar{G}}(x)=i_{\bar{G}}(y)\ \forall x, y \in V_G\}$.

A v-invariant is *universally incomplete* if there exists a polynomial time bounded graph transformation T_i such that for any graph G, $T_i(G)\in S_i$ and furthermore $G_1 \cong G_2 \Leftrightarrow T_i(G_1)\cong T_i(G_2)$ for any G_1, G_2 (that is G is uniquely represented by a graph which demonstrates the strong incompleteness of i). As a corollary of universal incompleteness we establish the isomorphism completeness of the family S_i.

We now extend the concept of universal incompleteness to derived g-invariants. If i is a v-invariant, let

$$P_i=\{\langle G, G'\rangle | G, G' \in S_i,\ |V_G|=|V_{G'}|,\ |E_G|=|E_{G'}| \text{ and } i_G(x)=i_{G'}(y) \text{ and } i_{\bar{G}}(x)=i_{\bar{G}'}(y),\ \forall x \in V_G,\ \forall y \in V_{G'}\}.$$

If I is the derived g-invariant of i, then I is *universally incomplete* if there exists a polynomial time bounded transformation T_I taking an arbitrary pair of graphs $\langle G_1, G_2\rangle$ into a unique pair $\langle G_1', G_2'\rangle \in P_i$. It should be noted that i being universally incomplete does not imply that the derived g-invariant of i is universally incomplete since $\langle T_i(G_1), T_i(G_2)\rangle$ might not be a member of P_i. In fact it is possible for the derived g-invariant of a universally incomplete v-invariant to be complete.

In § 5 we shall discuss commonly used v-invariants and in later sections show that all of these v-invariants (and their derived g-invariants), both individually and collectively, are universally incomplete. As a consequence of this, we shall establish new isomorphism complete restricted families of graphs. Furthermore, we will show that the automorphism partitioning problem on these restricted families of graphs is also isomorphism complete. (This may not follow in general; consider for example rigid or transitive graphs.) In the next section we present some known isomorphism complete problems.

4. Known isomorphism complete problems. Presently known isomorphism complete problems (see also [6]) fall into the following three classes: automorphism problems, isomorphism complete restricted families of graphs and isomorphism problems on nongraph structures. As mentioned previously the general automorphism partitioning problem is isomorphism complete. Babai [3] and Mathon [35] have independently established that determining the order of a graph's automorphism group is also isomorphism complete.

Examples of complete restricted families of graphs include digraphs, labeled graphs, bipartite graphs, line graphs (see [24]), chordal graphs [7], transitively orientable graphs [7], acyclic rooted digraphs [1] and regular graphs [5], [36]. The question of whether c-regular graph isomorphism (for some fixed c) is isomorphism complete is still an open problem; this and other related issues will be discussed in § 7. As mentioned in § 1, there is little reason to assume that current practical algorithms would work any better or worse on these restricted graphs (with the possible exceptions of bipartite and regular graphs) than on general graphs. For some restricted families of graphs (e.g., planar graphs [26], graphs with distinct eigenvalues [3], transitive series parallel digraphs [33] and interval graphs [8]) the isomorphism problem is known to be polynomial. Isomorphism problems on other structures such as semigroups [5], finite automata [5], and finitely presented algebras [32] are known to be isomorphism complete. Finally, the problem of determining whether a regular graph is self-complementary is also isomorphism complete [12].

5. Commonly used vertex invariants. As pointed out in § 3, most heuristic algorithms for the graph isomorphism and automorphism partitioning problems utilize v-invariants. We now present the v-invariants which are commonly used in practice. Typically, these v-invariants are applied to both the given graph(s) and the complement(s) and are used in conjunction with a refinement procedure (see § 3). For a survey of practical algorithms see [39].

The degree v-invariant is automatically determined (and refined) by a standard refinement procedure. This v-invariant is a special case of the general *subgraph embedding* v-*invariant* which involves the different ways of embedding a particular graph H in the given graph G, where h, the order of H is less than the order of G. Each node x of G is assigned an *H-vector* $E_H(x)=(e_1, e_2, \cdots, e_h)$ where $e_i=$ the number of embeddings of H where x is covered by node i of H. For example, the degree of a node x in G is one half the number of embeddings of K_2 into G such that x is covered by a node of K_2. In particular [18], [9] and [22] have shown that this invariant based on small complete and void graphs together with the refinement procedure works very well for strongly regular and BIBD graphs. Other commonly used embedding v-invariants specify H as a path or cycle. In general, we may consider using an embedding v-invariant based on all subgraphs up to a fixed size c. Schmidt and Druffel [42] use a v-invariant based on the distance matrix of the given graph. Under this scheme each vertex x is assigned a *distance vector* $(d_1, d_2, \cdots, d_{n-1}, d_\infty)$ where $d_i=$ the number of vertices distance i from x. In §§ 6–9 we will show that both individually and collectively all these v-invariants and the associated derived g-invariants are universally incomplete.

6. Transformations. We now introduce the three types of transformations which will be used to show that the various v-invariants are universally incomplete. In § 9, we will also prove that it is sufficient to use these transformations to show that the corresponding derived g-invariants are also universally incomplete. For each transformation various properties will be stated. Throughout this and subsequent sections, we will assume that for all given graphs each connected component has maximum degree $\geqq 3$. If this is not the case, then minor modifications are required.

The first transformation involves node replacement. In the general case node $x \in V_G$ will be replaced by a graph R_x where R_x has deg (x) connector nodes. The deg (x) edges incident with x are arbitrarily assigned to the deg (x) connector nodes in R_x. In constructing such a node replacement graph R_x, one must in general assure that the group of automorphisms which stabilize the set of connector nodes induces the symmetric group on the set. We will use the node replacement transformation in the restricted situation where G is k-regular thereby permitting the same replacement graph to be used for each node in G.

NODE (R, k): The node replacement transformation. If $\Delta(G')$ (where $G'=$ NODE $(R, k)[G]$) is to equal $\Delta(G)$ then each connector node in R must have degree $\leqq$

$k-1$ and each nonconnector node must have degree $\leqq k$. One obvious candidate for such a replacement graph is K_k. The transformation NODE has been studied by various researchers (see [36] and [10]) in an attempt to determine whether c-regular graph isomorphism (for some constant c) is isomorphism complete; further details are given in § 7.

As an example of the node replacing transformation, one way of transforming a k-regular multigraph G into a k-regular graph G' is by setting $G' = \text{NODE}\ (K_k, k)[G]$. The general node replacing transformation can be used to transform an arbitrary multigraph into a general graph and thereby illustrate that multigraph isomorphism is isomorphism complete. The following properties hold for $G' = \text{NODE}\ (K_k, k)[G]$ where G is a k-regular graph.

NR1. The transformation NODE (K_k, k) can be computed in polynomial time.

NR2. G' uniquely represents G.

Proof. This can be shown by demonstrating how G may be reconstructed from G'. Since every k-clique in G' corresponds to a vertex in G, G may be immediately reconstructed by shrinking these cliques. □

NR3. G' has nk nodes, is regular of degree k and is nontransitive if G is nontransitive.

NR4. AP (G) is polynomially recoverable from AP (G').

Proof. Given node $x \in G$, let the vertices in the clique replacing x be denoted x_y where $y \in \Gamma(x)$. Thus in G', $(x_y, w_z) \in E'$ iff either (i) $x = w$ or (ii) $x = z$ and $y = w$. The result follows from the observation that $x_w \sim y_z$ in $G' \Rightarrow x \sim y$ in G. □

The second transformation replaces each edge of the given graph G by a specified graph R. As with the NODE transformation, the edge replacement transformation could be extended to allow each edge e of G to be replaced by a specific graph R_e.

EDGE (R, α, β): The edge replacing transformation. The replacement graph R has two connector nodes α and β. Each edge (x, y) in the given graph is replaced by R where the remaining edges incident with x are now incident with α (or β) and the remaining edges incident with y are now incident with β (or α).

In constructing such a replacement graph R one must in general assure that $\alpha \sim \beta$. Otherwise, the graph resulting from the transformation may not be uniquely defined. If $\alpha \sim \beta$, then it doesn't matter whether x is paired with α or β; thus $G' = \text{EDGE}\ (R, \alpha, \beta)[G]$ is uniquely defined. To illustrate this transformation we shall consider the generalized edge replacement transformation where each edge e in G is replaced by an individual R_e. This transformation will be used to transform G, a graph with labeled edges onto an unlabeled graph G'. Assume the edges are labeled E_j $(1 \leqq j \leqq m)$ and replace an edge with label E_j with the graph given in Fig. 2.

Graphs with labeled vertices can be handled in a similar fashion. As a consequence, labeled graph isomorphism is known to be isomorphism complete.

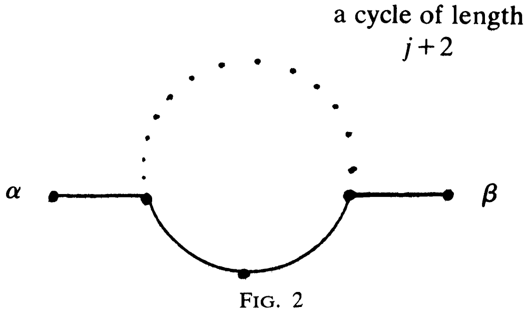


FIG. 2

Returning to the uniform edge replacement transformation, the following properties hold for $G' = \text{EDGE}(R, \alpha, \beta)[G]$ where $\alpha \sim \beta$.

ER1. G' can be computed in time polynomial in n and the size of R.

ER2. If R is connected, then G' uniquely represents G.

Proof. We first observe that node $x \in V_{G'}$ with $\deg_{G'}(x) \geqq 3$ corresponds to a node $x \in V_G$ iff the number of disjoint embeddings of R in G' such that α (or β) covers $x = \deg_{G'}(x)$. Since each connected component in G has at least one vertex with degree $\geqq 3$, we are able to recover the nodes of G from G'. □

ER3. G' has $m \cdot (|V_R| - 2) + n$ nodes and is nontransitive if G is nontransitive. Furthermore, if $\deg_R(\alpha) = \deg_R(\beta) = 1$, then $\Delta(G') = \max(\Delta(G), \Delta(R))$.

ER4. If R is connected and α and β are the only nodes of R with degree 1, then AP (G) is polynomially recoverable from AP (G').

Proof. Since $\alpha \sim \beta$ in R we see that if $x \sim y$ in G' and x and y correspond to vertices in G then $x \sim y$ in G. □

The final transformation is based on the graph operation of composition $G_1[G_2]$ defined in § 2. In our usage of this operation, we will always set G_1 to C_5 (see Fig. 3).

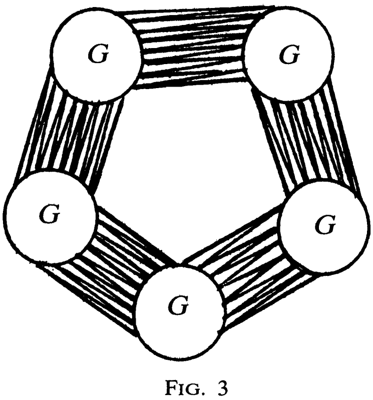


FIG. 3

COMP (C_5): The composition transformation. The following properties hold for $G' = \text{COMP}(C_5)[G]$:

CO1. The transformation COMP (C_5) can be computed in polynomial time.

CO2. G' uniquely represents G.

Proof. We prove this by reconstructing G by noting that two vertices $x, y \in G'$ lie in the same copy of G iff $|\Gamma_x \cap \Gamma_y| \geqq 2n$. □

CO3. G' is compact with order $5n$. Furthermore, if G is k-regular and nontransitive, then G' is $(2n+k)$-regular and nontransitive.

Proof. The compactness of G' can be established by considering any two nonadjacent vertices x, y in G'. If x and y belong to the same copy of G, then clearly $d_{G'}(x, y) = 2$ by following edges to a neighboring copy of G. If x and y belong to different copies of G, then there exists a unique copy of G which is adjacent to both of these copies, thereby establishing that $d_{G'}(x, y) = 2$. To show that $\bar{G}'$ also has the compactness property, we note that $\overline{\text{COMP}(C_5)[G]} \cong \text{COMP}(C_5)[\bar{G}]$ and use the above argument. □

CO4. AP (G) is polynomially recoverable from AP (G').

These three transformations are now used to develop new transformations for establishing that various v-invariants are universally incomplete. In § 9 it will be shown that these new transformations will also prove the universal incompleteness of the associated derived g-invariants.

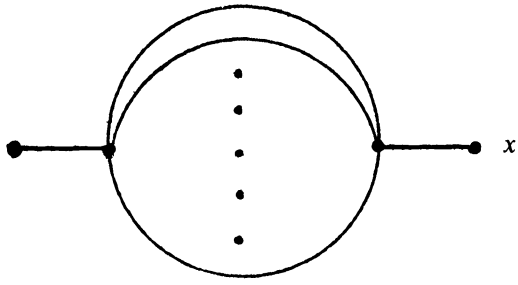
7. The production of nontransitive regular graphs. In order to show a v-invariant i to be universally incomplete, we must be able to transform any given graph G into $G' \in S_i$ where G' has various properties including nontransitivity and regularity. In this section we develop a transformation which given any graph G produces a nontransitive regular graph G'. The problem of transforming any given graph G into a regular graph G' has received considerable attention. Booth [5] constructs a graph G' which is m-regular whereas Miller [36] constructs a G' with regularity $\Delta(G)$ if $\Delta(G)$ is odd and $\Delta(G)+1$ if $\Delta(G)$ is even. In the following, we will show how to produce a nontransitive $\Delta(G)$-regular graph regardless of the parity of $\Delta(G)$. To make the given graph nontransitive we simply add a copy of $K_{1,3}$. The resulting graph uniquely represents G and is nontransitive. The $\Delta(G)$-regularity is accomplished by applying the following transformation, DEG, to this nontransitive graph. Providing $k \geqq \Delta(G)$, DEG constructs a G' which is k-regular.

DEG (k): The degree transformation.

(i) Form two disjoint copies of G denoted G_1 and G_2. If $x \in V_G$ then the corresponding vertices in V_{G_1} and V_{G_2} are denoted x_1 and x_2 respectively.

(ii) For each $x \in V_G$ add k-deg(x) multiple edges of the type given in Fig. 4 between x_1 and x_2. This new graph, $\hat{G}$ (possibly a multigraph) is k-regular.

(iii) Set $G' = \text{NODE}(K_k, k)[\hat{G}]$.



$k-1$ edges

FIG. 4

For the graph G in Fig. 5 and $k = 3$, the results of steps (ii) and (iii) are illustrated in Figs. 6 and 7 respectively.

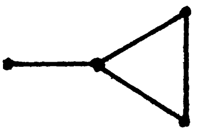


FIG. 5

Provided $k \geqq \Delta(G)$, the following properties hold for $G' = \text{DEG}(k)[G]$:

DE1. The transformation DEG can be computed in polynomial time.

DE2. G' uniquely represents G.

Proof. This follows from using property NR3 to retrieve $\hat{G}$. The retrieval of G is then straightforward. □

DE3. G' is nontransitive, k-regular and has order $2k(n+nk-2m)$.

DE4. AP (G) is polynomially recoverable from AP (G').

As mentioned in § 4, the question of whether c-regular graph isomorphism is isomorphism complete is unresolved. In particular is the cubic graph isomorphism

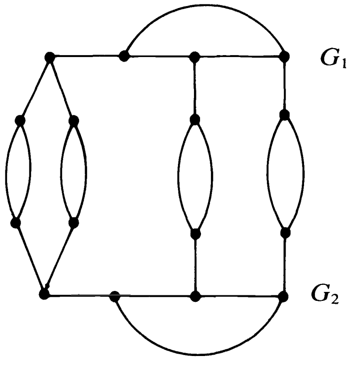


FIG. 6

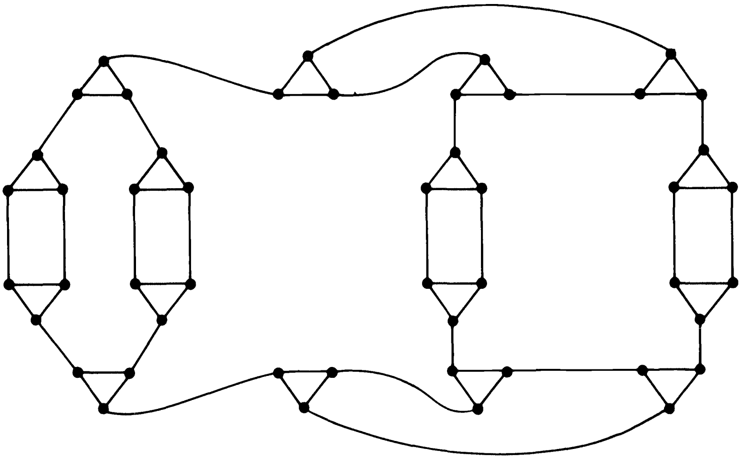


FIG. 7

problem isomorphism complete? (Obviously 2-regular isomorphism $\in P$.) Furthermore, is k-regular graph isomorphism equivalent to $(k-1)$-regular graph isomorphism? One approach to this problem is to use NODE, the node replacement transformation described in the preceding section. Miller [36] has shown that for $k \neq 5$, no appropriate replacement graph exists. For $k = 5$, Carter [10] has constructed a valid replacement graph thereby showing that 5-regular isomorphism is equivalent to 4-regular isomorphism. A related problem concerns the placement of the isomorphism of arbitrary irregular graphs into this hierarchy. Our transformation DEG shows that an arbitrary graph G may be uniquely represented by a $\Delta(G)$-regular graph G'. If one were able to produce a G' with lower regularity for $\Delta(G) > 3$ then as a consequence, the isomorphism completeness of the k-regular graph isomorphism problem would be established.

8. The universal incompleteness of various vertex invariants and new isomorphism complete problems. The transformation DEG may be used to represent an arbitrary graph G by a nontransitive k-regular ($k \geqq \Delta(G)$) graph G'. We now turn our attention to showing that various v-invariants are universally incomplete. In particular for v-invariant i and a given graph G, we want to construct a nontransitive regular graph G' where $i_{G'}(x) = i_{G'}(y)$ and $i_{\bar{G}'}(x) = i_{\bar{G}'}(y)$ $\forall x, y \in V_{G'}$. The first v-invariant we examine is the seemingly powerful (albeit expensive) subgraph embedding v-invariant. Recall that each vertex $x \in G$ is assigned a vector $E_H(x) = (e_1, e_2, \cdots, e_h)$ where $h = |H|$ and e_i = the number of embeddings of H in G such that vertex x is covered by vertex i of H. Graph G is H-regular if $E_H(x) = E_H(y)$ $\forall x, y \in G$ and a graph is c-subgraph regular if it is H-regular for all H where $|H| \leqq c$. We now show that a v-invariant based on the H-vectors for any or all H with $|V_H| \leqq$ an arbitrary constant c is universally incomplete.

First we introduce the GIRTH transformation which is used to construct a graph G' which is k-regular with given girth g.

GIRTH (g, k): The girth transformation. This transformation is a special case of the edge replacement transformation EDGE. In particular we need to specify an edge replacement graph R (with connector nodes α, β) for such k and g. The existence of k-regular graphs with girth $\geqq g$ for arbitrary k and g will be discussed subsequently; for now we assume that we have a connected k-regular graph $\mathscr{G}$ with girth $\geqq g$. Choose an arbitrary nonbridge edge (x, y) of $\mathscr{G}$ and remove it. Take two copies $\mathscr{G}_1$, $\mathscr{G}_2$ of this graph and denote the x, y vertices as x_1, y_1, x_2, y_2. Form R by adding the edges (y_1, y_2), (α, x_1) and (β, x_2) where α and β are new vertices. (See Fig. 8.) Clearly $\alpha \sim \beta$.

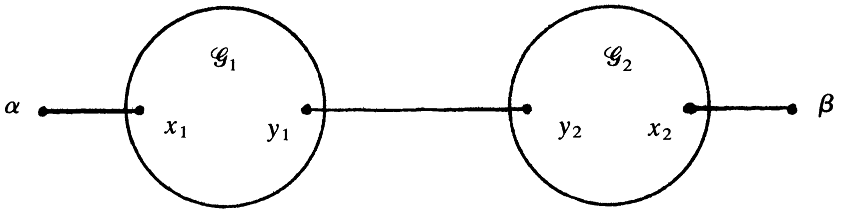


FIG. 8

This graph R will replace every edge in a nontransitive regular graph formed by the transformation DEG (k). Thus GIRTH (g, k)[G] is defined to be EDGE (R, α, β)[DEG (k)[G]] where $k \geqq \Delta(G)$. Before stating the properties of G' = GIRTH (g, k)[G] we discuss the existence and construction of connected graphs with specified girth and regularity. Erdös [19] has shown that $f(k, g)$, the minimum number of nodes required for a k-regular graph to have girth $\geqq g$ satisfies:

$$f(k, g) \leqq 4\left\{\frac{(k-1)^{g-1}-1}{k-2}\right\} = f'(k, g).$$

This bound is quite tight, since Kárteszi [30] has shown that

$$f(k, g) \geqq \frac{k(k-1)^r-2}{k-2} \quad \text{if } g = 2r+1,$$

$$f(k, g) \geqq \frac{2(k-1)^r-2}{k-2} \quad \text{if } g = 2r.$$

Erdös' proof of the $f'(k, g)$ upper bound for $f(k, g)$ can be altered in a straightforward manner to produce an algorithm for constructing $\mathscr{G}$ and thus R. The graph

$G' = \text{GIRTH } (g, k)[G]$ (i.e., $= \text{EDGE } (R, \alpha, \beta)[\text{DEG}(k)[G]]$ where $k \geqq \Delta(G)$) has the following properties:

GI1. The transformation GIRTH can be computed in polynomial time.

GI2. G' uniquely represents G.

Proof. This follows immediately from properties ER2 and DE2.

GI3. G' is nontransitive, k-regular and has order $2k(n+nk-2m)\cdot[k\cdot f'(k, g)+1]$. Furthermore $\gamma(G') \geqq g$.

GI4. AP (G) is polynomially recoverable from AP(G').

It remains to be shown that the c-subgraph embedding v-invariant is universally incomplete for any or all subgraphs of order $\leqq c$. Sachs [40], [41] has studied the number of embeddings of forests in regular graphs of high girth. This work has been extended in [31] to determine the number of embeddings of forests in regular high girth graphs with the added constraint that a particular vertex is covered by the forest. The following theorem which is an immediate consequence of Theorem 5.6 in [31] motivates the GIRTH transformation.

THEOREM 1. *If G is regular, then G is $(\gamma(G)-1)$-subgraph regular.*

We now establish the universal incompleteness of the c-subgraph embedding v-invariant.

LEMMA 1. *For any constant c, the c-subgraph embedding v-invariant is universally incomplete.*

Proof. The universal incompleteness of this invariant follows from consideration of the transformation

$$G' = \text{GIRTH } (c+2,k)[G].$$

Provided $k \geqq \Delta(G)$ this transformation establishes that the vertex invariant is universally incomplete from properties GI1, GI2, GI3 and Theorem 1. □

We now examine the distance v-invariant.

LEMMA 2. *The distance v-invariant is universally incomplete.*

Proof. Consider the transformation $G' = \text{COMP } (G_5)[\text{DEG } (k)[G]]$ where $k \geqq \Delta(G)$. The universal incompleteness of the distance v-invariant follows from properties CO1, CO2, CO3, DE1, DE2 and DE3. □

Note that both G' and $\bar{G}'$ are at least $2n$-connected. Repeated compositions with C_5 can make the connectivity arbitrarily high.

Having shown that individually the embedding and distance v-invariants are universally incomplete, it remains to establish that together they are also universally incomplete. First we prove the following:

LEMMA 3. *If G is c-subgraph regular, then $C_5[G]$ is also c-subgraph regular.*

Proof. We wish to show that $E_H(x) = E_H(y)$ $\forall x, y \in C_5[G]$ and for all H with $|V_H| \leqq c$. From the automorphisms on $C_5[G]$ we may assume that x and y belong to the same copy of G denoted $\tilde{G}$. Any embedding of H in $C_5[G]$ has a subset H_1 of H embedded in $\tilde{G}$ and a subset H_2, the rest of H, embedded outside $\tilde{G}$. (H_2 may be the null graph.) The number of ways of embedding H_2 outside $\tilde{G}$ is independent of the particular embedding of H_1 inside $\tilde{G}$. This follows from the construction of $C_5[G]$ and the c-subgraph regularity of G. Since $|V_{H_1}| \leqq c$ and $\tilde{G}$ is c-subgraph regular, $E_{H_1}(x) = E_{H_1}(y)$ $\forall x, y \in \tilde{G}$. Thus $E_H(x) = E_H(y)$ $\forall x, y \in C_5[G]$. □

To prove that collectively the two v-invariants are universally incomplete we will use the following transformation:

$$G' = \text{COMP } (C_5)[\text{GIRTH } (c+2, k)[G]].$$

THEOREM 2. *The v-invariant consisting of the conjunction of the distance v-invariant and the c-subgraph embedding v-invariant for any constant c is universally incomplete.*

Proof. This follows from using the properties of the preceding transformation and Lemmas 1, 2 and 3. □

As a consequence of this theorem and various previous properties we have the following corollaries.

COROLLARY 1. *For any constant c, the compact c-subgraph regular graph isomorphism problem is isomorphism complete.*

COROLLARY 2. *For any constant c, the problem of determining the automorphism partitioning of a compact c-subgraph regular graph is isomorphism complete.*

We now turn our attention to the various derived g-invariants and show that they are universally incomplete.

9. The universal incompleteness of the associated derived g-invariants. We now show that the transformations used to establish the universal incompleteness of the various vertex invariants also establish the universal incompleteness of the corresponding graph invariants. Before applying these transformations to arbitrary G_1 and G_2 we must transform G_1 and G_2 into nontransitive graphs with the same numbers of nodes and edges. This is accomplished by the following transformation.

STAND (N, M): The standardizing transformation. This transformation maps the pair of graphs G_1 and G_2 onto the pair G'_1, G'_2 both with N nodes and M edges, provided N and M are large enough with respect to n_1, m_1, n_2 and m_2. The transformation is outlined by describing its effect on a single given graph G. G' is produced from G by forming the union of G with a cycle of size n, another cycle of size $n+m$ and $M-2(n+m)$ copies of K_2 and $N+n+3m-2M$ copies of K_1. The following four properties hold for $(G'_1, G'_2) = \text{STAND}\ (N, M)[G_1, G_2]$. In each property the pair of graphs (G, G') refers either to the pair (G_1, G'_1) or (G_2, G'_2).

ST1. The transformation STAND can be computed in polynomial time.

ST2. G' uniquely represents G.

ST3. $\Delta(G') = \Delta(G)$ and G' is nontransitive.

ST4. AP (G) is polynomially recoverable from AP (G').

Having produced graphs G'_1 and G'_2, both nontransitive and with N nodes and M edges, we are in a position to demonstrate that previously introduced transformations will establish the universal incompleteness of the various derived g-invariants. Instead of individually examining each v-invariant discussed in §§ 7 and 8, we will concentrate on the v-invariant formed by the conjunction of these v-invariants. In particular we will show that the derived g-invariant corresponding to the conjunction of the distance and c-subgraph embedding (for any constant c) v-invariants is universally incomplete. Similar results will obviously hold for the various individual derived g-invariants.

THEOREM 3. *The derived g-invariant corresponding to the conjunction of the distance and c-subgraph embedding (for any constant c) v-invariants is universally incomplete.*

Proof. Given G_1 and G_2 we produce G'_1 and G'_2 by the transformation STAND (N, M)[G_1, G_2]. Let $G''_i = \text{COMP}\ (C_5)[\text{GIRTH}\ (c+2, k)[G'_i]]$, $i = 1, 2$. If N and M are sufficiently large with respect to n_1, n_2, m_1 and m_2 and $k \geqq \max(\Delta(G_1), \Delta(G_2))$ then both G''_1 and G''_2 are compact and c-subgraph regular. It remains to show that their subgraph regularities are identical. This is established by the following two lemmas. The first is another consequence of Theorem 5.6 in [31].

LEMMA 4. *If G is k-regular and $\gamma(G) > |V_H|$ then for any $x \in V_H$ and $y \in V_G$, the number of ways of embedding H in G such that x covers y depends only on H, k and $|V_G|$.*

LEMMA 5. *Given two graphs H and G with $|V_H| < |V_G|$. For any $v \in V_H$, $y \in V_G$, if the number of ways of embedding H in G with x covering y depends only on H, x and $|V_G|$, then the same property holds for $C_5[G]$.*

The proof of Lemma 5 is very similar to that of Lemma 3 and is omitted. Together the above lemmas show that G_1'' and G_2'' are indistinguishable on the basis of embeddings of subgraphs of order less than or equal to c. Thus, the derived g-invariant is universally incomplete □

10. Concluding remarks. In the preceding sections we have shown that many seemingly powerful invariants are universally incomplete. Thus any heuristic isomorphism or automorphism partition algorithm based on any or all of these invariants together with the refinement procedure, must fail to give any information regarding isomorphism or the automorphism partition for a very broad class of graphs. As an outcome of these results, it was shown that for any constant c, the compact c-subgraph regular isomorphism problem is isomorphism complete. These results seem to indicate the sterility of heuristics which attempt to gain global graph information (namely isomorphism or automorphism properties) from local graph properties. The compact c-subgraph regular graphs unlike other known incomplete families, do lend some insight into the difficulty of the graph isomorphism problem insofar as most commonly used heuristic isomorphism algorithms fail utterly on these graphs.

Although the basic refinement procedure presented in § 3 is the most commonly used such procedure, considerable effort has been spent on the development of more powerful refining algorithms. One such attempt [17] is to examine individually each node x which belongs to a "final" cell of order $\geqq 2$. This is done by assigning a unique label to x (i.e., forcing $x \not\sim y$ $\forall y \in V$, $y \neq x$) and using the refinement procedure to try to construct the automorphism partition of G_x, the graph with x assigned to a unique cell. Clearly, if $x \sim y$ then the outcome of the refinement procedure applied to G_x is identical to the outcome when applied to G_y. If the outcomes are not the same, and x and y belonged to the same "final" cell in the original partition then this cell can be refined further. In some cases (e.g., strongly regular graphs), this approach is known to fail [15]. See [15] for other refinement procedures which are useful for difficult graphs of this type.

For these various strengthened refinement procedures, it is natural to modify our definitions of universally incomplete invariants to encompass these procedures. One would then like to show that the various invariants considered in this paper are still universally incomplete under these strengthened definitions. It is interesting to note that our transformations do not succeed in this task since the strengthened refinement procedure outlined in the previous paragraph will succeed in partitioning some of the nontransitive compact c-subgraph regular graphs produced by our transformations. Since strongly regular graphs are known to be impervious to this refining technique, one would like to strengthen the definitions of universal incompleteness by replacing the regularity requirement by a strongly regular requirement. If any v-invariant could be shown to be universally incomplete under this strengthened definition, one would have established that strongly regular graph isomorphism is isomorphism complete. Since strongly regular graphs form counter-examples to many conjectures (and algorithms) in the graph isomorphism area, such a result would attract great interest. A similar comment applies to BIBD graph isomorphism. Another intriguing open problem is to determine whether k-regular graph isomorphism is equivalent to $(k-1)$-regular graph isomorphism for $k \neq 5$. In particular is 4-regular isomorphism equivalent with 3-regular isomorphism? It is interesting to note that for all of these open questions, the general techniques of node and/or edge replacement may be shown to fail.

Other open questions suggested by this paper include the following: Find the minimum value of r such that G being r-subgraph regular implies that G is transitive. What if $r = n - 1$? Also if G is K_t-regular and $\bar{K}_t$-regular for all t does that imply that G is transitive? If G is K_t-regular and $\bar{K}_t$-regular for $t \leqq c$, is G c-subgraph regular?

Acknowledgment. The authors wish to thank the Natural Sciences and Engineering Research Council of Canada for financial assistance.

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. P. AMBLER, H. G. BARROW, C. M. BROWN, R. M. BURSTALL AND R. J. POPPLESTONE, *A versatile computer-controlled assembly system*, Proc. of Third International Joint Conf. on Artificial Intelligence, Stanford (1973), pp. 298–307.

[3] L. BABAI, *On the isomorphism problem*, Proc. FCT Conf., Poznan–Kornak (1977).

[4] L. BABAI AND P. ERDÖS, *Random graph isomorphism*, to be submitted.

[5] K. S. BOOTH, *Isomorphism testing for graphs, semigroups, and finite automata are polynomial equivalent problems*, this Journal, 7 (1978), pp. 273–279.

[6] K. S. BOOTH AND C. J. COLBOURN, *Problems polynomially equivalent to graph isomorphism*, TR CS-77-D4, Dept. of Computer Science, University of Waterloo (1979).

[7] K. S. BOOTH AND G. S. LUEKER, *Linear algorithms to recognize interval graphs and test for consecutive ones property*, Proc. 7th Annual ACM Symp. on Theory of Computing (1975), pp. 255–265.

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

[9] F. C. BUSSEMAKER AND J. J. SEIDEL, *Symmetric Hadamard matrices of order 36*, Tech. Rep. Dept. of Mathematics, Technological University of Eindhoven (1970).

[10] L. CARTER, *A four-gadget*, SIGACT News 9 (1977), p. 36.

[11] C. J. COLBOURN, *A bibliography of the graph isomorphism problem*, TR 123/78, Dept. of Computer Science, Univ. of Toronto, 1978.

[12] C. J. COLBOURN AND M. J. COLBOURN, *Isomorphism problems involving self-complementary graphs and tournaments*, Proc. 8th Manitoba Conference on Numerical Computing (1978), to appear.

[13] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd Annual ACM Symp. on Theory of Computing (1971), pp. 151–158.

[14] S. A. COOK AND R. A. RECKHOW, *On the lengths of proofs in the propositional calculus*, Proc. 6th Annual ACM Symp. on Theory of Computing (1974), pp. 135–148.

[15] D. G. CORNEIL, *Graph isomorphism*, Ph.D. thesis, Univ. of Toronto (1968).

[16] ———, *Recent results on the graph isomorphism problem*, Proc. 8th Manitoba Conference on Numerical Computing (1978), to appear.

[17] D. G. CORNEIL AND C. C. GOTLIEB, *An efficient algorithm for graph isomorphism*, J. Assoc. Comput. Mach., 17 (1970), pp. 51–64.

[18] D. G. CORNEIL AND R. A. MATHON, *Algorithmic techniques for the generation and analysis of strongly regular graphs and other combinatorial configurations*, Ann. Discrete Math. 2 (1978), pp. 1–32.

[19] P. ERDÖS AND H. SACHS, *Reguläre Graphen gegebener Taillenweite mit minimaler Knotenzahl*, Wiss. Z. Univ. Halle (12), 3 (1963), pp. 251–258.

[20] M. R. GAREY AND D. S. JOHNSON, *Approximation algorithms for combinatorial problems: An annotated bibliography*, Algorithms and Complexity, New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 41–52.

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

[22] P. B. GIBBONS, R. A. MATHON AND D. G. CORNEIL, *Computing techniques for the construction and analysis of block designs*, Utilitas Math., 11 (1977), pp. 161–192.

[23] D. GRIES, *Describing an algorithm by Hopcroft*, Acta Informat., 2 (1973), pp. 97–109.

[24] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[25] J. HOPCROFT, *An n log n algorithm for minimizing states in a finite automaton*, Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, 1971, pp. 189–196.

[26] J. E. HOPCROFT AND J. K. WONG, *Linear time algorithm for isomorphism of planar graphs*, Extended abstract, Proc. 6th Annual ACM Symp. on Theory of Computing (1974), pp. 172–184.

[27] R. M. KARP, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

[28] ———, *The fast approximate solution of hard combinatorial problems*, Proc. 6th S-E Conf. Combinatorics, Graph Theory and Computing (1975), pp. 15–31.

[29] ———, *Probabilistic analysis of some combinatorial search algorithms*, Algorithms and Complexity, New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 1–19.

[30] F. KÁRTESZI, *Piani finite ciclici come risoluzioni di un certo problema de minimo*, Boll. Un. Mat. Ital. (3), 15 (1960), pp. 522–528.

[31] D. G. KIRKPATRICK AND D. G. CORNEIL, *Forest embeddings in regular graphs of large girth*, J. Combinatorial Theory Ser. B (1979), to appear.

[32] D. KOZEN, *Complexity of finitely presented algebras*, Proc. 9th Annual ACM Symp. on Theory of Computing (1977), pp. 164–177.

[33] E. LAWLER, *Graphical algorithms and their complexity*, Math. Centrum Tracts, 81 (1976), pp. 3–32.

[34] R. A. MATHON, *Sample graphs for isomorphism testing*, Proc. 9th S-E Conf. Combinatorics, Graph Theory and Computing (1978).

[35] ———, *A note on the graph isomorphism counting problem*, submitted.

[36] G. L. MILLER, *Graph isomorphism, general remarks*, Proc. 9th Annual ACM Symp. on Theory of Computing (1977), pp. 143–150, J. Comput. System Sci., to appear.

[37] L. PÓSA, *Hamiltonian circuits in random graphs*, Discrete Math., 14 (1976), pp. 359–364.

[38] V. R. PRATT, *Every prime has a succinct certificate*, this Journal, 4 (1975), pp. 214–220.

[39] R. C. READ AND D. G. CORNEIL, *The graph isomorphism disease*, J. Graph Theory, 1 (1977), pp. 339–363.

[40] H. SACHS, *Abzählung von Wäldern eines gegebenen Typs in regulären und biregulären Graphen I*, Publ. Math. Debrecen., 11 (1964), pp. 74–84.

[41] ———, *Abzählung von Wäldern eines gegebenen Typs in regulären und biregulären Graphen II*, Ibid., 12 (1965), pp. 7–24.

[42] D. C. SCHMIDT AND L. E. DRUFFEL, *A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices*, J. Assoc. Comput. Mach., 23 (1976), pp. 433–445.

[43] E. H. SUSSENGUTH JR., *A graph-theoretic algorithm for matching chemical structures*, J. Chem. Doc., 5 (1965), pp. 36–43.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0007 $01.00/0

OPTIMAL MERGING OF 3 ELEMENTS WITH n ELEMENTS*

F. K. HWANG†

Abstract. Suppose we are given two disjoint linearly-ordered subsets A_m and B_n of a linearly-ordered set S. The problem is to determine the linear ordering of their union (i.e., to "merge" A_m and B_n) by means of a sequence of pairwise comparisons between an element of A_m and an element of B_n. An algorithm to merge A_m and B_n is called M-optimal if it minimizes the maximum number of comparisons required where the maximum is taken over all possible ordering of $A_m \cup B_n$. Let $f_m(k)$ denote the largest n such that A_m and B_n can be merged in k comparisons by an M-optimal algorithm. The determination of $f_3(k)$ has been an open problem for a long time. In this paper we give a constructive proof that

$$f_3(1)=0,\quad f_3(2)=1,\quad f_3(3)=1,\quad f_3(4)=2,$$
$$f_3(5)=3,\quad f_3(6)=4,\quad f_3(7)=6,\quad f_3(8)=8,$$

and for $r \geqq 3$,

$$f_3(3r)=[\tfrac{43}{7}2^{r-2}]-2,$$
$$f_3(3r+1)=[\tfrac{107}{7}2^{r-3}]-2,$$
$$f_3(3r+2)=\left[\frac{17\cdot 2^r-6}{7}\right]-1.$$

where $[x]$ denotes the integral part of x.

1. Introduction. Suppose we are given two disjoint linearly-ordered subsets A_m and B_n of a linearly-ordered set S, say

$$A_m=\{a_1<a_2<\cdots<a_m\},$$
$$B_n=\{b_1<b_2<\cdots<b_n\}.$$

The problem is to determine the linear ordering of their union (i.e., to *merge* A_m and B_n) by means of a sequence of pairwise comparisons between an element of A_m and an element of B_n. Given any algorithm g to solve this problem, which we refer to as the (m, n) problem, let $K_g(m, n)$ denote the maximum of comparisons required considering all possible orderings of $A_m \cup B_n$. An algorithm g is said to be *M-optimal* if $K_g(m, n) = K(m, n)$, where $K(m, n) = \min_x K_x(m, n)$ where x ranges over all possible algorithms g.

The determination of $K(m, n)$ for general m and n is still an open problem. The several special classes of the (m, n) problem for which $K(m, n)$ are known are (see [1], [2], [3]):

(i) $m = 1$ and 2.

(ii) $m+4 \geqq n \geqq m$.

In particular, the $(3, n)$ problem has been mentioned as an open problem in [1] and [3]. The purpose of this paper is to settle the $(3, n)$ problem.

Let $f_m(k)$ denote the largest n such that the (m, n) problem can be done in k comparisons. Then for fixed m, $K(m, n)$ is known for every n if and only if $f_m(k)$ is known for every k. It has been determined in the literature [1], [2], [3] that

$$f_1(k)=2^k-1$$

and

$$f_2(k)=\begin{cases}[\tfrac{12}{7}2^{r-1}]-1 & \text{for } k=2r-1,\\ [\tfrac{17}{7}2^{r-1}]-1 & \text{for } k=2r.\end{cases}$$

where $[x]$ denotes the largest integer not exceeding x. In this paper, we give the values of

* Received by the editors March 29, 1978, and in revised form January 10, 1979.

† Bell Laboratories, Murray Hill, New Jersey 07974.

$f_3(k)$ as follows:

$$f_3(1)=0,\quad f_3(2)=1,\quad f_3(3)=1,\quad f_3(4)=2,$$
$$f_3(5)=3,\quad f_3(6)=4,\quad f_3(7)=6,\quad f_3(8)=8,$$

and for $r \geqq 3$,

$$f_3(3r)=[\tfrac{43}{7}2^{r-2}]-2,$$
$$f_3(3r+1)=[\tfrac{107}{7}2^{r-3}]-2,$$
$$f_3(3r+2)=\left[\frac{17\cdot 2^r-6}{7}\right]-1.$$

A table for $f_2(k)$ and $f_3(k)$ for $k \leqq 18$ is given in the following:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$f_2(k)$	0	1	2	3	5	8	12	18	26	37	53	76	108	154	218	309	437	620
$f_3(k)$	0	1	1	2	3	4	6	8	10	13	17	22	28	37	47	59	75	96

2. An algorithm for the $(3, f_3(k))$ problem. At a given stage of the merging process, a certain number of comparisons have been made and partial knowledge on the relations between a_i's and b_j's has been accumulated. We use a diagraph, called a *configuration*, to characterize the known relations at the current stage. In a configuration, each node represents an element and a link $x \to y$ indicates the relation that x is less than y. We also require that the configuration be *minimal* in the sense that no link can be omitted without reducing our knowledge of the relations. A configuration X is said to *dominate* a configuration Y if every relation in X is implied by a relation in Y. Note that X dominates Y implies that the number of comparisons required to do X cannot exceed the number required to do Y.

We present a merging algorithm in the format of a rooted binary tree where each node is associated with a configuration. The root of the tree is associated with the configuration before any comparison is made. Every internal node is also associated with a comparison and its two outlinks represent the two possible outcomes. The two outlinks lead to two nodes each associated with a new configuration absorbing the outcome of the comparison just made. A terminal node is usually associated with a configuration which can be readily analyzed. We indicate the number of comparisons required to merge an a_i (or a group of a_i) under that a_i (or that group) in the configuration and verify it in a later analysis. The sum of these indicated numbers plus the number of comparisons already taken to reach that configuration should not exceed k for an algorithm for the $(3, f_3(k))$ problem.

A few conventions are adopted to facilitate the presentation of a configuration. First of all, we replace the links between elements of $B_n(A_3)$ by undirected edges "—" since these elements are always ordered in a row from left to right and no confusion can arise. Secondly, as n can be very large, to save space, elements of B_n do not appear in the configuration unless they are linked to some element of A_3. Consequently, it is necessary to label the elements of B_n which do appear in the configuration. On the other hand, the elements of A_3 appear in every configuration (ordered in a row), hence need no labeling. Thirdly, the two nodes involved in the current comparison are denoted by solid circles while all other nodes are denoted by empty circles.

The values of $f_3(k)$ for $k \leqq 7$ are known (see [3]). It is also easy to verify that the (3, 8) problem can be done in eight comparisons and the (3, 10) problem in nine. For $k \geqq 10$, we present the algorithm in three parts depending on the residue classes of k mod 3 and using induction on k. We first study the case $k = 3r \geqq 12$. For $r = 4$, the algorithm is illustrated in Fig. 1.

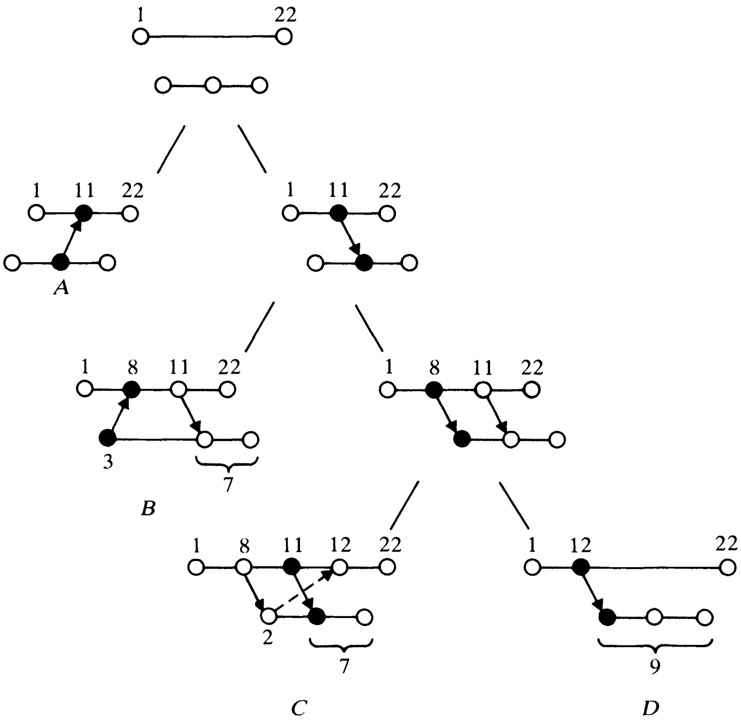


FIG. 1. $k=12$.

Analysis for the terminal nodes in Fig. 1. A. The merging problem is not affected if the direction of every link is reversed. Therefore, a configuration can be considered either in its own form or the reverse form. In that sense configuration A dominates the configuration to its right.

B, C, D. These are easily checked by using the definitions $f_1(k)$, $f_2(k)$ and $f_3(k)$. For $k=3r\geqq 15$, the algorithm is given in Fig. 2.

Definitions of symbols used in Fig. 2.

$$c_0=f_3(3r)=2^r+f_2(2r-3)+f_2(2r-7)+1=\lceil \tfrac{43}{7}2^{r-2}\rceil-2,$$

$$c_1=\left\lceil\frac{f_3(3r)+1}{2}\right\rceil=\left\lceil\frac{43\cdot 2^{r-3}-4}{7}\right\rceil,$$

$$c_2=2^{r-1}+2^{r-2}+f_3(3r-4)+1=\left\lceil\frac{38\cdot 2^{r-2}-6}{7}\right\rceil,$$

$$c_3=f_3(3r)-2^{r-4}+1=\lceil \tfrac{165}{7}2^{r-4}\rceil,$$

$$c_4=2^{r-1}+2^{r-2}+f_2(2r-3)+1=\lceil \tfrac{33}{7}2^{r-2}\rceil,$$

$$c_5=c_4+2^{r-3}=\lceil \tfrac{73}{7}2^{r-3}\rceil,$$

$$c_6=c_5+2^{r-4}=\lceil \tfrac{153}{7}2^{r-4}\rceil,$$

$$c_3'=c_5'=2^{r-1},$$

$$c_4'=c_6'=2^{r-1}+2^{r\cdot 2}=3\cdot 2^{r-2}.$$

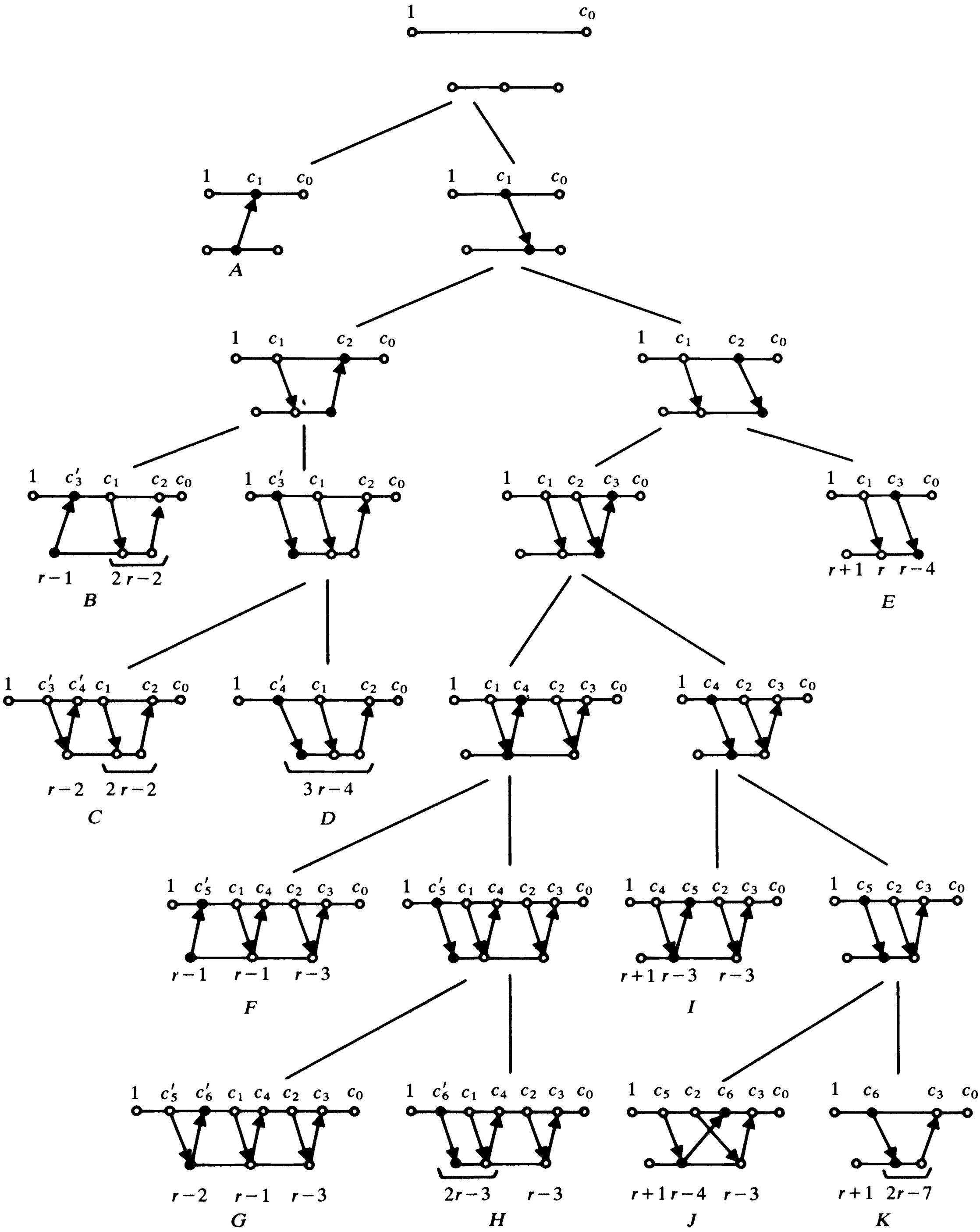

1 c_0
1 c_1 c_0
A
1 c_1 c_0
1 c_1 c_2 c_0
1 c_1 c_2 c_0
1 c_3' c_1 c_2 c_0
$r-1$ $2r-2$
B
1 c_3' c_1 c_2 c_0
1 c_1 c_2 c_3 c_0
1 c_1 c_3 c_0
$r+1$ r $r-4$
E
1 c_3' c_4' c_1 c_2 c_0
$r-2$ $2r-2$
C
1 c_4' c_1 c_2 c_0
$3r-4$
D
1 c_1 c_4 c_2 c_3 c_0
1 c_4 c_2 c_3 c_0
1 c_5' c_1 c_4 c_2 c_3 c_0
$r-1$ $r-1$ $r-3$
F
1 c_5' c_1 c_4 c_2 c_3 c_0
1 c_4 c_5 c_2 c_3 c_0
$r+1$ $r-3$ $r-3$
I
1 c_5 c_2 c_3 c_0
1 c_5' c_6' c_1 c_4 c_2 c_3 c_0
$r-2$ $r-1$ $r-3$
G
1 c_6' c_1 c_4 c_2 c_3 c_0
$2r-3$ $r-3$
H
1 c_5 c_2 c_6 c_3 c_0
$r+1$ $r-4$ $r-3$
J
1 c_6 c_3 c_0
$r+1$ $2r-7$
K

FIG. 2. $k=3r\geqq 15$.

It is clear that

$$c_0 \geqq c_3 \geqq c_6 \geqq c_2 \geqq c_5 \geqq c_4 \geqq c_1 \geqq c_4' \geqq c_3'.$$

Analysis for the terminal nodes in Fig. 2.

A. Configuration A dominates the configuration to its right.
B. a_1: $c_3' - 1 = f_1(r-1)$. a_2 and a_3: $c_2 - c_1 - 1 \leqq [\frac{33}{7}2^{r-3}] \leqq [\frac{17}{7}2^{r-2}] - 1 = f_2(2r-2)$.
C. a_1: $c_4' - c_3' - 1 = f_1(r-2)$.

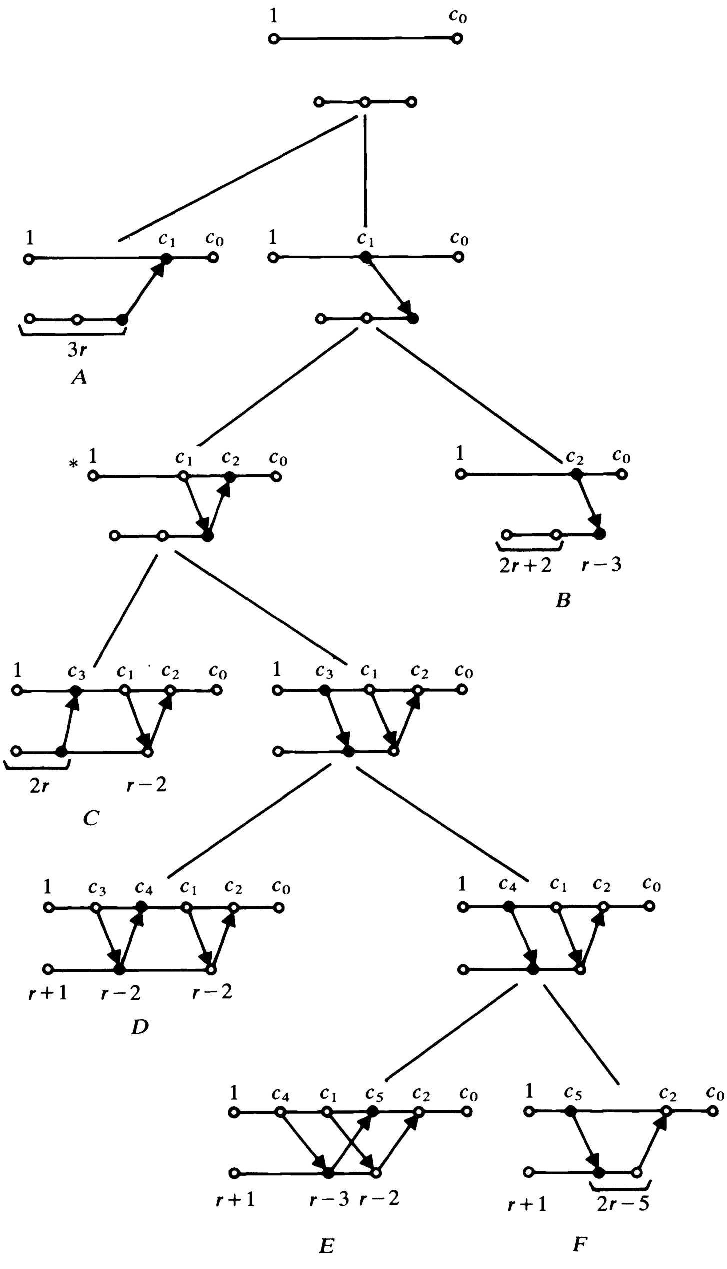


FIG. 3. $k = 3r+1 \geqq 10$.

D. Configuration D certainly dominates the configuration D′ obtained from D by deleting the link $c_1 \rightarrow a_2$. Since $c_2 - c_4' - 1 = f_3(3r-4)$, D′, hence D, can be done in $3r-4$ comparisons.

E. a_3: $c_0 - c_3 = f_1(r-4)$. a_2: $c_0 - c_1 \leqq [(43 \cdot 2^{r-3} - 4)/7] \leqq 2^r - 1 = f_1(r)$. a_1: $c_0 = [\frac{43}{7}2^{r-2}] - 2 \leqq 2^{r+1} - 1 = f_1(r+1)$.

F. a_3: $c_3 - c_2 - 1 \leqq [\frac{13}{7}2^{r-4}] \leqq 2^{r-3} - 1 = f_1(r-3)$. a_2: $c_4 - c_1 \leqq [\frac{23}{7}2^{r-3}] \leqq 2^{r-1} - 1 = f_1(r-1)$. a_1: $c_5' - 1 = f_1(r-1)$.

G. This is similar to F.

H. Ignoring the link $c_1 \rightarrow a_2$, then since $c_4 - c_6' - 1 = f_2(2r-3)$, a_1 and a_2 together can be merged in $2r-3$ comparisons.

I. a_2: $c_5 - c_4 - 1 = f_1(r-3)$.

J. a_2: $c_6 - c_5 - 1 = f_1(r-4)$.

K. a_2 and a_3: $c_3 - c_6 - 1 = f_3(3r) - 2^{r-4} - (2^{r-1} + 2^{r-2} + f_2(2r-3) + 1 + 2^{r-3} + 2^{r-4}) = f_2(2r-7)$.

Next we study the case $k = 3r+1 \geqq 10$. The algorithm is given in Fig. 3.

Definitions of symbols used in Fig. 3.

$$
\begin{aligned}
&c_0 = f_3(3r+1) = 2^{r-2} + 2^{r-3} + f_3(3r) = [\tfrac{107}{7}2^{r-3}] - 2,\\
&c_1 = f_3(3r) + 1 = [\tfrac{43}{7}2^{r-2}] - 1,\\
&c_2 = c_1 + 2^{r-2} = [\tfrac{50}{7}2^{r-2}] - 1,\\
&c_3 = f_2(2r) + 1 = [\tfrac{17}{7}2^{r-1}],\\
&c_4 = c_3 + 2^{r-2} = [\tfrac{41}{7}2^{r-2}],\\
&c_5 = c_4 + 2^{r-3} = [\tfrac{89}{7}2^{r-3}].
\end{aligned}
$$

It is clear that

$$c_0 \geqq c_2 \geqq c_5 \geqq c_1 \geqq c_4 \geqq c_3.$$

Analysis for the terminal nodes in Fig. 3.

A. a_1, a_2 and a_3: $c_1 - 1 = f_3(3r)$.

B. a_3: $c_0 - c_2 = f_1(r-3)$. a_1 and a_2: $c_0 = [\frac{107}{7}2^{r-3}] - 2 \leqq [\frac{17}{7}2^r] - 1 = f_2(2r+2)$.

C. a_3: $c_2 - c_1 - 1 = f_1(r-2)$. a_1 and a_2: $c_3 - 1 = f_2(2r)$.

D. a_2: $c_4 - c_3 - 1 = f_1(r-2)$. a_1: $c_4 - 1 = [\frac{41}{7}2^{r-2}] - 1 \leqq 2^{r+1} - 1 = f_1(r+1)$.

E. a_2: $c_5 - c_4 - 1 = f_1(r-3)$. a_1: $c_5 - 1 = [\frac{89}{7}2^{r-3}] - 1 \leqq 2^{r+1} - 1 = f_1(r+1)$.

F. a_2 and a_3: $c_2 - c_5 - 1 \leqq [\frac{11}{7}2^{r-3}] - 1 \leqq [\frac{12}{7}2^{r-3}] - 1 = f_2(2r-5)$. a_1: $c_2 - 1 = [\frac{50}{7}2^{r-2}] - 2 \leqq 2^{r+1} - 1 = f_1(r+1)$.

Note that comparison 3, or comparisons 3 and 4 together, can be taken before comparison 2 without affecting the results.

Finally, we study the case $k = 3r+2 \geqq 11$. The algorithm is given in Fig. 4.

Definitions of symbols used in Fig. 4.

$$
\begin{aligned}
&c_0 = f_3(3r+2) = 2f_2(2r) + 1 = \left[\frac{17 \cdot 2^r - 6}{7}\right] - 1,\\
&c_1 = \left[\frac{c_0 + 1}{2}\right] = \left[\frac{17}{7}2^{r-1}\right],\\
&c_2 = 2^r,\\
&c_3 = c_2 + 2^{r-1} = 3 \cdot 2^{r-1}.
\end{aligned}
$$

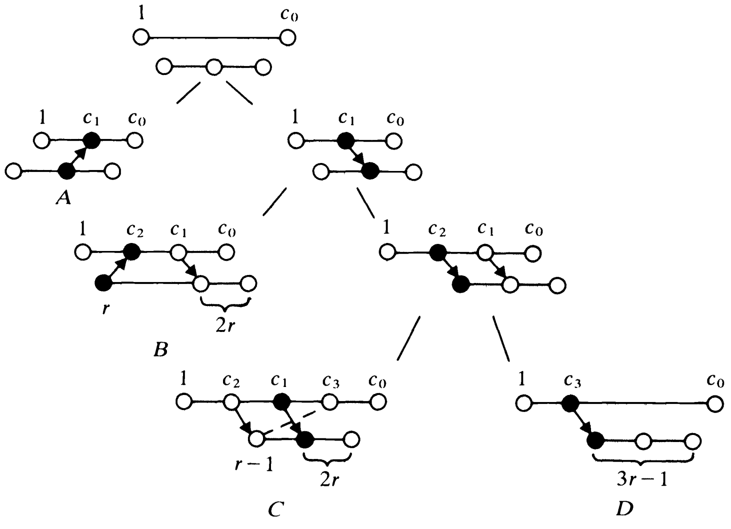


FIG. 4. $k = 3r+2 \geqq 11$.

Analysis for the terminal nodes in Fig. 4.

A. Configuration A is symmetric to the configuration to its right.

B. a_1: $c_2 - 1 = 2^r - 1 = f_1(r)$. a_2 and a_3: $c_0 - c_1 = f_2(2r)$.

C. a_1: $c_3 - c_2 - 1 = 2^{r-1} - 1 = f_1(r-1)$.

D. a_1, a_2 and a_3: $c_0 - c_3 = [(13 \cdot 2^{r-1} - 6)/7] - 1 \leqq [(17 \cdot 2^{r-1} - 6)/7] - 1 = f_3(3r-1)$.

3. Optimality of the algorithm. To prove the optimality of the algorithm, we have to show that given a $(3, f_3(k)+1)$ problem, then regardless of which comparison a_i vs. b_j we start with, the problem cannot be done in k comparisons. Since there are too many choices of a_i vs. b_j to be dealt with, we use the following scheme to simplify things. We first classify all the possible choices into three categories depending on which a_i, $i = 1, 2, 3$, the comparison involves. Then we deal with each category by using the following lemma.

LEMMA 0. *Suppose we can show for fixed a_i, neither of the following two cases can be done in k comparisons*: (i) *The first comparison is a_i vs. b_x and results in $a_i \to b_x$.*

(ii) *The first comparison is a_i vs. b_{x-1} and results in $a_i \leftarrow b_{x-1}$.*

Then any algorithm whose first comparison involves a_i cannot be done in k comparisons.

Proof. Suppose an algorithm starts with the first comparison a_i vs. b_y. If $y \geqq x$, consider the outcome that $a_i \to b_y$. Then the resultant configuration is dominated by the configuration stated in (i) of Lemma 0. If $y \leqq x-1$, consider the outcome that $a_i \leftarrow b_y$. Then the resultant configuration is dominated by the configuration stated in (ii) of Lemma 0. In either case, at least one outcome will result in more than k comparisons, hence the Lemma.

The proof of the optimality of the algorithm will again be illustrated as a rooted tree (but no longer a binary tree). There will be two types of nodes in the tree: a *configuration node* shows the configuration after the current comparison, and an *index node* indexes the history of comparisons before a configuration node is reached. The index used is the ordered sequence of a_i's the comparisons have been involved including the current one. We also indicate the outcomes of the previous comparisons by using $\bar{a}_i$ ($\underline{a}_i$) to denote the outcome that $a_i \leftarrow b_x$ ($a_i \to b_x$) for some b_x. Thus the tree starts with a configuration node showing the configuration before any comparison is made. Then it branches into three index nodes labeled by a_1, a_2 and a_3 respectively. Each index node, say, the one labeled by a_i, now branches into two configuration nodes showing the configurations obtained after the comparison $a_i \leftarrow b_x$ and the comparison $a_i \rightarrow b_{x-1}$ respectively. Each configuration node then branches into three more index nodes and the branching process goes on. If the configuration associated with a configuration node can be readily analyzed, that node becomes a terminal node with no more branching from it. Again we indicate the least numbers of comparisons needed to merge such a_i or a group of a_i's under the a_i's. The sum of these numbers plus the number of comparisons already taken to reach that configuration should exceed k in our proof. The following lemmas are crucial for our analysis.

LEMMA 1. *Suppose that a configuration contains the two links $a_i \rightarrow b_x$ and $a_{i+1} \leftarrow b_y$ for some i and $y \geqq x-1$. Then the configuration can be replaced by two subconfigurations, one induced by the nodes $(a_1, \cdots, a_i)$ and $(b_1, b_2, \cdots, b_{x-1})$, and the other by the nodes $(a_i, \cdots, a_3)$ and $(b_y, b_{y+1}, \cdots)$.*

Proof. The decomposition is possible since the two subproblems are completely independent.

Sometimes Lemma 1 does not apply directly to a configuration unless an additional link is added. We use a broken link to indicate that the configuration we wish to consider is the one when the broken link is added (and any link now carrying redundant information should be removed). Since it is obvious that the new configuration dominates the original one, it suffices to show that the new configuration cannot be done in a designated number of comparisons.

For many terminal nodes, an application of Lemma 1 to the configurations in there splits the configurations into several subconfigurations each of which can be considered as the starting configuration of a $(1, t)$ problem or a $(2, t)$ problem for some number t. Therefore we need only use the definitions of $f_1(t)$ and $f_2(t)$ to determine how many comparisons are needed for these subconfigurations. As t, $f_1(t)$ and $f_2(t)$ are all explicit, checking is straightforward and can be verified by the reader. We will give analysis only for those subconfigurations whose numbers of comparisons cannot be determined by just knowing $f_1(t)$ and $f_2(t)$. To be more specific, if $N(x, y)$ denotes the number of comparisons required to do the configuration of the following type, then our analysis usually is to determine $N(x, y)$ for some given x and y. The following two lemmas play a crucial role in that kind of analysis.

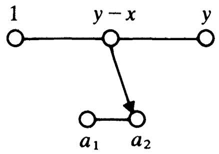


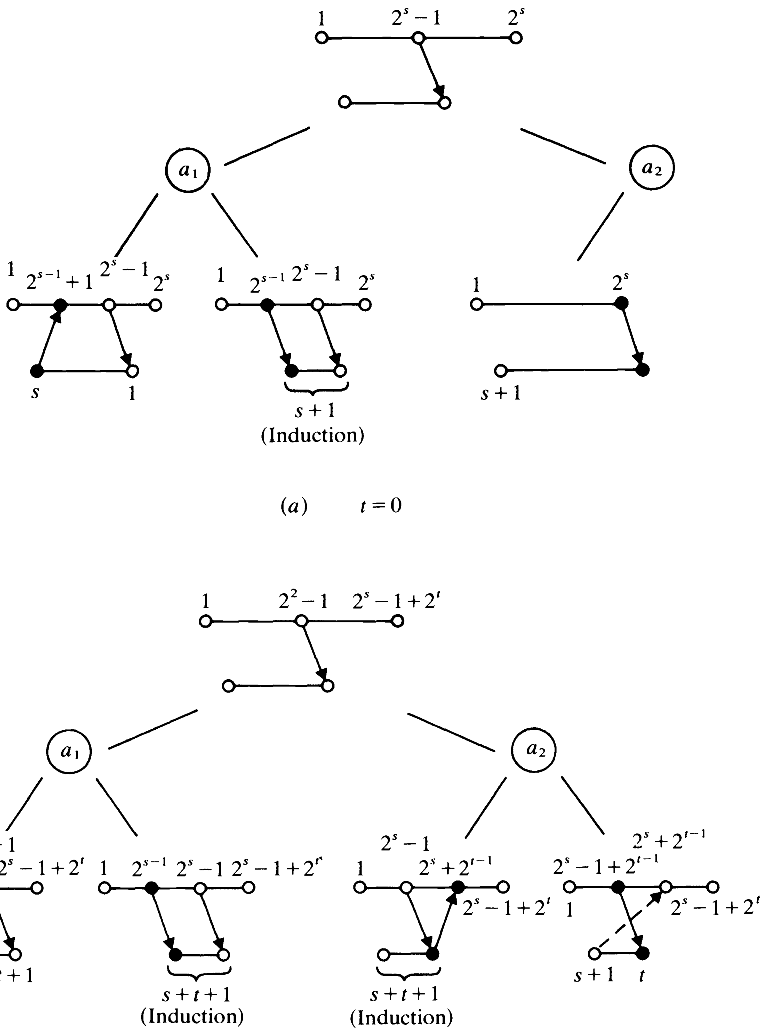
FIG. 5. *An $N(x, y)$ configuration.*

LEMMA 2. $N(2^t, 2^s-1+2^t) > s+t+1$ *for* $s, t \geqq 0$.

Proof. Suppose $s=0$. Then $N(2^t, 2^t)$ is the number of comparisons required for the $(2, 2^t)$ problem. Since $f_2(t+1) < 2^t$, Lemma 2 follows immediately. Therefore we assume $s>0$. The proof of Lemma 2 is given in Fig. 6(a) and 6(b) for the two cases $t=0$ and $t>0$ respectively.

LEMMA 3. *For* $k \geqq t+q+1$ *and* $t-2 \geqq q \geqq 0$, *define*

$$g(k, t, q) = 2^{t-2} + 2^{t-3} + \cdots + 2^{q+1} + 2^{k-t-1} + 2^{k-t-2} + \cdots + 2^{q+1} + f_2(2q+3) + 1.$$



(*a*) $t=0$

(*b*) $t>0$

FIG. 6. *Proof for Lemma* 2.

Then

$$N(x, y) > k \quad \textit{if} \quad x \geqq 2^{t-1}+2^q \quad \textit{and} \quad y \geqq g(k, t, q).$$

Proof. Since the configuration corresponding to $N(2^{t-1}+2^q, g(k, t, q))$ dominates the configuration corresponding to $N(x, y)$, it suffices to prove

$$N(2^{t-1}+2^q, g(k, t, q)) > k.$$

The proof is given by the tree in Fig. 7.

Definitions of symbols used in Fig. 7

$$c_0 = g(k, t, q),$$
$$c_1 = g(k, t, q) - 2^{t-1} - 2^q,$$
$$c_2 = 2^{k-t-1}, \qquad c_3 = c_0 - 2^{t-2}.$$

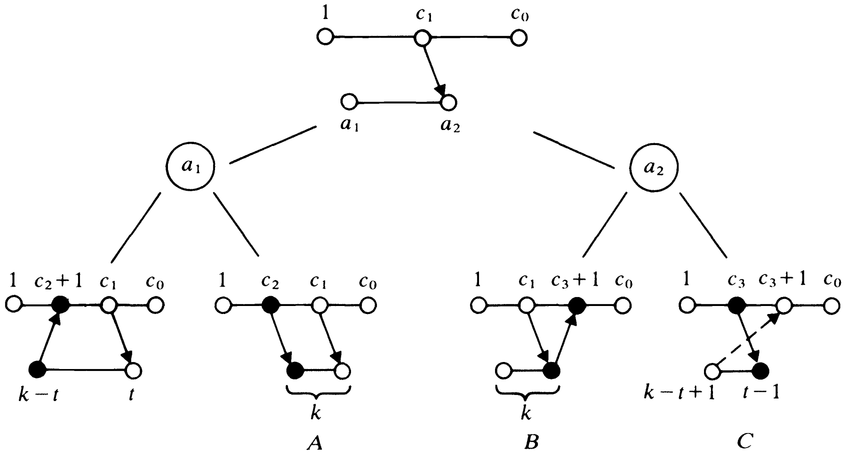


FIG. 7. *Proof for Lemma* 3.

Analysis for the labeled terminal nodes:

A. Since

$$c_0-c_2=g(k-1,t,q)$$

and

$$c_0-c_1=2^{t-1}+2^q.$$

Configuration A needs at least k comparisons by induction.

B. For $q<t-2$,

$$c_3=g(k,t,q)-2^{t-2}=g(k-1,t-1,q)$$

and

$$c_3-c_1=2^{t-2}+2^q.$$

Configuration B needs at least k comparisons by induction.

For $q=t-2$, consider the outcome $a_1<c_1+1$. Then a_2 needs t comparisons and a_1 needs $k-t$ comparisons to merge by using Lemma 1 and the fact

$$\begin{aligned}c_1&=2^{k-t-1}+2^{k-t-2}+\cdots+2^{t-1}+f_2(2t-1)+1-2^{t-1}-2^{t-2}\\&=2^{k-t-1}+2^{k-t-2}+\cdots+2^{t-1}+[\tfrac{3}{7}2^{t-2}]\geqq f_1(k-t-1)+1.\end{aligned}$$

C. For $q\leqq k-t-2$,

$$\begin{aligned}c_3&=g(k,t,q)-2^{t-2}\\&\geqq 2^{k-t-1}+2^{k-t-2}+\cdots+2^{q+1}+f_2(2q+3)+1-2^q\\&=2^{k-t}+[\tfrac{3}{7}2^q]\geqq f_1(k-t)+1.\end{aligned}$$

For $q\geqq k-t-1$,

$$\begin{aligned}c_3&=g(k,t,q)-2^{t-2}\\&\geqq f_2(2q+3)+1-2^q\\&=[\tfrac{12}{7}2^{q+1}]-2^q\geqq 2^{q+1}\geqq f_1(k-t)+1.\end{aligned}$$

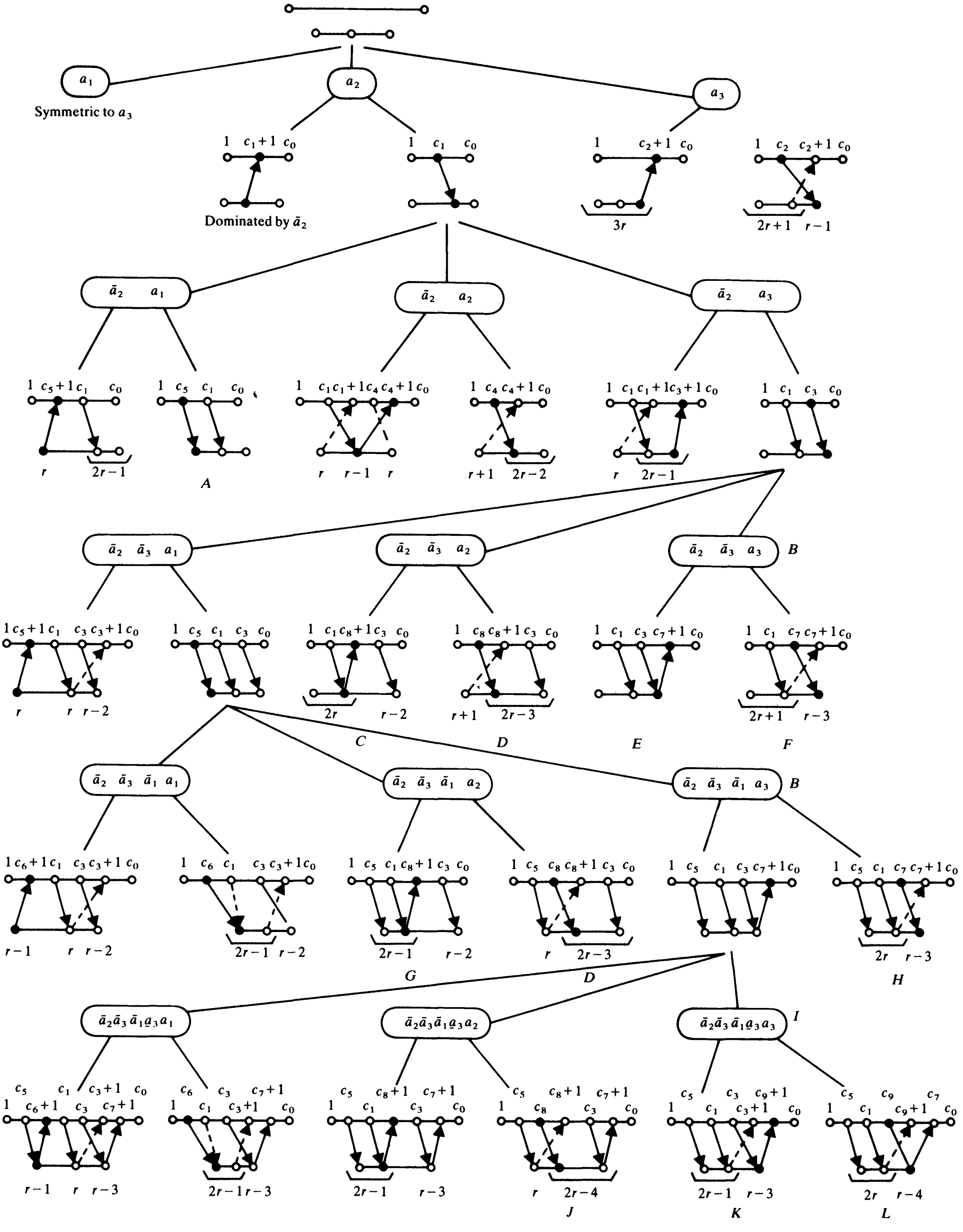

a_1
Symmetric to a_3
a_2
a_3
1 c_1+1 c_0
Dominated by $\bar{a}_2$
1 c_1 c_0
1 c_2+1 c_0
$3r$
1 c_2 c_2+1 c_0
$2r+1$ $r-1$
$\bar{a}_2$ a_1
$\bar{a}_2$ a_2
$\bar{a}_2$ a_3
1 c_5+1 c_1 c_0
r $2r-1$
1 c_5 c_1 c_0
A
1 c_1 c_1+1 c_4 c_4+1 c_0
r $r-1$ r
1 c_4 c_4+1 c_0
$r+1$ $2r-2$
1 c_1 c_1+1 c_3+1 c_0
r $2r-1$
1 c_1 c_3 c_0
$\bar{a}_2$ $\bar{a}_3$ a_1
$\bar{a}_2$ $\bar{a}_3$ a_2
$\bar{a}_2$ $\bar{a}_3$ a_3
B
1 c_5+1 c_1 c_3 c_3+1 c_0
r r $r-2$
1 c_5 c_1 c_3 c_0
1 c_1 c_8+1 c_3 c_0
$2r$ $r-2$
C
1 c_8 c_8+1 c_3 c_0
$r+1$ $2r-3$
D
1 c_1 c_3 c_7+1 c_0
E
1 c_1 c_7 c_7+1 c_0
$2r+1$ $r-3$
F
$\bar{a}_2$ $\bar{a}_3$ $\bar{a}_1$ a_1
$\bar{a}_2$ $\bar{a}_3$ $\bar{a}_1$ a_2
$\bar{a}_2$ $\bar{a}_3$ $\bar{a}_1$ a_3
B
1 c_6+1 c_1 c_3 c_3+1 c_0
$r-1$ r $r-2$
1 c_6 c_1 c_3 c_3+1 c_0
$2r-1$ $r-2$
1 c_5 c_1 c_8+1 c_3 c_0
$2r-1$ $r-2$
G
1 c_5 c_8 c_8+1 c_3 c_0
r $2r-3$
D
1 c_5 c_1 c_3 c_7+1 c_0
1 c_5 c_1 c_7 c_7+1 c_0
$2r$ $r-3$
H
$\bar{a}_2\bar{a}_3\bar{a}_1\underline{a}_3a_1$
$\bar{a}_2\bar{a}_3\bar{a}_1\underline{a}_3a_2$
$\bar{a}_2\bar{a}_3\bar{a}_1\underline{a}_3a_3$
I
c_5 c_1 c_3+1 c_0
1 c_6+1 c_3 c_7+1
$r-1$ r $r-3$
c_6 c_3 c_7+1
1 c_1 c_3+1 c_0
$2r-1$ $r-3$
c_5 c_8+1 c_7+1
1 c_1 c_3 c_0
$2r-1$ $r-3$
c_5 c_8+1 c_7+1
1 c_8 c_3 c_0
r $2r-4$
J
c_5 c_3 c_9+1
1 c_1 c_3+1 c_0
$2r-1$ $r-3$
K
c_5 c_9 c_7
1 c_1 c_9+1 c_0
$2r$ $r-4$
L

FIG. 8. *Proof for* $k=3r\geqq 9$.

We are now ready to prove the optimality of the algorithm. The optimality of the algorithm for $k \leqq 7$ is well documented in [3]. Therefore, we will only be concerned with the case $k \geqq 8$. Again, the proof is separated into three parts depending on the residue classes of k mod 3. We also define $f_1(r) = -1$ for negative r to avoid separate discussions for certain special cases. We first study the case $K = 3r \geqq 9$. The proof is illustrated in Fig. 8.

Definitions of symbols used in Fig. 8.

$$c_0 = f_3(3r)+1 = \lfloor\tfrac{43}{7}2^{r-2}\rceil - 1,$$

$$c_1 = \left[\frac{c_0-1}{2}\right] = \lfloor\tfrac{43}{7}2^{r-3}\rceil,$$

$$c_2 = c_0 - 2^{r-2} = \lfloor\tfrac{36}{7}2^{r-2}\rceil - 1,$$

$$c_3 = c_1 + f_2(2r-2)+1 = 11 \cdot 2^{r-3} - 1,$$

$$c_4 = c_1 + 2^{r-2} = \lfloor\tfrac{57}{7}2^{r-3}\rceil,$$

$$c_5 = 2^{r-1},$$

$$c_6 = c_5 + 2^{r-2} = 3 \cdot 2^{r-2},$$

$$c_7 = c_0 - f_1(r-4) - 1 = \begin{cases} c_0 & \text{for } r = 3, \\ \lfloor\tfrac{165}{7}2^{r-4}\rceil - 1 & \text{for } r \geqq 4, \end{cases}$$

$$c_8 = 2^{r-1} + 2^{r-2} + f_2(2r-3)+1 = \lfloor\tfrac{33}{7}2^{r-2}\rceil,$$

$$c_9 = c_7 - f_1(r-5) - 1 = \begin{cases} c_7 & \text{for } r = 3, 4, \\ \lfloor\tfrac{323}{7}2^{r-5}\rceil - 1 & \text{for } r \geqq 5. \end{cases}$$

Analysis for the labeled nodes in Fig. 8. (It is understood that Lemma 3 is to be applied in each case and we just verify the conditions of Lemma 3 in the following.)

A. If the next comparison involves a_3, future comparisons are same as (except a comparison of a_1 with c_5 or c_{5+1} should be ignored) and analysis is analogous to the $\bar{a}_2\bar{a}_3$ case. If the next comparison involves a_2, comparison is same as and analysis is analogous to the $\bar{a}_2a_2$ case. If the next comparison involves a_1, we consider the two cases $a_1 < c_6 + 1$ and $a_1 > c_6$. In the former case, a_1 needs $r-1$ more comparisons while a_2 and a_3 together needs $2r-1$ comparisons. In the latter case, assume $a_1 > c_1$. Then the problem is reduced to a $(3, c_0 - c_1) = (3, \lfloor\tfrac{43}{7}2^{r-3}\rceil - 1)$ which needs $3r-2$ more comparisons since $f_3(3r-3) = \lfloor\tfrac{43}{7}2^{r-3}\rceil - 2$ by induction.

B. For $r = 3$, $c_9 = c_0$. Therefore the cases $\bar{a}_2\bar{a}_3\underline{a}_3$ and $\bar{a}_2\bar{a}_3\bar{a}_1\underline{a}_3$ cannot occur.

C. $c_8 - c_1 = \lfloor\tfrac{23}{7}2^{r-3}\rceil \geqq 2^{r-2} + 2^{r-3}$, $c_8 = \lfloor\tfrac{33}{7}2^{r-2}\rceil = g(2r-1, r-1, r-3)$.

D. Trivially true for $r = 3, 4$. For $r \geqq 5$, $c_0 - c_3 = \lfloor\tfrac{9}{7}2^{r-3}\rceil \geqq 2^{r-3} + 2^{r-5}$, $c_0 - c_8 = \lfloor\tfrac{10}{7}2^{r-2}\rceil = g(2r-4, r-2, r-5)$.

E. If the next comparison involves a_3, consider the two cases $a_3 < c_9 + 1$ and $a_3 > c_9$. Analysis is analogous to J and K. If the next comparison involves a_2, comparison is same as and analysis is analogous to the case $\bar{a}_2\bar{a}_3a_2$. If the next comparison involves a_1, future comparisons are same as (except a comparison of a_3 with c_7 or $c_7 + 1$ should be ignored) and analysis is analogous to the $\bar{a}_2\bar{a}_3a_1$ case.

F. $c_7 - c_1 \geqq 2^{r-1} + 2^{r-3}$, $c_7 \geqq \lfloor\tfrac{5}{7}2^{r+1}\rceil = g(2r, r, r-3)$.

G. $c_8 - c_1 = \lfloor\tfrac{23}{7}2^{r-3}\rceil \geqq 2^{r-2} + 2^{r-3}$, $c_8 - c_5 = c_8 - 2^{r-1} = g(2r-1, r-1, r-3) - 2^{r-1} = g(2r-2, r-1, r-3)$.

H. Since $c_7 \geqq c_9$, H needs at least as many comparisons as L.

I. For $r=4$, $c_9=c_0$. Therefore $\bar{a}_2\bar{a}_3\bar{a}_1\underline{a}_3\underline{a}_3$ cannot occur.

J. $c_7-c_3 \geqq 2^{r-4}+2^{r-5}$, $c_7-c_8=c_0-c_8-2^{r-4}=g(2r-4, r-2, r-5)-2^{r-4}=g(2r-5, r-3, r-5)$.

K. Since $c_3 \geqq c_8$, K needs at least as many comparisons as G.

L. For $r \geqq 4$, $c_9-c_1 \geqq 2^{r-1}+2^{r-3}$, $c_9 \geqq [\frac{5}{7}2^{r+1}]=g(2r, r, r-3)$.

Next we study the case $k=3r+1 \geqq 10$. The proof is illustrated in Fig. 9.

Definitions of symbols used in Fig. 9.

$$c_0=f_3(3r+1)+1=[\tfrac{107}{7}2^{r-3}]-1,$$

$$c_1=c_0-2^{r-3}=[\tfrac{25}{7}2^{r-1}]-1,$$

$$c_2=c_1-f_1(r-4)-1=\begin{cases} c_1 & \text{for } r=3, \\ [\tfrac{193}{7}2^{r-4}]-1 & \text{for } r \geqq 4, \end{cases}$$

$$c_3=c_1-2^{r-3}=[\tfrac{93}{7}2^{r-3}]-1,$$

$$c_4=c_5-f_1(r-4)-1=\begin{cases} c_5 & \text{for } r=3, \\ [\tfrac{179}{7}2^{r-4}]-1 & \text{for } r \geqq 4, \end{cases}$$

$$c_5=c_3-2^{r-3}=[\tfrac{43}{7}2^{r-2}]-1,$$

$$d_1=2^{r-1},$$

$$d_2=d_1+2^{r-3}=5 \cdot 2^{r-3},$$

$$d_3=d_2+f_1(r-4)+1=\begin{cases} d_2 & \text{for } r=3, \\ 11 \cdot 2^{r-4} & \text{for } r \geqq 4, \end{cases}$$

$$d_4=d_3+f_1(r-5)+1 \doteq \begin{cases} d_3 & \text{for } r=3, 4, \\ 23 \cdot 2^{r-5} & \text{for } r \geqq 5, \end{cases}$$

$$d_5=d_1+f_2(2r-4)+1=[\tfrac{45}{7}2^{r-3}],$$

$$d_6=d_1+2^{r-1}=2^r,$$

$$d_7=d_6+2^{r-2}=5 \cdot 2^{r-2},$$

$$d_8=d_7+2^{r-3}=11 \cdot 2^{r-3},$$

$$d_9=d_8+f_1(r-4)+1=\begin{cases} d_8 & \text{for } r=3, \\ 23 \cdot 2^{r-4} & \text{for } r \geqq 4, \end{cases}$$

$$e_1=2^r(=d_6),$$

$$e_2=f_2(2r)+1=[\tfrac{17}{7}2^{r-1}],$$

$$e_3=c_0-2^{r-2}-2^{r-3}-f_2(2r-4)-1=[\tfrac{69}{7}2^{r-3}],$$

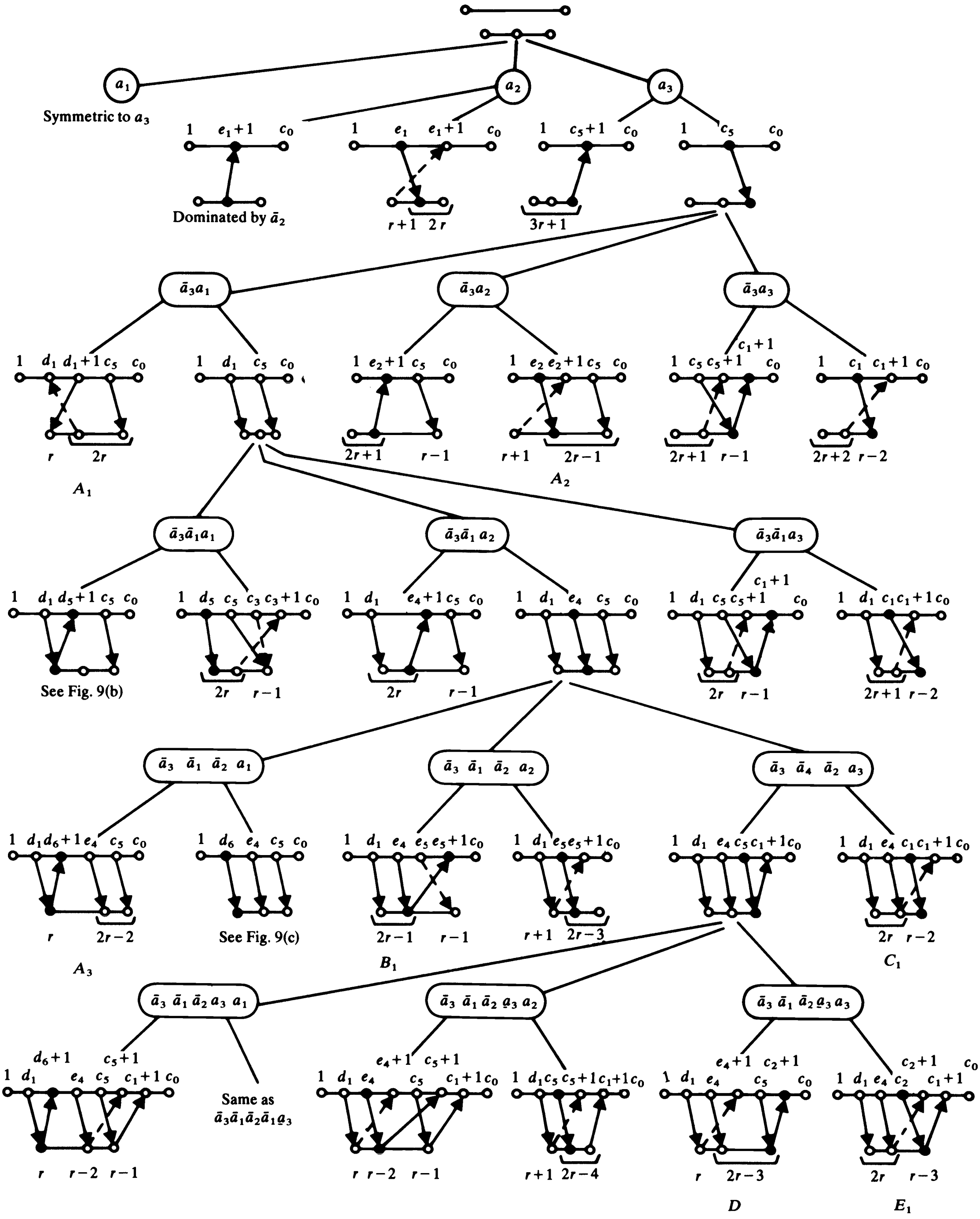
$$e_4=d_1+f_2(2r-1)=[\tfrac{19}{7}2^{r-1}],$$

$$e_5=c_1-f_2(2r-5)-1=[\tfrac{25}{7}2^{r-1}]-[\tfrac{3}{7}2^{r-1}]-1,$$

$$e_6=c_0-f_2(2r-4)-1=[\tfrac{45}{7}2^{r-2}].$$

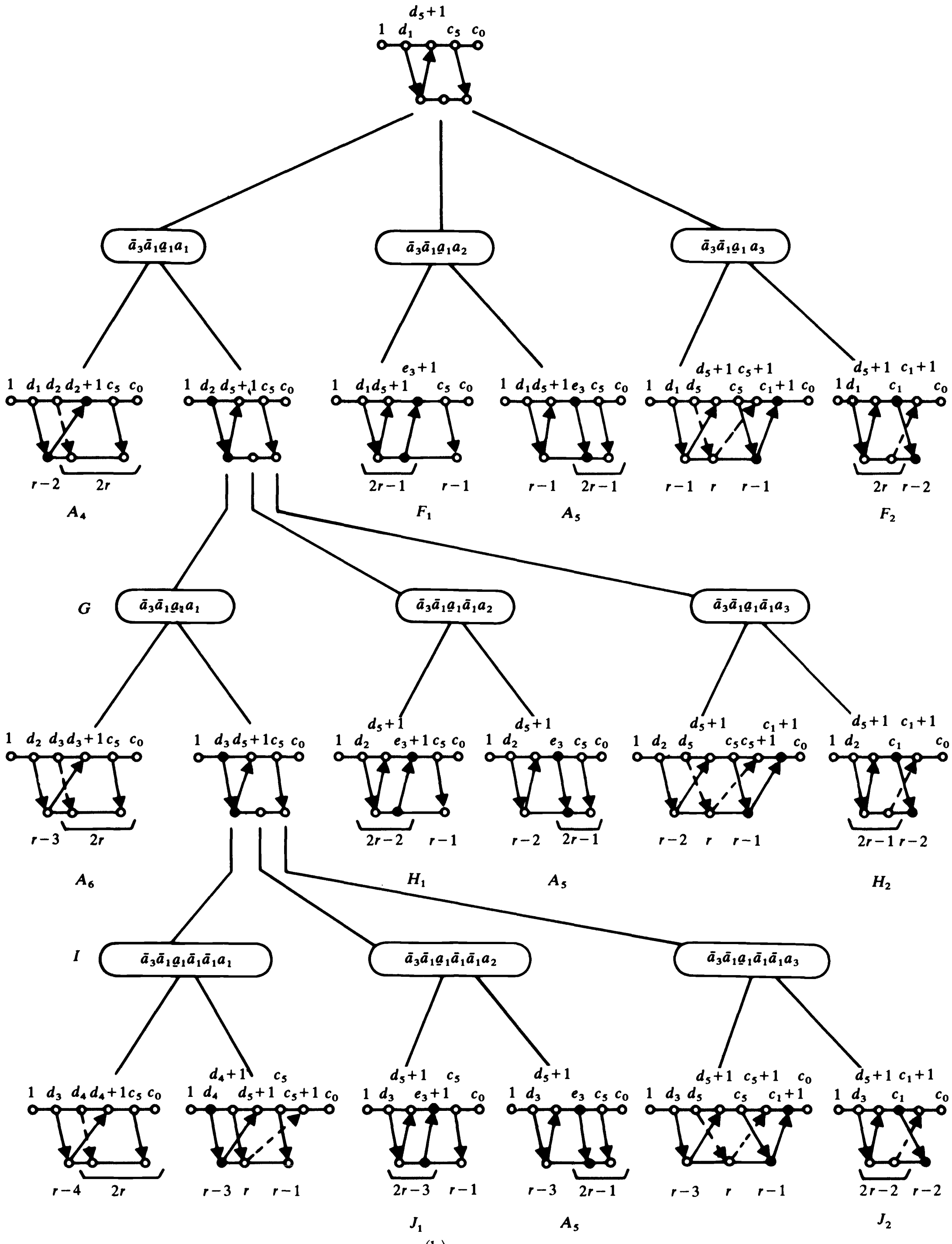
It is clear that

$$\begin{aligned} c_0 &\geqq c_1 \geqq c_2 \geqq c_3 \geqq e_6 \geqq c_4 \geqq e_5 \geqq c_5 \geqq d_9 \geqq e_4 \geqq e_3 \geqq e_2 \\ &\geqq d_8 \geqq d_7 \geqq d_6 = e_1 \geqq d_5 \geqq d_4 \geqq d_3 \geqq d_2 \geqq d_1. \end{aligned}$$


a_1
a_2
a_3
Symmetric to a_3
1 e_1+1 c_0
Dominated by $\bar{a}_2$
1 e_1 e_1+1 c_0
$r+1$ $2r$
1 c_5+1 c_0
$3r+1$
1 c_5 c_0
$\bar{a}_3a_1$
$\bar{a}_3a_2$
$\bar{a}_3a_3$
1 d_1 d_1+1 c_5 c_0
r $2r$
A_1
1 d_1 c_5 c_0
1 e_2+1 c_5 c_0
$2r+1$ $r-1$
1 e_2 e_2+1 c_5 c_0
$r+1$ $2r-1$
A_2
c_1+1
1 c_5 c_5+1 c_0
$2r+1$ $r-1$
1 c_1 c_1+1 c_0
$2r+2$ $r-2$
$\bar{a}_3\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\,a_2$
$\bar{a}_3\bar{a}_1a_3$
1 d_1 d_5+1 c_5 c_0
See Fig. 9(b)
1 d_5 c_5 c_3 c_3+1 c_0
$2r$ $r-1$
1 d_1 e_4+1 c_5 c_0
$2r$ $r-1$
1 d_1 e_4 c_5 c_0
c_1+1
1 d_1 c_5 c_5+1 c_0
$2r$ $r-1$
1 d_1 c_1 c_1+1 c_0
$2r+1$ $r-2$
$\bar{a}_3$ $\bar{a}_1$ $\bar{a}_2$ a_1
$\bar{a}_3$ $\bar{a}_1$ $\bar{a}_2$ a_2
$\bar{a}_3$ $\bar{a}_4$ $\bar{a}_2$ a_3
1 d_1 d_6+1 e_4 c_5 c_0
r $2r-2$
A_3
1 d_6 e_4 c_5 c_0
See Fig. 9(c)
1 d_1 e_4 e_5 e_5+1 c_0
$2r-1$ $r-1$
B_1
1 d_1 e_5 e_5+1 c_0
$r+1$ $2r-3$
1 d_1 e_4 c_5 c_1+1 c_0
1 d_1 e_4 c_1 c_1+1 c_0
$2r$ $r-2$
C_1
$\bar{a}_3$ $\bar{a}_1$ $\bar{a}_2$ a_3 a_1
$\bar{a}_3$ $\bar{a}_1$ $\bar{a}_2$ $\underline{a}_3$ a_2
$\bar{a}_3$ $\bar{a}_1$ $\bar{a}_2$ $\underline{a}_3$ a_3
d_6+1 c_5+1
1 d_1 e_4 c_5 c_1+1 c_0
r $r-2$ $r-1$
Same as
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3$
e_4+1 c_5+1
1 d_1 e_4 c_5 c_1+1 c_0
r $r-2$ $r-1$
1 d_1 c_5 c_5+1 c_1+1 c_0
$r+1$ $2r-4$
e_4+1 c_2+1
1 d_1 e_4 c_5 c_0
r $2r-3$
D
c_2+1 c_0
1 d_1 e_4 c_2 c_1+1
$2r$ $r-3$
E_1

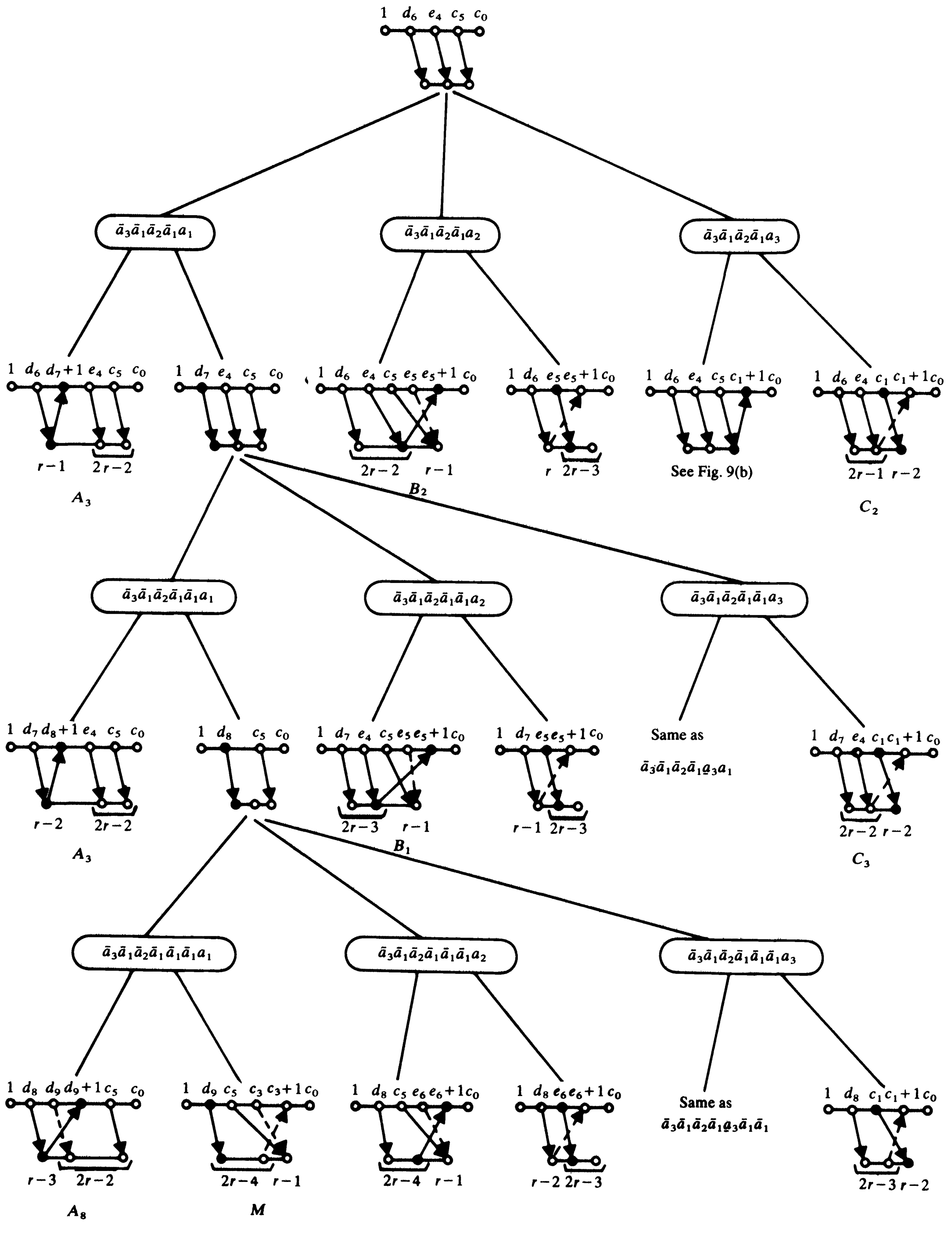
(a)

FIG. 9. *Proof for* $k=3r+1\geqq 10$.


d_5+1
1 d_1 c_5 c_0
$\bar{a}_3\bar{a}_1\underline{a}_1a_1$
$\bar{a}_3\bar{a}_1\underline{a}_1a_2$
$\bar{a}_3\bar{a}_1\underline{a}_1a_3$
1 d_1 d_2 d_2+1 c_5 c_0
$r-2$ $2r$
A_4
1 d_2 d_5+1 c_5 c_0
e_3+1
1 d_1d_5+1 c_5 c_0
$2r-1$ $r-1$
F_1
1 $d_1d_5+1e_3$ c_5 c_0
$r-1$ $2r-1$
A_5
d_5+1 c_5+1
1 d_1 d_5 c_5 c_1+1 c_0
$r-1$ r $r-1$
d_5+1 c_1+1
1 d_1 c_1 c_0
$2r$ $r-2$
F_2
G
$\bar{a}_3\bar{a}_1\underline{a}_1a_1$
$\bar{a}_3\bar{a}_1\underline{a}_1\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\underline{a}_1\bar{a}_1a_3$
1 d_2 $d_3d_3+1c_5$ c_0
$r-3$ $2r$
A_6
1 $d_3d_5+1c_5$ c_0
d_5+1
1 d_2 e_3+1c_5 c_0
$2r-2$ $r-1$
H_1
d_5+1
1 d_2 e_3 c_5 c_0
$r-2$ $2r-1$
A_5
d_5+1 c_1+1
1 d_2 d_5 c_5c_5+1 c_0
$r-2$ r $r-1$
d_5+1 c_1+1
1 d_2 c_1 c_0
$2r-1$ $r-2$
H_2
I
$\bar{a}_3\bar{a}_1\underline{a}_1\bar{a}_1\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\underline{a}_1\bar{a}_1\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\underline{a}_1\bar{a}_1\bar{a}_1a_3$
1 d_3 $d_4d_4+1c_5$ c_0
$r-4$ $2r$
d_4+1 c_5
1 d_4 d_5+1 c_5+1 c_0
$r-3$ r $r-1$
d_5+1 c_5
1 d_3 e_3+1 c_0
$2r-3$ $r-1$
J_1
d_5+1
1 d_3 e_3 c_5 c_0
$r-3$ $2r-1$
A_5
d_5+1 c_5+1 c_0
1 d_3 d_5 c_5 c_1+1
$r-3$ r $r-1$
d_5+1 c_1+1
1 d_3 c_1 c_0
$2r-2$ $r-2$
J_2

(b)

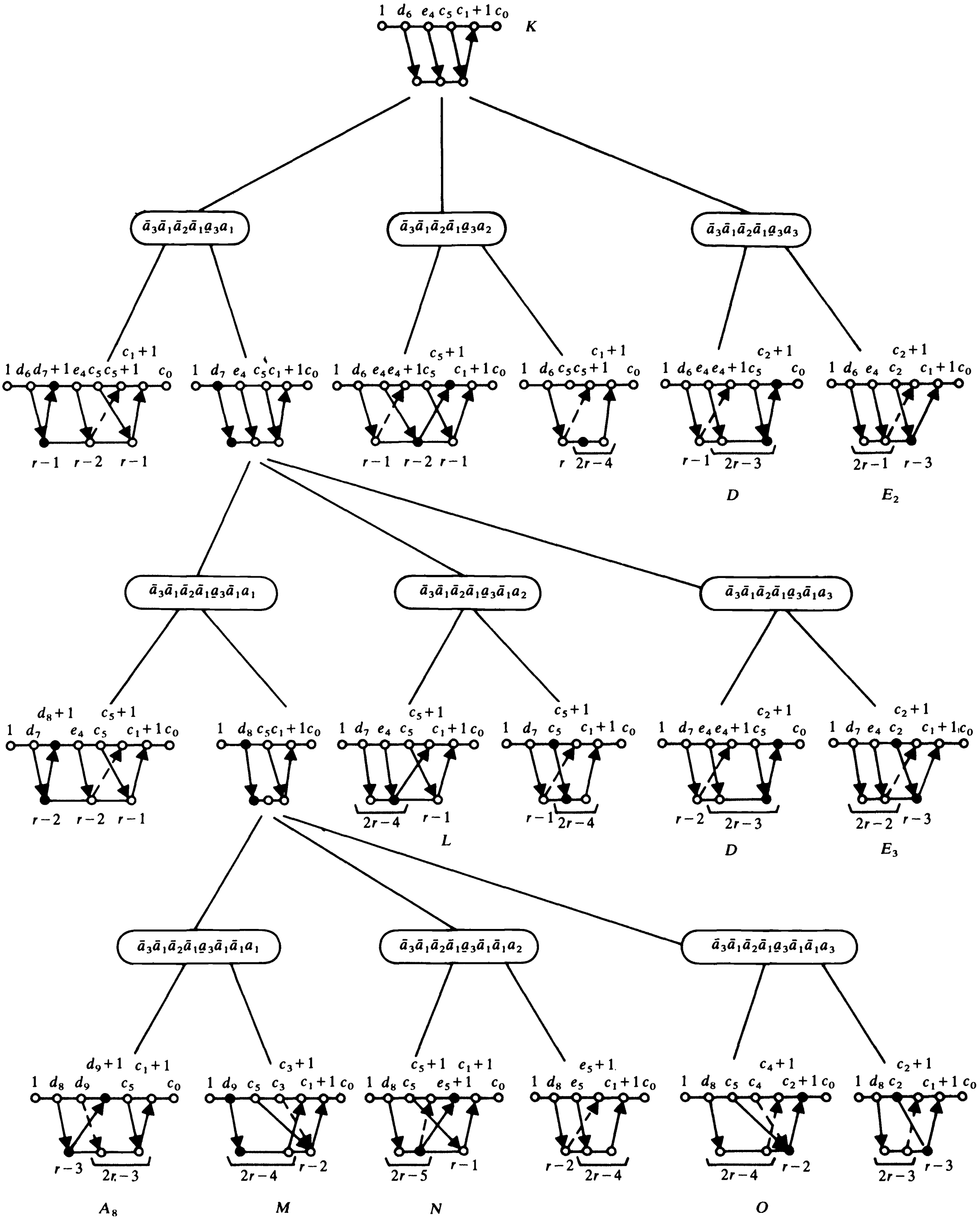
FIG. 9. *Proof for* $k=3r+1 \geqq 10$.



$1\ d_6\ e_4\ c_5\ c_0$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1a_3$
$1\ d_6\ d_7+1\ e_4\ c_5\ c_0$
$r-1\quad 2r-2$
A_3
$1\ d_7\ e_4\ c_5\ c_0$
$1\ d_6\ e_4\ c_5\ e_5\ e_5+1\ c_0$
$2r-2\quad r-1$
B_2
$1\ d_6\ e_5 e_5+1\ c_0$
$r\quad 2r-3$
$1\ d_6\ e_4\ c_5\ c_1+1\ c_0$
See Fig. 9(b)
$1\ d_6\ e_4\ c_1\ c_1+1\ c_0$
$2r-1\quad r-2$
C_2
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1a_3$
$1\ d_7\ d_8+1\ e_4\ c_5\ c_0$
$r-2\quad 2r-2$
A_3
$1\ d_8\ c_5\ c_0$
$1\ d_7\ e_4\ c_5\ e_5\ e_5+1\ c_0$
$2r-3\quad r-1$
B_1
$1\ d_7\ e_5 e_5+1\ c_0$
$r-1\ 2r-3$
Same as
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1a_3a_1$
$1\ d_7\ e_4\ c_1 c_1+1\ c_0$
$2r-2\ r-2$
C_3
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\bar{a}_1\bar{a}_1a_3$
$1\ d_8\ d_9 d_9+1\ c_5\ c_0$
$r-3\quad 2r-2$
A_8
$1\ d_9\ c_5\ c_3\ c_3+1\ c_0$
$2r-4\quad r-1$
M
$1\ d_8\ c_5\ e_6\ e_6+1\ c_0$
$2r-4\quad r-1$
$1\ d_8\ e_6 e_6+1\ c_0$
$r-2\ 2r-3$
Same as
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1a_3\bar{a}_1\bar{a}_1$
$1\ d_8\ c_1 c_1+1\ c_0$
$2r-3\ r-2$

(c)

FIG. 9. *Proof for* $k = 3r+1 \geqq 10$.


1 d_6 e_4 c_5 c_1+1 c_0
K
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3a_3$
c_1+1
1 d_6d_7+1 $e_4c_5c_5+1$ c_0
$r-1$ $r-2$ $r-1$
1 d_7 e_4 $c_5c_1+1c_0$
c_5+1
1 d_6 $e_4e_4+1c_5$ c_1+1c_0
$r-1$ $r-2$ $r-1$
c_1+1
1 $d_6c_5c_5+1$ c_0
r $2r-4$
c_2+1
1 $d_6e_4e_4+1c_5$ c_0
$r-1$ $2r-3$
D
c_2+1
1 d_6 e_4 c_2 c_1+1c_0
$2r-1$ $r-3$
E_2
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1a_3$
d_8+1 c_5+1
1 d_7 e_4 c_5 c_1+1c_0
$r-2$ $r-2$ $r-1$
1 d_8 $c_5c_1+1c_0$
c_5+1
1 d_7 e_4 c_5 c_1+1c_0
$2r-4$ $r-1$
L
c_5+1
1 d_7 c_5 c_1+1 c_0
$r-1$ $2r-4$
c_2+1
1 d_7 $e_4e_4+1c_5$ c_0
$r-2$ $2r-3$
D
c_2+1
1 d_7 e_4 c_2 c_1+1c_0
$2r-2$ $r-3$
E_3
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1\bar{a}_1a_1$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1\bar{a}_1a_2$
$\bar{a}_3\bar{a}_1\bar{a}_2\bar{a}_1\underline{a}_3\bar{a}_1\bar{a}_1a_3$
d_9+1 c_1+1
1 d_8 d_9 c_5 c_0
$r-3$ $2r-3$
A_8
c_3+1
1 d_9 c_5 c_3 c_1+1c_0
$2r-4$ $r-2$
M
c_5+1 c_1+1
1 d_8 c_5 e_5+1 c_0
$2r-5$ $r-1$
N
e_5+1
1 d_8 e_5 c_1+1c_0
$r-2$ $2r-4$
c_4+1
1 d_8 c_5 c_4 c_2+1c_0
$2r-4$ $r-2$
O
c_2+1
1 d_8c_2 c_1+1c_0
$2r-3$ $r-3$

(d)

FIG. 9. *Proof for* $k=3r+1\geqq 10$.

Analysis for labeled nodes in Fig. 9.

A. $c_0 - c_5 = 2^{r-2} + 2^{r-3}$

—A$_7$. $c_0 - d_4 = [\frac{267}{7} 2^{r-5}] - 1 \geqq [\frac{33}{7} 2^{r-2}] = g(2r-1, r-1, r-3)$ for $r \geqq 4$.

(Configuration A$_7$ does not occur when $r = 3$, see G.)

—A$_6$.

$$c_0 - d_3 = \begin{cases} 9 & (\text{for } r = 3) \\ [\frac{137}{7} 2^{r-4}] - 1 & (\text{for } r \geqq 4) \end{cases}$$

$$\geqq [\tfrac{33}{7} 2^{r-2}] = g(2r-1, r-1, r-3).$$

—A$_4$. $c_0 - d_2 = [\frac{72}{7} 2^{r-3}] - 1 \geqq [\frac{33}{7} 2^{r-2}] = g(2r-1, r-1, r-3)$.

—A$_1$. $c_0 - d_1 = c_0 - d_2 \geqq g(2r-1, r-1, r-3)$.

—A$_5$. $c_0 - e_3 = [\frac{19}{7} 2^{r-2}] = g(2r-2, r-1, r-3)$.

—A$_2$. $c_0 - e_2 \geqq c_0 - e_3 = g(2r-2, r-1, r-3)$.

—A$_8$.

$$c_0 - d_9 = \begin{cases} 3 & (\textit{for } r = 3) \\ [\frac{53}{7} 2^{r-4}] - 1 & (\textit{for } r \geqq 4) \end{cases}$$

$$\geqq [\tfrac{12}{7} 2^{r-2}] = g(2r-3, r-1, r-3).$$

—A$_3$. $c_0 - e_4 = [\frac{31}{7} 2^{r-3}] \geqq [\frac{12}{7} 2^{r-2}] = g(2r-3, r-1, r-3)$.

B. $e_5 - e_4 \geqq [\frac{3}{7} 2^{r-1}] \geqq 2^{r-3} + 2^{r-4}$.

—B$_1$. $e_5 - d_1 \geqq \left[\dfrac{15 \cdot 2^{r-1} - 2}{7}\right] \geqq [\frac{59}{7} 2^{r-3}] = g(2r-2, r-2, r-4)$.

—B$_2$. $e_5 - d_6 = e_5 - d_1 - 2^{r-1} \geqq g(2r-2, r-2, r-4) - 2^{r-1} = g(2r-3, r-2, r-4)$.

—B$_3$. $e_5 - d_7 = e_5 - d_6 - 2^{r-2} \geqq g(2r-3, r-2, r-4) - 2^{r-2} = g(2r-4, r-2, r-4)$.

C. $c_1 - e_4 = [\frac{24}{7} 2^{r-3}] \geqq 2^{r-2} + 2^{r-3}$.

—C$_1$. $c_1 - d_1 = [\frac{72}{7} 2^{r-3}] - 1 \geqq [\frac{33}{7} 2^{r-2}] = g(2r-1, r-1, r-3)$.

—C$_2$. $c_1 - d_6 = c_1 - d_1 - 2^{r-1} \geqq g(2r-1, r-1, r-3) - 2^{r-1} = g(2r-2, r-1, r-3)$.

—C$_3$. $c_1 - d_7 = c_1 - d_6 - 2^{r-2} \geqq g(2r-2, r-1, r-3) - 2^{r-2} = g(2r-3, r-1, r-3)$.

—C$_4$. $c_1 - d_8 = c_1 - d_7 - 2^{r-3} \geqq g(2r-3, r-1, r-3) - 2^{r-3} = g(2r-4, r-1, r-3)$.

D. For $r = 3$, $c_2 = c_1$. Therefore it suffices to consider the case $a_3 \leftarrow c_2$ only. Namely, configuration D does not occur when $r = 3$. For $r \geqq 4$,

$$c_2 - c_5 = 2^{r-3} + 2^{r-4}.$$

$$c_2 - e_4 = [\tfrac{41}{7} 2^{r-4}] \geqq [\tfrac{33}{7} 2^{r-4}] = g(2r-4, r-2, r-4).$$

E. For $r = 3$, E$_2$ and E$_3$ are covered by K. For E$_1$, $N(c_2 - e_4, c_2 - d_1) = N(3, 9) \geqq N(2, 9) = N(2, 2 + 2^3 - 1) > 5$ from Lemma 2. The proof for $r = 4$ is illustrated in Fig. 10.

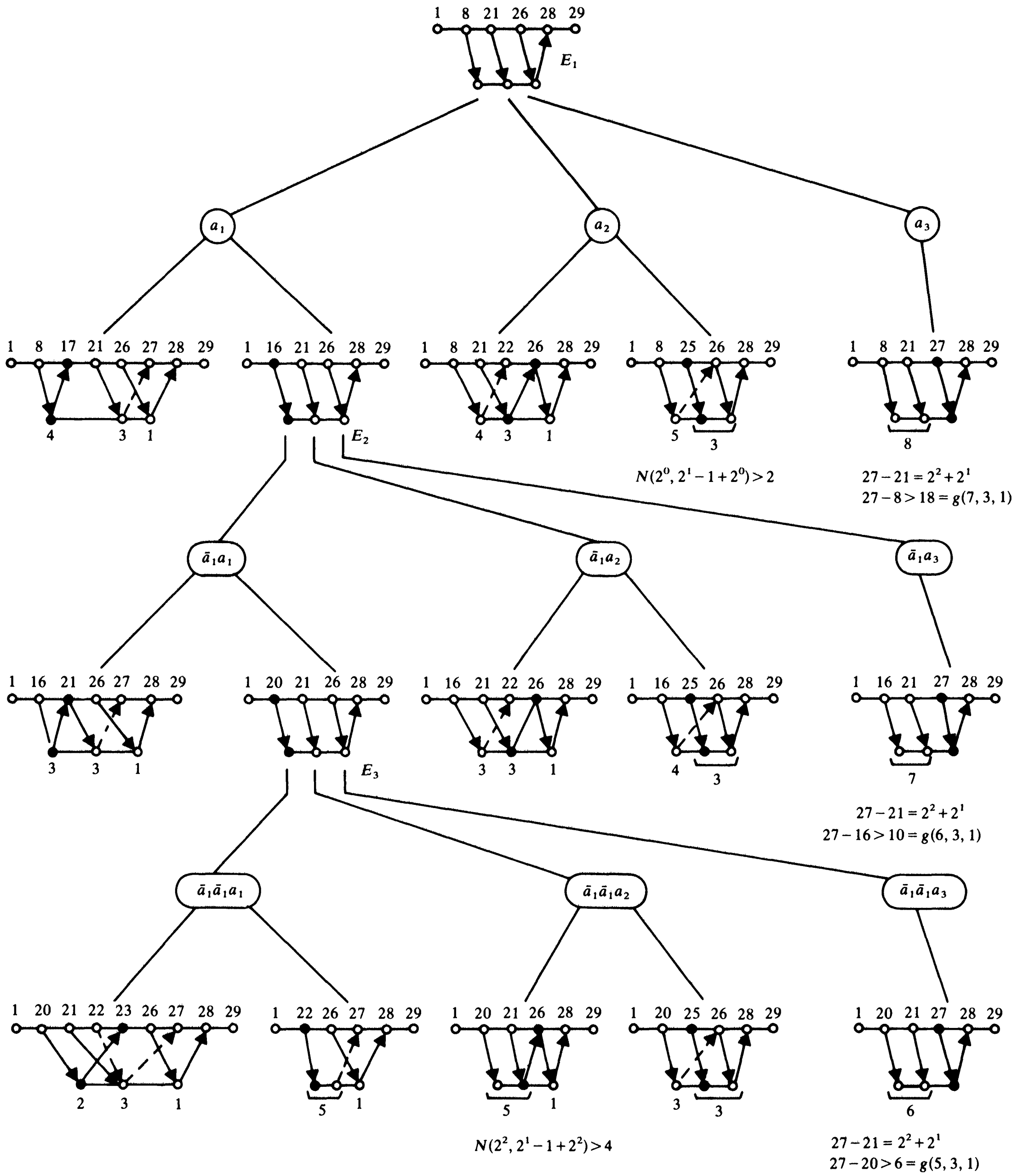


FIG. 10. *Proof for E with k* = 13.

For $r \geqq 5$,

$$c_2 - c_4 = [\tfrac{41}{7}2^{r-4}] \geqq 2^{r-2} + 2^{r-4}.$$

—E$_1$. $c_2 - d_1 = [\tfrac{137}{7}2^{r-4}] - 1 \geqq [\tfrac{17}{7}2^{r-1}] = g(2r-1, r-1, r-4)$.

—E$_2$. $c_2 - d_6 = c_2 - d_1 - 2^{r-1} \geqq g(2r-1, r-1, r-4) - 2^{r-1} = g(2r-2, r-1, r-4)$.

—E$_3$. $c_2 - d_7 = c_2 - d_6 - 2^{r-2} \geqq g(2r-2, r-1, r-4) - 2^{r-2} = g(2r-3, r-1, r-4)$.

F. For $r = 3$ and 4, $d_5 - d_1 = 2^{r-2}$.

—F$_1$. $e_3 - d_1 = 2^{r-1} + 2^{r-2} - 1$. Therefore $N(d_5 - d_1, e_3 - d_1) > (r-1) + (r-2) + 1 = 2r - 2$ by Lemma 2.

—F$_2$. $c_1 - d_1 = 2^r + 2^{r-2} - 1$. Therefore $N(d_5 - d_1, c_1 - d_1) > r + (r-2) + 1 = 2r - 1$ by Lemma 2.

For $r \geqq 5$,

$$d_5 - d_1 = [\tfrac{17}{7}2^{r-3}] \geqq 2^{r-2} + 2^{r-5}.$$

—F$_1$. $e_3 - d_1 = [\tfrac{41}{7}2^{r-3}] = g(2r-2, r-1, r-5)$.

—F$_2$. $c_1 - d_1 = [\tfrac{72}{7}2^{r-3}] - 1 \geqq [\tfrac{69}{7}2^{r-3}] = g(2r-1, r-1, r-5)$.

G. For $r = 3$, $d_3 = d_2$. Therefore it suffices to consider the case $a_1 \to d_3 + 1$ only.

H. For $r = 3$ and 4, the analysis is similar to F. For $r \geqq 5$,

$$d_5 - d_2 = d_5 - d_1 - 2^{r-3} \geqq 2^{r-3} + 2^{r-5}.$$

—H$_1$. $e_3 - d_2 = e_3 - d_1 - 2^{r-3} \geqq g(2r-2, r-1, r-5) - 2^{r-3} = g(2r-3, r-2, r-5)$.

—H$_2$. $c_1 - d_2 = c_1 - d_1 - 2^{r-3} \geqq g(2r-1, r-1, r-5) - 2^{r-3} = g(2r-2, r-2, r-5)$.

I. For $r = 4$, $d_4 = d_3$. Therefore it suffices to consider the case $a_1 \to d_4 + 1$ only.

J. For $r = 4$, the analysis is similar to F. For $r \geqq 5$,

$$d_5 - d_3 = d_5 - d_2 - 2^{r-4} \geqq 2^{r-4} + 2^{r-5}.$$

—J$_1$. $e_3 - d_3 = e_3 - d_2 - 2^{r-4} \geqq g(2r-3, r-2, r-5) - 2^{r-4} = g(2r-4, r-3, r-5)$.

—J$_2$. $c_1 - d_3 = c_1 - d_2 - 2^{r-4} \geqq g(2r-2, r-2, r-5) - 2^{r-4} = g(2r-3, r-3, r-5)$.

K. For $r = 3$, the configuration is the following one:

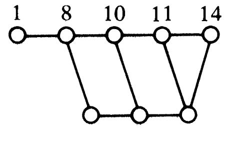


FIG. 10'. *Configuration K for r = 3.*

Add a broken link $a_1 \cdots > b_{11}$, then it is easy to check that a_1 needs two comparisons to merge while a_2 and a_3 together needs four. Since five comparisons have already been used, the sum is $11 > 3r + 1 = 10$.

K'. For $r = 4$, the configuration is the following one:

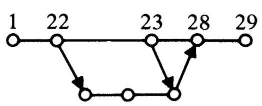


FIG. 11. *Configuration K' for r = 4.*

Seven comparisons have been made and the number of possible orderings of the 29 elements is reduced to 55. However, any further comparison, with one exception, will partition the number of 55 into two numbers with one greater than 32, hence at least six more comparisons, the only exception is when we compare a_2 with b_{25}. Consider the case $b_{25} \rightarrow a_2$. Then the new configuration is the following one:

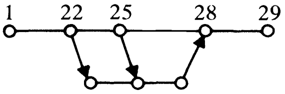


FIG. 12. *A configuration.*

However, when we do the (3, 5) problem and compare a_2 with b_3, regardless of which one is greater, we obtain a configuration equivalent to the above one. Since we know that the (3, 5) problem needs seven comparisons, the above configuration needs six more comparisons.

L. For $r \geqq 4$.

$$c_5 - e_4 = [\tfrac{5}{7}2^{r-2}] \geqq 2^{r-3} + 2^{r-5},$$

$$c_5 - d_7 = [\tfrac{8}{7}2^{r-2}] - 1 \geqq [\tfrac{13}{7}2^{r-3}] = g(2r-5, r-2, r-5).$$

M. For $r \geqq 5$,

$$c_3 - d_9 = [\tfrac{25}{7}2^{r-4}] - 1 \geqq [\tfrac{12}{7}2^{r-3}] = f_2(2r-5) + 1.$$

For $r = 4$, $d_9 = c_5 = 23$. Therefore configuration M without the broken link is the following one:

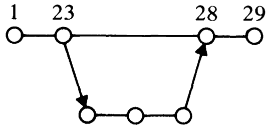


FIG. 13. *Configuration M for $r = 4$.*

which needs six comparisons since it is equivalent to the (3, 4) problem.

M′. Similar to M for $r \geqq 5$.

N. For $r \geqq 5$,

$$c_5 - d_5 = [\tfrac{9}{7}2^{r-3}] - 1 \geqq [\tfrac{17}{7}2^{r-4}] = f_2(2r-6) + 1.$$

O. For $r \geqq 5$,

$$c_4 - d_8 = [\tfrac{25}{7}2^{r-4}] - 1 \geqq [\tfrac{12}{7}2^{r-3}] = f_2(2r-5) + 1.$$

P. If configuration P can be done in less than $2r-3$ comparisons, then we show that configuration A_8 can be done in less than $2r-2$ comparisons, a contradiction to our analysis of A_8. For configuration A_8, we first compare a_3 with $c_1 + 1$. If $a_3 \rightarrow c_1 + 1$, then we obtain configuration P which, by assumption, needs less than $2r-3$ configurations. If $c_1 + 1 \rightarrow a_3$, then since

$$c_0 - c_1 - 1 = f_1(r-3), \qquad c_0 - d_1 \leqq f_1(r-1),$$

a_3 can be merged in $r-3$ and a_2 in $r-1$ comparisons. In any case, configuration A_8 needs at most $2r-3$ comparisons.

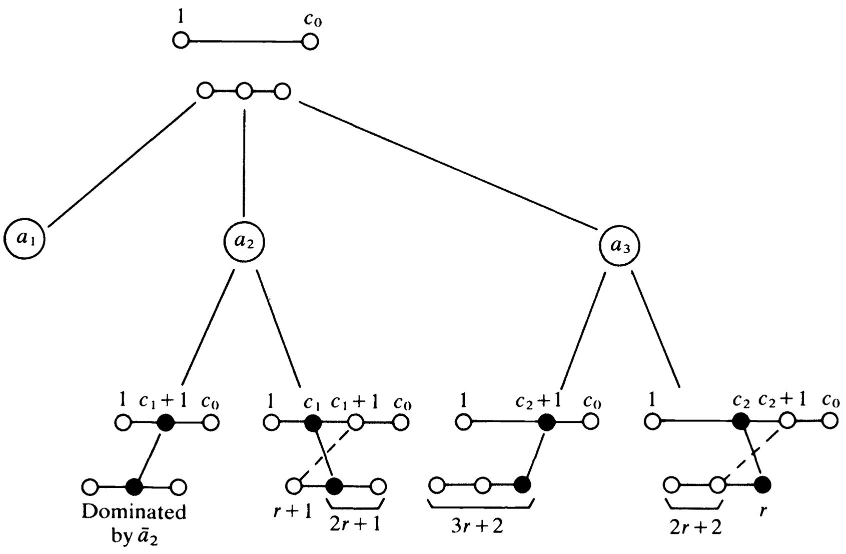


FIG. 14. *Proof for* $k=3r+2\geqq 8$.

Finally, we study the case $k=3r+2\geqq 8$. The proof is given in Fig. 14. Definitions of symbols used in Fig. 14:

$$c_0=f_3(3r+2)+1=\begin{cases}9 & \text{for } r=2,\\ \left[\dfrac{17\cdot 2^r-6}{7}\right] & \text{for } r\geqq 3,\end{cases}$$

$$c_1=\left[\frac{c_0}{2}\right]=\begin{cases}5 & \text{for } r=2,\\ \left[\frac{17}{7}2^{r-1}\right] & \text{for } r\geqq 3,\end{cases}$$

$$c_2=f_3(3r+1)+1=\left[\tfrac{107}{7}2^{r-3}\right]-1.$$

Analysis for the labeled terminal nodes are straightforward and omitted.

4. Some concluding remarks. Manacher [4], [5] recently gives an algorithm to sort n elements which beats the Ford and Johnson sorting algorithm over infinite many values of n. The Manacher sorting algorithm makes use of optimal merging results, typically when one subset to be merged is very small. Therefore, the knowledge of $f_m(k)$ for small m is likely to help making further improvement over the Manacher sorting algorithm.

With the formulas of $f_3(k)$ now available, we can venture to make some conjecture about the general $f_m(k)$. Noting that

$$\frac{f_1(r)}{f_1(r-1)}\to 2,$$

$$\frac{f_2(2r)}{f_2(2r-1)}\to\frac{17}{12}\sim 1.42,\qquad \frac{f_2(2r+1)}{f_2(2r)}\to\frac{24}{17}\sim 1.41,$$

$$\frac{f_3(3r)}{f_3(3r-1)}\to\frac{43}{34}\sim 1.26\frac{f_3(3r+1)}{f_3(3r)}\to\frac{107}{86}\sim 1.24,\qquad \frac{f_3(3r+2)}{f_3(3r+1)}\to\frac{136}{107}\sim 1.27,$$

we conjecture

$$\frac{f_m(k)}{f_m(k-1)} \to 2^{1/m}.$$

Since we know

$$f_m(2m) = m+1$$

using the conjecture, we obtain

$$f_m(k) \to (m+1)2^{k/m-2}.$$

Checking with known results for $m = 1, 2, 3$, we note

$$\frac{f_1(k)}{2 \cdot 2^{k-1}} \to 2, \qquad \frac{f_2(k)}{3 \cdot 2^{k/3-2}} \to 1.6, \qquad \frac{f_3(k)}{4 \cdot 2^{k/4-2}} \to 1.5.$$

Therefore the statistic $n \cdot 2^{(k+1)/m-2}$ is not a particularly good estimate of $f_m(k)$.

On the other hand, we note that the numbers $f_2(2r+2)$ differ very little from the numbers $f_3(3r+2)$. In fact,

$$1 \geqq f_2(2r+2) - f_3(3r+2) \geqq 0.$$

Furthermore, since $f_4(6) = 3 = f_3(5)$, $f_4(10) = 8 = f_3(8)$, and the strongly suspected relation $f_m(mr+i) \sim 2f_m(m(r-1)+i)$, we anticipate the numbers $f_4(4r+2)$ will look very much like the numbers $f_3(3r+2)$ and $f_2(2r+2)$. The same can be said for the numbers $f_5(5r+2)$. Although we cannot conclude anything in general, it is possible that a set of numbers similar to $f_2(2r+2)$ will keep turning up in $f_m(k)$ for every m. If that turns out to be true, then using this set of numbers as base plus using the conjecture $f_m(k)/f_m(k-1) \sim 2^{1/m}$, we can come up with a very good estimate for the general $f_m(k)$. Another interesting relation is observed from the following facts:

$$f_1(r+1) - f_1(r) = 2^r, \qquad f_2(2r+1) - f_2(2r) = 2^{r-1}, \qquad f_3(3r+1) - f_3(3r) = 3 \cdot 2^{r-3}.$$

Whether this neat expression exists for $f_m(mr+1) - f_m(mr)$ for all m, or it is purely a coincidental event for these particular values of m is too early to conclude.

REFERENCES

[1] R. L. GRAHAM, *On sorting by comparisons*, Proc. of Atlas Symposium, 2 (1971), pp. 263–269.

[2] F. K. HWANG AND S. LIN, *An optimal algorithm for merging an ordered set of length two with another ordered set*, Acta Information, 1, 2 (1971), pp. 145–158.

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

[4] G. K. MANACHER, *The Ford–Johnson sorting algorithm is not optimal*, to appear.

[5] ———, *Nontrivial improvements to the Hwang–Lin algorithm*, to appear.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0008 $01.00/0

NEW FAST ALGORITHMS FOR MATRIX OPERATIONS*

V. YA. PAN†

Abstract. A new technique of trilinear operations of aggregating, uniting and canceling is introduced and applied to constructing fast linear noncommutative algorithms for matrix multiplication. The result is an asymptotic improvement of Strassen's famous algorithms for matrix operations.

Key words. Fast algorithms, complexity of computation, arithmetic complexity, linear algebraic problems, matrix multiplication, bilinear forms, trilinear form

1. Introduction. Probably the most exciting result in algebraic complexity theory was obtained by V. Strassen in 1968 (see [23]). He discovered that matrix multiplication (MM), matrix inversion (MI), evaluation of determinant (ED) and solving linear system of equations (SLS) can be done by $O(N^\alpha)$ arithmetic operations (where N is the size of the problem that is the order of the square matrices involved, and $\alpha = \log_2 7 \approx 2.807$), rather than by $O(N^3)$ operations, required in classical methods. Strassen's algorithms reduce these four problems to a problem of constructing a fast linear algorithm for multiplying two 2×2 matrices. It seemed surprising that fast algorithms for all these and for some other important problems like the transitive closure problem in graph theory, see e.g. [1], of any large size could be immediately constructed if a fast linear algorithm of a certain type (we will use the notation LA for such algorithms) for multiplying two matrices of a specific size was given. Even a small improvement of such a linear algorithm for matrix multiplication (e.g., reducing the complexity of LA in comparison with Strassen's algorithm by even 1 for any size $N = 2^k$) would automatically result in asymptotic improvement of the algorithms for the above-mentioned problems MM, MI, ED, SLS, etc. The attempts to find such an improvement of linear algorithms for matrix multiplication were numerous before and particularly after the publication of Strassen's paper. Despite several bright ideas suggested and despite the progress in understanding the problem (see [3]–[13], [16], [19], [21], [22], [24]–[30], for surveys see [2]–[8] and also Remarks 1–3 and § 14 in the present paper) no algorithms were constructed such that they would give an asymptotic improvement of Strassen's method. In this paper a new technique for LA transformations from a trivial one with the complexity n^3 to fast ones is presented. The transformations are the chains of elementary ones which will be called trilinear operations. Each LA can be written as a chain of these elementary operations of two to four kinds. Such a representation makes the ways of constructing fast LA more comprehensive. Trilinear operations of uniting terms reduce the complexity of LA. Thus the main objective will be in increasing the number of unitings in the chains. Though the exploration of this technique has just been started (not counting a short period in 1972, see [19], its power has already been demonstrated in this paper. LA which are asymptotically faster than Strassen's are described in §§ 7–14. By using these algorithms and the previously mentioned reduction of the other problems to constructing LA, all the problems MM, MI, ED, SLS, etc. of the size N can be solved by $O(N^{2.780})$, rather than by $O(N^{2.807})$, arithmetic operations.

It is well known (see [19], [24]) that any LA can be written in either a bilinear, or trilinear version and both are equivalent. However, in this paper, all the new fast LA are presented in the trilinear version which seems more appropriate for them. The bilinear version and some auxiliary techniques are exposed in the next section. The trilinear

* Received by the editors February 1, 1979, and in revised form May 18, 1979.

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

version and an example of fast LA from [19] are described in § 3 where also a simple example demonstrating the technique of trilinear aggregating and uniting terms is introduced and studied. An LA from [19] is analyzed and generalized as a model. In § 4 the procedure of constructing such an LA is generalized and more formally defined (in § 5). Then in §§ 4–6 the lower and upper bounds on the complexity of LA resulting from such a procedure are established. They are nonlinear and are proved to be optimal for this procedure. A modification of the latter resulting in a slightly faster LA is also described in § 6. The LA presented in § 6 give nonasymptotic improvements over Strassen's method. In §§ 7–13 the technique is further developed. Then we combine aggregating and uniting terms with canceling terms. The result is an asymptotic improvement of Strassen's algorithms. The technique of an asymptotically fast LA and the estimates of its complexity are presented in §§ 7–10. In § 11 such an LA (which is faster than Strassen's) is formally presented as a chain of identities and without any reference to the technique of aggregating, uniting and canceling terms. Thus § 11 is independent from others. So are the asymptotically fast algorithms for the problems MM, IM, ED, SLE, etc. which are immediate from the LA described in § 11. The only necessary reference is the not very difficult and now well-known theorem about a reduction of the mentioned problems to constructing LA represented in the trilinear form (the latter reduction is reintroduced in §§ 2 and 3; see Theorem 1 and the equivalency of formulae (1) and (2) there). In § 12 we reintroduce operations of aggregating and uniting, so that they both can be considered from one general point of view. We also show that any decomposition of any trilinear form can be obtained as a chain of aggregatings and inverse operations (disaggregatings). Then in § 13 we apply these generalized aggregatings to construct even faster LA. In § 14 the main theorem is stated, the complexity of presented algorithms is illustrated by tables and the technique used is summarized. In § 15 some open problems are listed. The last section contains Acknowledgements.

The author pursued two objectives in this paper: to make the construction of fast LA comprehensible not only for mathematicians, and, on the other hand, to mark a line for a possible extension and generalization of the technique used. Thus a less formal or simplified exposition sometimes will precede proofs and formal description. The reader who is not mathematically minded certainly may skip part of the material, particularly, the whole § 5 and most parts of §§ 4 and 12 (except formulae (3), (4), and (7)). In the extreme case he may do with just introductory sections 1–3, Algorithms 1, 2, 2a, and 3, theorems and tables of § 14. On the other hand, the reader who is more interested in the technique used can find some exercises in §§ 6, 10, and 13.

2. Some notation, definitions, and auxiliary techniques. Let integers n and M be given. Let A, B, C denote $n \times n$ matrices, a_{ij}, b_{ij}, c_{ij} denote their entries, such that

$$A = \|a_{ij}\|, \quad B = \|b_{ij}\|, \quad C = \|c_{ij}\|.$$

Let $L(A)$, $L(B)$, $L(C)$ denote linear forms of the entries of the matrices A, B, C. Let M triplets of linear forms $L'_q(A)$, $L^2_q(B)$, $L^3_q(C)$ be given, such that

$$L^1_q(A) = \sum_{i,j=0}^{n-1} \alpha^q_{ij} a_{ij},$$

$$L^2_q(B) = \sum_{i,j=0}^{n-1} \beta^q_{ij} b_{ij},$$

$$L^3_q(C) = \sum_{i,j=0}^{n-1} \gamma^q_{ij} c_{ij}, \qquad q = 1, \cdots, M.$$

Suppose that for any pair of matrices A, B and for any pair of integers (k, l), such that $0 \leqq k, l \leqq n-1$ the following system of identities hold:

$$(1)\quad \begin{aligned} &p_q = L_q^1(A)L_q^2(B), \qquad q = 1, 2, \cdots, M, \\ &\sum_{s=0}^{n-1} a_{ks}b_{sl} = \sum_{q=1}^{M} \gamma_{kl}^q p_q = \sum_{q=1}^{M} \gamma_{kl}^q L_q^1(A)L_q^2(B), \qquad k, l = 0, 1, \cdots, n-1. \end{aligned}$$

Then if the entries of the matrices A and B are given, (1) describes an algorithm for computing $C = AB$ which is called a linear noncommutative algorithm for multiplying two $n \times n$ matrices A and B, n is called the size of the problem, M is called the complexity of the algorithm. We will use a notation LA for the latter.

Remark 1. The definitions and the technique presented in this paper can be easily generalized for the problem of multiplying $n \times p$ by $p \times m$ matrices with any n, p, m and in many cases for the problem of the evaluation of a set of bilinear forms or (which is the same) of a trilinear form (see [5], [19], [24]). Note that the size of the problem MM is always determined by a triplet (m, n, p) of integers. It is easy to observe (see [19] or [13]) that the complexity of the optimal LA for a problem MM is invariant to all six possible permutations in a triplet (m, n, p), e.g. substituting the problem of multiplying $m \times n$ by $n \times p$ matrices for the problem of multiplying $p \times m$ by $m \times n$ matrices. The described $\mathrm{LA} = \mathrm{LA}(n)$ can be turned into $\mathrm{LA}(m, n, p)$ for nonsquare matrix multiplications.

Strassen's fast algorithms for MM, MI, ED, SLS are based on two following theorems:

THEOREM 1. (V. Strassen [23]). *Let a positive integer n and a linear, noncommutative algorithm* LA *for multiplying two $n \times n$ matrices be given such that its complexity is equal to M. Then algorithm for solving the problems* MM, MI, ED, SLS *by only $O(N^{\log_n M})$ arithmetic operations can be constructed for any N. Here N is the size of square matrices involved in the problems* MM, MI, ED, SLS.

Remark 2. The converse theorems expressing the lower bounds on the complexity of the problems MM and MI through the lower bounds on the complexity of LA also hold (see the theorems for MM without divisions in [28] and in the general case in [19], [25]).

THEOREM 2 (V. Strassen [23]). *There exists a linear noncommutative algorithm* LA *for multiplying two 2×2 matrices whose complexity is equal to* 7.

Remark 3. The bound 7 on $M = M(2)$ for the size $n = 2$ is sharp since $m(2) \geqq 7$ always; see [29]. Moreover, LA for the size $n = 2$ whose complexity $M(2)$ is equal to 7 is unique up to within a linear transformation (see [19], [10], [13]). A further asymptotic speed-up could be achieved by constructing a fast LA for $n = 3$ such that $M = M(3) \leqq 21$. Yet this problem turned out to be very difficult (if solvable). Thus the more promising way (as it seemed, at least to the present author) consisted in constructing a fast LA for a greater size of $n \times n$ matrices. An appropriate value for n as a basis for a recursion via Theorem 1 can be easily chosen if a formula for a number $M = M(n)$ of multiplications used in LA for any n has been obtained. Of course, an asymptotically fast algorithm can be constructed via an analogous recursion if a fast enough algorithm for multiplication of two nonsquare matrices of any particular sizes has been designed (see [19] or [13]).

3. Linear algorithms for matrix multiplication as decompositions of a given trilinear form. The evaluation of a set of bilinear forms in M multiplications and decomposing a trilinear form as a sum of M terms are two equivalent problems [19], [24]. In particular, the evaluation of the product of $n \times p$ by $p \times m$ matrices and decomposing the trace of the product of three matrices ($n \times p$ one by $p \times m$ one by $m \times n$ one) are two equivalent problems [19]. Here is the decomposition in the case $n = p = m$, presented in general form:

$$\sum_{i,j,k=0}^{n-1} a_{ij}b_{jk}c_{ki} = \sum_{q=1}^{M(n)} L_q^1(A)L_q^2(B)L_q^3(C). \tag{2}$$

It is easy to verify that (1) and (2) are equivalent. In the sequel we can and will study LA in the form (2) rather than in the form (1). In a sense, (1) seems even simpler than (2). However, some LA can be better expressed in (2) than in (1). Consider the following example from [19].

Notation. For the sake of simplicity, in the sequel n is always even and positive, $n = 2s$, $s \geqq 1$, and all sub-indices of a, b, and c are always considered modulo n, that is $f_{l+n,m+n} = f_{l,m}$ where f stands for a, b, and c.

ALGORITHM 1.

$$\begin{aligned}&\sum_{i+j+k \text{ is even}} (a_{ij} + a_{k+1,i+1})(b_{jk} + b_{i+1,j+1})(c_{ki} + c_{j+1,k+1})\\ &\quad - \sum_{i,k=0}^{n-1} a_{k+1,i+1} \sum_{j:i+j+k \text{ is even}} (b_{jk} + b_{i+1,j+1})c_{ki}\\ &\quad - \sum_{i,j=0}^{n-1} a_{ij}b_{i+1,j+1} \sum_{k:i+j+k \text{ is even}} (c_{ki} + c_{j+1,k+1})\\ &\quad - \sum_{k,j=0}^{n-1} \sum_{i:i+j+k \text{ is even}} (a_{ij} + a_{k+1,i+1})b_{jk}c_{j+1,k+1}\\ &\quad \equiv \sum_{i,j,k} a_{ij}b_{jk}c_{ki}.\end{aligned}$$

It is easy to verify (see the next section) that for any n Algorithm 1 is an LA whose complexity is equal to $n^3/2 + 3n^2$.

Algorithm 1 is faster than the classical one, but not faster than Strassen's. Yet it will be used as a model for improving Strassen's. Now we will analyze the construction of Algorithm 1.

DEFINITIONS. Any product of three nonzero linear forms is a term. If a trilinear form T is written as a sum of M terms, then this gives a decomposition $R = R(T)$ of a given form T whose complexity (norm) $\|R\|$ is equal to $M = M(R)$. In this case R is said to consist of or to include exactly M terms and have a complexity (a norm) $M = \|R\|$. Each LA is a decomposition of a given trilinear form $\sum_{i,j,k} a_{ij}b_{jk}c_{ki}$ as a sum of M terms, where $M = \|\text{LA}\|$ is the complexity of the LA.

Remark 4. It is obvious that (unlike the rank of the tensor of a trilinear form; see [24]) norm $\|R(T)\|$ is not determined just by a given trilinear form T. Strassen's LA and Algorithm 1 give nontrivial examples.

DEFINITION. An LA determined by the trivial decomposition that is given by the identity

$$\sum_{i,j,k} a_{ij}b_{jk}c_{ki} = \sum_{i,j,k} a_{ij}b_{jk}c_{ki}$$

is called trivial and denoted LA(0). $\|\text{LA}(0)\| = n^3$.

In the sequel we seek chains of transformations from the slow LA(0) into a fast LA.

Notation and definitions. Each term $\xi a_{ij}b_{kl}c_{mp}$ where ξ is any number and $0 \leqq i, j, k, l, m, p \leqq n-1$, is called elementary. Each elementary term is either a desirable one, iff it is a term $a_{ij}b_{jk}c_{ki}$ of LA(0), or an undesirable one otherwise. The q given elementary terms $a_{i_rj_r}b_{k_rl_r}c_{m_rp_r}$, $r = 1, \cdots, q$ have a resemblence, if at least one of the following three chains of equalities holds for all r, $r = 1, \cdots, q$:

$$a_{i_rj_r} = a_{ij},$$

$$b_{k_rl_r} = b_{kl},$$

$$c_{m_rp_r} = c_{mp}.$$

If ν of these equalities hold where $0 \leqq \nu \leqq 3$, then the given q terms are said to have ν resemblences or just to be ν-resemble terms. 2-resemble terms will be also called kin. The set of q kin terms will also be called a q-family.

It is obvious that the sum of 2-resemble or 3-resemble terms is always identically either a term, or zero. Thus we can immediately reduce the complexity of a trilinear form if the latter includes a sum of ν-resemble terms where $\nu \geqq 2$. This simple trick will be called uniting of terms. It has been used in Algorithm 1. Generalizing this idea we could partition all the terms of any LA into groups (families) of ν-resemble terms where $\nu \geqq 2$ and then measure the complexity of this LA by the number of such groups. Unfortunately, this trick cannot be directly applied to LA(0) since LA(0) includes no 2-resemble and no 3-resemble terms. However, such terms can be created by forming terms (aggregates) $(a_{ij}+a_{k+1,i+1})(b_{jk}+b_{i+1,j+1})(c_{ki}+c_{j+1,k+1})$ for each triplet i, j, k such that $i+j+k$ is even.

DEFINITION. A term $(a_{ij}+a_{k_1i_1})(b_{jk}+b_{i_1j_1})(c_{ki}+c_{j_1k_1})$ is called an aggregate of two desirable terms $a_{ij}b_{jk}c_{ki}$ and $a_{k_1i_1}b_{i_1j_1}c_{j_1k_1}$, or a 2-aggregate.

It is easy to notice 2-aggregates among the terms of Algorithm 1 if we write $i_1 = i+1$, $j_1 = j+1$, $k_1 = k+1$.

Now Algorithm 1 follows immediately from the obvious identity whose left and right parts are summed for all i, j, k such that $i+j+k$ is even.

$$\begin{aligned}(a_{ij}+a_{k+1,i+1})(b_{jk}+b_{i+1,j+1})(c_{ki}+c_{j+1,k+1})\\ = a_{k+1,i+1}(b_{jk}+b_{i+1,j+1})c_{ki}+a_{ij}b_{i+1,j+1}(c_{ki}+c_{j+1,k+1})+(a_{ij}+a_{k+1,i+1})b_{jk}c_{j+1,k+1}\\ +a_{ij}b_{jk}c_{ki}+a_{k+1,i+1}b_{i+1,j+1}c_{j+1,k+1}.\end{aligned}$$

For each i, j, k the aggregate on the left is decomposed as a sum of two desirable terms and six trivial ones. The latter should cancel all the unnecessary products if the aggregate is expanded. In the sequel such trivial terms will be called correction terms. After summing for i, j, k we notice that the sum of $n^3/2$ aggregates is decomposed into the sum of all n^3 desirable terms and of $3n^3$ correction terms. Then we notice that all the correction terms can be partitioned into $3n^2$ families of kin terms. By uniting the terms within each family we obtain the desired decomposition of LA(0) as a sum of $n^3/2+3n^2$ terms.

4. Short and full 2-procedures.

In this and in the three subsequent sections we will define 2-procedures which generalize the construction of Algorithm 1 and are based on the following simple identities:

$$\begin{aligned}a_{ij}b_{jk}c_{ki}+a_{k_1i_1}b_{i_1j_1}c_{j_1k_1} = (a_{ij}+a_{k_1i_1})(b_{jk}+b_{i_1j_1})(c_{ki}+c_{j_1k_1})\\ -a_{k_1i_1}(b_{i_1j_1}+b_{jk})c_{ki}-a_{ij}b_{i_1j_1}(c_{j_1k_1}+c_{ki})\\ -(a_{k_1i_1}+a_{ij})b_{jk}c_{j_1k_1}.\end{aligned} \tag{3}$$

$$\sum_{i,j,k} a_{k_1i_1}(b_{i_1j_1}+b_{jk})c_{ki}=\sum_{k,i} a_{k_1i_1}c_{ki}\sum_{j}(b_{i_1j_1}+b_{jk});$$

(4)
$$\sum_{i,j,k} a_{ij}b_{i_1j_1}(c_{j_1k_1}+c_{ki})=\sum_{i,j} a_{ij}b_{i_1j_1}\sum_{k}(c_{j_1k_1}+c_{ki});$$

$$\sum_{i,j,k}(a_{k_1i_1}+a_{ij})b_{jk}c_{j_1k_1}=\sum_{j,k} b_{jk}c_{j_1k_1}\sum_{i}(a_{k_1i_1}+a_{ij}).$$

Here $i_1=i_1(i), j_1=j_1(j), k_1=k_1(k)$ are considered given functions of i, j, k (respectively) e.g., $i_1=i+1, j_1=j+1, k_1=k+1$ for Algorithm 1. "Commuting" of symbols under the signs $\sum$ in (4) is purely for notational convenience.

Let $T(ijk)$ denote the term $a_{ij}b_{jk}c_{ki}$ of LA(0), $T^0(ijk)$, $-T^m(ijk)$, $m=1, 2, 3$ denote four terms in the right part of (3). Following the construction and definitions of the previous section we will say that (3) describes aggregating a pair of terms, namely, substituting $T^0(ijk)-\sum_{m=1}^3 T^m(ijk)$ for the sum $T(ijk)+T(i_1j_1k_1)$. Condition (4) describes uniting the correction terms which are kin and belong to the same family of terms. Let $n^3/2$ pairs óf terms of LA(0) be aggregated by applying (3) and all terms in the left parts of these (3) be all different. Then we obtain LA by summing all left parts and all right parts of these identities. Applying (4) to unite 2-resemble (kin) terms of the latter algoritħm we obtain a new LA having the

$$\|\mathrm{LA}\|\leqq\frac{n^3}{2}+3n^2.$$

complexity. Now we are going to formalize this procedure. In the sequel the following notation will be used.

Notation. $|S|$ is the cardinality of a given set S; $S(1)$, $S(2)$, $S(3)$ are three sets: of all integers i, of all pairs of integers (ij) and of all triplets of integers (ijk), such that in all three cases each integer is modulo n.

Now we will define a 2-procedure as a chain of transformations of LA's, starting with forming 2-aggregates from pairs of 0-resemble terms of LA(0). Then we unite kin terms within each family. Equations (3) and (4) are the basic identities for a 2-procedure.

DEFINITION OF 2-PROCEDURE. Define a subset $S\subseteq S(3)$, and partition it into pairs of triplets $(i^rj^rk^r)$ and $(k_1^ri_1^rj_1^r)$, $r=1, 2, \cdots, |S|/2$, such that $i_1^r=i_1(i^r)$, $j_1^r=j_1(j^r)$, $k_1^r=k_1(k^r)$, $r=1, 2, \cdots, |S|/2$, i_1, j_1, k_1 are simple functions on $S(1)$, and $m(i^rj^rk^r)=(k_1^ri_1^rj_1^r)$ is a mapping from a subset $S^1=\{(i^rj^rk^r), r=1, \cdots, |S|/2\}$ of S onto another subset $S^2=\{(k_1^ri_1^rj_1^r), r=1, \cdots, |S|/2\}$ of S, such that $S^2=m(S^1)$, $|S^1|=|S^2|=|S|/2$, $S^1\cup S^2=S$. Construct LA^0 from LA(0) by decomposing 2-aggregates according to (3) for all |S|/2 pairs of desirable terms $T(i^rj^rk^r)$ and $T(k_1^ri_1^rj_1^r)$ for each integer $r\leqq|S|/2$. Transform LA^0 into another LA (which will be denoted LA(1)) by applying (4) to unite kin terms of LA^0 within their families. If (as in Algorithm 1) $S=S(3)$, then a 2-procedure is said to be full. Otherwise, it is said to be short.

LEMMA 1. *For any* LA(1) *constructed by a full* 2*-procedure the following inequalities hold*:

$$\frac{n^3}{2}+\frac{9}{4}n^2\leqq\|\mathrm{LA}(1)\|\leqq\frac{n^3}{2}+3n^2.$$

Proof.

$$\sum_{(ijk)\in S(3)}T(ijk)=\sum_{(ijk)\in S^1}T(ijk)+\sum_{(ijk)\in S^2}T(ijk)=\sum_{(ijk)\in S^1}(T(ijk)+T(k_1, i_1, j_1))$$

since $S^1\cup S^2=S=S(3)$, $S^1\cap S^2$ is empty (since $|S^1|=|S^2|=|S|/2)$, $S^2=m(S^1)$, $(k_1i_1j_1)=m(ijk)$. Now the right inequality has been proven by applications of (3) and (4). The left inequality is proven in the next section.

5. Lower bound on the complexity of LA resulting from a full 2-procedure. Except for the lower bound itself, the technique and the results of these sections are not referred to in the sequel. Each 2-procedure is determined by the choice of two sets: $S \subseteq S(3)$ and $S^1 \subset S$ and by a mapping $m(ijk) = (k_1 i_1 j_1)$. A set S^1 and a mapping m can be defined by a partition of a chosen set S. This construction will be used for establishing the desirable lower bound. It could also be used to generalize 2-procedures.

Notation and definitions. S is an abstract set; $|S| = pq$, p, q are integers. A correspondence between pq elements of S and the squares of a $p \times q$ table is said to be a $p \times q$ representation of S (there are $(pq)!$ different ones). Assuming a $p \times q$ representation of S given, a point of S is denoted (g, h) iff it corresponds with the square (g, h) of the table. The representation determines two partitions of S into p row-subsets S^1, $S^2, \cdots, S^p$ and q column-subsets $S_1, S_2, \cdots, S_q$, such that $|S^V| = q, |S_r| = p, v = 1, \cdots, p, r = 1, \cdots, q$. Also the representation determines the following mappings:

$$m^r(S^V) = m^r(v, r) = (v, 1) \quad \text{from } S^V \text{ to } S^1 \text{ and}$$

$$m_v(S_r) = m_v(v, r) = (1, r) \quad \text{from } S_r \text{ to } S_1,$$

where $v = 1, \cdots, p, r = 1, \cdots, q$. If $\hat{S} \subseteq S(3)$, then $P_1(\hat{S}), P_2(\hat{S}), P_3(\hat{S})$ are the projections of $\hat{S}$ onto the coordinate planes (ki), (ij), (jk).

Now in a way similar to the proof of the right inequality of Lemma 1 given in § 4 we obtain the following result.

LEMMA 2. *For any* LA(1) *constructed by a full* 2-*procedure*

$$\|\text{LA}(1)\| = \frac{n^3}{2} + |P_1(S^1)| + |P_2(S^1)| + |P_3(S^1)|.$$

Now the desired lower bound $n^3/2 + \frac{9}{4}n^2$ on $\|\text{LA}(1)\|$ follows immediately from Lemma 2 and from the next lemma.

LEMMA 3. $|P_1(S^1)| + |P_2(S^1)| + |P_3(S^1)| \geqq \frac{9}{4}n^2$ *for any partition of* $S = S(3)$ *by* $2 \times (n^3/2)$ *representation, such that* $m_2(ijk) = (\tilde{k}(k), \tilde{\imath}(i), \tilde{j}(j)), \tilde{k}(k), \tilde{\imath}(i), \tilde{j}(j)$ *are simple functions on* $S(1)$.

Sketch of the proof of Lemma 3. We will use a duality between S^1 and S^2. If a pair $(k_0 i_0) \notin P_1(S^1)$, then all the triplets $(i_0 j k_0)$, $j = 0, 1, \cdots, n-1$ do not belong to S^1; thus they belong to $\tilde{S}^1 = S^2$, and $P_2(S^2) \ni (i_0 j), j = 0, 1, \cdots, n-1$, $P_3(S^2) \ni (jk_0), j = 0, 1, \cdots, n-1$. Similar properties hold if a pair $(ki) \notin P_1(S^2)$. Thus the minimum of $|P_1(S^1)| + |P_2(S^1)| + |P_3(S^1)|$ is achieved if $S^2 = S \backslash S^1$ and if $S^1 = S_3^1(p, q) \cup S_3^2(q, r) \cup S_3^3(r, p)$, $S_3^1(p, q) = \{(ijk) : i \leqq q, j \text{ arbitrary}, k \leqq p\}$, $S_3^2(q, r) = \{(ijk) : i \text{ arbitrary}, j \leqq r, k \leqq p\}$, $S_3^3(r, p) = \{(ijk) : i \leqq q, j \leqq r, k \text{ arbitrary}\}$. Then $p = q = r$, since $m_2(S^1) = S^2$, and therefore, $p = q = r = n/2 - 1$, since $|S^1| = |S^2|$.

6. Optimal full 2-procedure. Further improvement of LA by applying a short 2-procedure. In this section two examples of optimal full 2-procedure are exhibited such that the resulting LA have the complexity $\|\text{LA}\| = n^3/2 + \frac{9}{4}n^2$.

Notation. (S_1, S_2) and (S_1, S_2, S_3) are the Cartesian products of sets S_1, S_2 and S_1, S_2, S_3. $E \subset S(1)$ and $O \subset S(1)$ are two subsets of $S(1)$ consisting of all even and of all odd integers modulo n. P_2^1 is $S(2) \backslash (O, O) = (E, E) \cup (E, O) \cup (O, E)$, $\tilde{P}_1(fg)$ is $S(1)$, if $(fg) \in (E, E)$ and it is E, if $(fg) \in S(2) \backslash (E, E) = (O, O) \cup (E, O) \cup (O, E)$. $S^1 = (E, E, E) \cup (E, E, O) \cup (E, O, E) \cup (O, E, E)$, $(k_1, i_1, j_1) = (k+1, i+1, j+1)$ (here $S = S(3)$). Here and hereafter all indices i, j, k are modulo n, e.g. $(i+n, j, k) = (ijk)$, $T(i, j+n, k) = T(ijk)$ etc.

Now with this choice of S^1 we apply the full 2-procedure and obtain a desirable LA such that $\|\text{LA}\| = n^3/2 + \frac{9}{4}n^2$.

For each triplet $(ijk) \in S^1$ the terms $T(ijk)$ and $T(k_1 i_1 j_1) = T(k+1, i+1, j+1)$ are aggregated by applying (3), all the left parts and separately all the right parts of these identities (3) for $(ijk) \in S^1$ are summed and all terms in the right part of the resulting identity are partitioned into groups of kin terms. The latter are united by applying (4). It is easy to verify that this gives an LA whose complexity is $n^3/2 + \frac{9}{4}n^2$ (optimal within the class of all LA obtained from LA(0) by this procedure). Here is a formal presentation of this LA.

ALGORITHM 2.

$$T^0 = \sum_{(i,j,k) \in S^1} (a_{ij} + a_{k+1,i+1})(b_{jk} + b_{i+1,j+1})(c_{ki} + c_{j+1,k+1}),$$

$$T^1 = \sum_{(k,i) \in P_2^1} a_{k+1,i+1} \sum_{j \in \tilde{P}_1(ki)} (b_{jk} + b_{i+1,j+1}) c_{ki},$$

$$T^2 = \sum_{(i,j) \in P_2^1} a_{ij} b_{i+1,j+1} \sum_{k \in \tilde{P}_1(ij)} (c_{ij} + c_{i+1,k+1}),$$

$$T^3 = \sum_{(j,k) \in P_2^1} \sum_{i \in \tilde{P}_1(jk)} (a_{ij} + a_{k+1,i+1}) b_{jk} c_{j+1,k+1},$$

$$T^0 - T^1 - T^2 - T^3 = \sum_{j,i,k=0}^{n-1} a_{ij} b_{jk} c_{ki}.$$

Exercise 1. Let $(k_1, i_1, j_1) = m(i, j, k) = (k+s, i+s, j+s)$, where $n = 2s$. Let S^1 be equal to the set of all triplets of integers modulo n such that at least two integers in each triplet are less than s. Repeat the above described procedure to construct another LA of the complexity $n^3/2 + \frac{9}{4}n^2$. *What will substitute for* P_2^1, $\tilde{P}_1(fg)$?

By virtue of Lemma 1, we must change or modify the procedure of constructing LA from LA(0) to reduce $\|\text{LA}\|$ further. Here is an example of such a modification via a short 2-procedure, rather than full one. It results in a slight improvement of Algorithm 2.

Let all the terms of LA(0), but the terms $T(i, i+1, i+2)$, $(T(i+1, i+2, i)$, $T(i+2, i, i+1)$, $i = 0, 1, \cdots, n-1$, be aggregated by applying (3) and then be summed. In other words, let $R^0 = R^0(T^0)$ be a decomposition of a trilinear form T^0 as the sum $\sum_{(ijk) \in \tilde{S}} T^0(ijk)$ where $\tilde{S} = S^1 \backslash \tilde{S}^0$, $\tilde{S}^0$ consists of all the triplets $(i, i+1, i+2)$, $(i+1, i+2, i)$, $(i+2, i, i+1)$ where $i \in E$. Then $\|R^0\| = n^3/2 - 3n/2$, rather than $n^3/2$, but $3n$ terms $T(ijk)$ are missed in the sum of the left parts of (3). Yet this can be fixed by a special uniting procedure (different from (4)), since each missed term belongs to a family of kin terms in the decompositions of T^1, T^2 or T^3. Here is this modified version 2a of algorithm 2 whose complexity is equal to $n^3/2 - 3n/2 + \frac{9}{4}n^2$ for any even n.

$$T^0 = \sum_{(ijk) \in \tilde{S}} (a_{ij} + a_{k+1,i+1})(b_{jk} + b_{i+1,j+1})(c_{ki} + c_{j+1,k+1}),$$

$$T^1 = \sum_{(ki) \in P_2^1} a_{k+1,i+1} \left[\sum_{j \in \tilde{P}_1(k,\, i)} (b_{jk} + b_{i+1,j+1}) \sigma_{ijk} - \delta_{i,k+1} b_{i+1,k} \right] c_{ki},$$

$$T^2 = \sum_{(ij) \in P_2^1} a_{ij} b_{i+1,j+1} \left[\sum_{k \in \tilde{P}_1(i,j)} (c_{ki} + c_{j+1,k+1}) \sigma_{ijk} - \delta_{j,i+1} c_{j+1,i} \right],$$

$$T^3 = \sum_{(jk) \in P_2^1} \left[\sum_{i \in \tilde{P}_1(j,k)} (a_{ij} + a_{k+1,i+1}) \sigma_{ijk} - \delta_{k,j+1} a_{k+1,j} \right] b_{jk} c_{j+1,k+1},$$

$$T^0 - T^1 - T^2 - T^3 = \sum_{i,j,k=0}^{n-1} a_{ij} b_{jk} c_{ki}.$$

Here

$$\sigma_{ijk}=1-\delta_{i+2,k}\delta_{i+1,j}-\delta_{i+1,j}\delta_{i-1,k}-\delta_{i-2,j}\delta_{i-1,k},$$

$$\delta_{lm}=\begin{cases}0 & \text{if } l\neq m,\\ 1 & \text{if } l=m.\end{cases}$$

Exercise 2. Reduce by $3n/2$ the complexity of the LA constructed in Exercise 1 by excluding $3n/2$ elements from S^1.

7. Short 3-procedure for constructing fast LA. A 3-procedure can be easily derived from a 2-procedure. Indeed, we define an aggregate of 3 desirable terms, or just a 3-aggregate, similar to a 2-aggregate, that is, hereafter $(a_{ij}+a_{j_1k_1}+a_{k_2i_2})(b_{jk}+b_{k_1i_1}+b_{i_2j_2})(c_{k_i}+c_{i_1j_1}+c_{j_2k_2})$ is called the aggregate of three terms $T(ijk)$, $T(j_1k_1i_1)$ and $T(k_2i_2j_2)$, or just a 3-aggregate, for any choice of triplets (ijk), $(j_1k_1i_1)$ and $(k_2i_2j_2)$. Now let a subset S of the set of all n^3 desirable terms be partitioned into triplets of terms, and let the aggregates of all triplets be summed and expanded. Then let the desirable terms not belonging to S be added, and let them and the terms in the expansion be united within their families. As a result we have a 3-procedure which is called short if the original subset S of terms includes less than n^3 terms.

Now we present a concrete short 3-procedure resulting in an LA such that $\|\mathrm{LA}\|=(n^3-4n)/3+6n^2$. We start with rewriting the original trilinear from $F=\sum_{i,j,k=0}^{n-1}T(ijk)$ as a sum of eight forms:

$$\begin{aligned}F=F(0,0,0)&+F(0,0,1)+F(0,1,0)+F(1,0,0)\\&+F(1,1,1)+F(1,1,0)+F(1,0,1)+F(0,1,1),\end{aligned}$$

where

$$\begin{aligned}F(\alpha,\beta,\gamma)=\sum_{(i,j,k)\in S^1(s)}&(T(i+\alpha s,j+\beta s,k+\gamma s)+T(j+\gamma s,k+\alpha s,i+\beta s)\\&+T(k+\beta s,i+\gamma s,j+\alpha s))+\sum_{i=0}^{s-1}T(i+\alpha s,i+\beta s,i+\gamma s),\end{aligned}$$

each of α, β, γ is either 0, or 1, $n=2s$, and $S^1(s)$ is a subset of $S(3)$, characterized by the following property: $|S^1(s)|=(s^3-s)/3$, and therefore, $\|F(\alpha,\beta,\gamma)\|=s^3$ for any 0–1 triplet (α,β,γ), or equivalently, since $\|F\|=n^3=8s^3$, the presented decompositions of $F(\alpha,\beta,\gamma)$ include no common terms for different triplets (α,β,γ).

We will define $S^1(s)$ in § 10 by formula (5). Now we will focus on the following decomposition of $F(0,0,0)$.

$$\begin{aligned}F(0,0,0)=\sum_{i=0}^{s-1}a_{ii}b_{ii}c_{ii}&\\+\sum_{(ijk)\in S^1(s)}&(a_{ij}+a_{jk}+a_{ki})(b_{jk}+b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk})+R(0,0,0)\end{aligned}$$

where $S^1(s)$ is a subset of $S(3)$ which will be defined in the sequel, $R(0,0,0)$ is a sum of correction terms which should cancel all the cross-point products except desirable terms. The latter arise when the 3-aggregates in this decomposition are expanded.

Notice that the given decomposition represents a short 3-procedure (without uniting yet) applied to all terms of $F(0,0,0)$ except the terms $T(iii)$, $i=0,1,\cdots,s-1$. In this 3-procedure $i=i_1=i_2$, $j=j_1=j_2$, $k=k_1=k_2$. It is easy to verify that (i,j,k) and (j,k,i), or (i,j,k) and (k,i,j), or (j,k,i) and (k,i,j) coincide only if $i=j=k$. Therefore, each 3-aggregate in the decomposition of $F(0, 0, 0)$ is the aggregate of 3 different terms. It is obvious that we can choose a subset $S^1(s)$ of $S(3)$ such that $|S^1(s)|=(s^3-s)/3$, and each pair of different aggregates is derived from 6 different terms of $F(0, 0, 0)$ (one of such subsets is determined in the sequel by formula (5)). Thus we have a decomposition of $F(0, 0, 0)$ consisting of $(s^3-s)/3$ aggregates, of s terms $T(iii)$, and of correction terms.

8. Two groups of correction terms for 3-procedure. We seek to reduce the number of correction terms. We will write their sum using the following formulae.

$$R(0,0,0)=R^1(0,0,0)+R^2(0,0,0),$$

$$R^1(0,0,0)=\sum_{(ijk)\in S^1(s)}(a_{ij}b_{ki}c_{jk}+a_{jk}b_{ij}c_{ki}+a_{ki}b_{jk}c_{ij}),$$

$$\begin{aligned}R^2(0,0,0)=\sum_{(ijk)\in S^1(s)}&[a_{ij}b_{ij}(c_{ki}+c_{ij}+c_{jk})+a_{jk}b_{jk}(c_{ki}+c_{ij}+c_{jk})\\&+a_{ki}b_{ki}(c_{ki}+c_{ij}+c_{jk})+a_{ij}(b_{jk}+b_{ki})c_{ij}+a_{jk}(b_{ki}+b_{ij})c_{jk}\\&+a_{ki}(b_{jk}+b_{ij})c_{ki}\\&+(a_{jk}+a_{ki})b_{ij}c_{ij}+(a_{ij}+a_{ki})b_{jk}c_{jk}+(a_{ij}+a_{jk})b_{ki}c_{ki}].\end{aligned}$$

$R^2(0, 0, 0)$ consists of correction terms (which are analogous to correction terms of Algorithms 1, 2 and 2a). It is easy to verify that altogether with the terms $T(iii)$ for $i=0, 1, \cdots, s-1$ they form only $3s^2$ families of kin terms. Unfortunately, $\|R^1(0, 0, 0)\|=s^3-s$, and there are no kin terms among the terms of $R^1(0, 0, 0)$. We should seek another trick, besides uniting. We will find it via defining 3-procedures applied to the terms of eight forms $F(\alpha, \beta, \gamma)$'(each defined by a 0–1 triplet α, β, γ), such that all the terms of $R^1(\alpha, \beta, \gamma)$ will cancel themselves out. This can be considered the crucial step in our construction.

9. Table representation for aggregates, desirable and correction terms. Trilinear canceling. In order to make our canceling procedure more observable, we will represent each 3-aggregate as a 3×3 table by writing

$$(a_{ij}+a_{jk}+a_{ki})(b_{jk}+b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk})$$

as

$$\begin{pmatrix}a_{ij}+a_{jk}+a_{ki}\\ b_{jk}+b_{ki}+b_{ij}\\ c_{ki}+c_{ij}+c_{jk}\end{pmatrix}.$$

Thus the table represents a 3-aggregate derived from three desirable terms. Each term is the product of three entries of the same column of the table. We will say that such a table defines the 3-aggregate and is defined if a 3-aggregate is given. Now assume that some of the pluses in the table are substituted by minuses. Then the table defines a 3-aggregate with minuses, that is a product

$$(\pm a_{ij}\pm a_{jk}\pm a_{ki})(\pm b_{jk}\pm b_{ki}\pm b_{ij})(\pm c_{ki}\pm c_{ij}\pm c_{jk}).$$

We will require that there always be even number of minuses in each column of the table (either no, or two minuses). This guarantees that three columns of the table define a 3-aggregate derived from 3 desirable terms (each with the sign plus).

Now we will generalize this representation of 3-aggregates even further, so that it could define 3-procedure for the terms of all $F(\alpha, \beta, \gamma)$. Hereafter we will denote

$$\bar{i} = i(1) = i + s;\ \bar{j} = j(1) = j + s,\ \bar{k} = k(1) = k + s;\ i(0) = i;\ j(0) = j;\ k(0) = k,$$

where

$$i, j, k = 0, 1, \cdots, s-1.$$

We see that $i(\alpha)$ is defined by i and α, $i(\beta)$ is defined by i and $\beta, \cdots, k(\gamma)$ is defined by k and γ. For each $F(\alpha, \beta, \gamma)$ we will restrict ourselves to 3-procedures in which each 3-aggregate is defined by a 3×3 table of the following kind:

$$\begin{pmatrix} \pm a_{i(\alpha)j(\beta)} \pm a_{j(\gamma)k(\alpha)} \pm a_{k(\beta)i(\gamma)} \\ \pm b_{j(\beta)k(\gamma)} \pm b_{k(\alpha)i(\beta)} \pm b_{i(\gamma)j(\alpha)} \\ \pm c_{k(\gamma)i(\alpha)} \pm c_{i(\beta)j(\gamma)} + c_{j(\alpha)k(\beta)} \end{pmatrix}$$

While summing the aggregates for each of 8 triplets α, β, γ, we will assume that again $(ijk) \in S^1(s)$.

Keeping these rules for 3-procedures and considering the case $\alpha = \beta = \gamma = 0$ as a model for the cases of any 0-1 triplets α, β, γ we obtain the following estimates of the total numbers: $8(s^3 - s)/3$ for aggregates and eight $(3s^2) = 24s^2$ for different families of kin terms among the terms $T(i(\alpha), i(\beta), i(\gamma))$, $T(i(\beta), i(\gamma), i(\alpha))$, $T(i(\gamma), i(\alpha), i(\beta))$ and the terms of $R^2(\alpha, \beta, \gamma)$ (the latter denotes the second group of correction terms, analogous to $R^2(0, 0, 0)$).

As in the tables for the terms of $F(0, 0, 0)$ we will keep the rule for all our tables that there should always be even number (0 or 2) of minuses in each column of the table.

Now we notice that each 3×3 table also represents three terms of $\mathrm{R}^1(\alpha, \beta, \gamma)$. Namely, each of these three terms is either a product of three diagonal entries

$$(\pm a_{i(\alpha)j(\beta)})(\pm b_{k(\alpha)i(\beta)})(\pm c_{j(\alpha)k(\beta)}),$$

or it is

$$(\pm a_{k(\beta)i(\gamma)})(\pm b_{j(\beta)k(\gamma)})(\pm c_{i(\beta)j(\gamma)}),$$

or it is

$$(\pm a_{j(\gamma)k(\alpha)})(\pm b_{i(\gamma)j(\alpha)})(\pm c_{k(\gamma)i(\alpha)}).$$

Taking this into account we present the following desired distribution of pluses and minuses in the tables for all $F(\alpha, \beta, \gamma)$. (See Tables 1–8.)

TABLE 1

$F(0, 0, 0)$

$$\begin{pmatrix} a_{ij} + a_{jk} + a_{ki} \\ b_{jk} + b_{ki} + b_{ij} \\ c_{ki} + c_{ij} + c_{jk} \end{pmatrix}$$

TABLE 2

$F(1, 1, 1)$

$$\begin{pmatrix} a_{\bar{i}\bar{j}} + a_{\bar{j}\bar{k}} + a_{\bar{k}\bar{i}} \\ b_{\bar{j}\bar{k}} + b_{\bar{k}\bar{i}} + b_{\bar{i}\bar{j}} \\ c_{\bar{k}\bar{i}} + c_{\bar{i}\bar{j}} + c_{\bar{j}\bar{k}} \end{pmatrix}$$

TABLE 3

$F(0, 0, 1)$

$$\begin{pmatrix} -a_{ij} + a_{\bar{j}k} + a_{k\bar{i}} \\ b_{j\bar{k}} + b_{ki} + b_{\bar{i}j} \\ -c_{\bar{k}i} + c_{i\bar{j}} + c_{jk} \end{pmatrix}$$

TABLE 4

$F(1, 1, 0)$

$$\begin{pmatrix} -a_{\bar{i}\bar{j}} + a_{j\bar{k}} + a_{\bar{k}i} \\ b_{\bar{j}k} + b_{\bar{k}\bar{i}} + b_{i\bar{j}} \\ -c_{k\bar{i}} + c_{\bar{i}j} + c_{\bar{j}\bar{k}} \end{pmatrix}$$

TABLE 5

F(0, 1, 0)

$$\begin{pmatrix} a_{i\bar{j}}-a_{jk}+a_{\bar{k}i} \\ b_{\bar{j}k}+b_{k\bar{i}}+b_{ij} \\ c_{ki}-c_{\bar{i}j}+c_{j\bar{k}} \end{pmatrix}$$

TABLE 6

F(1, 0, 1)

$$\begin{pmatrix} a_{\bar{i}j}-a_{\bar{j}\bar{k}}+a_{k\bar{i}} \\ b_{j\bar{k}}+b_{\bar{k}i}+b_{\bar{i}\bar{j}} \\ c_{\bar{k}\bar{i}}-c_{i\bar{j}}+c_{\bar{j}k} \end{pmatrix}$$

TABLE 7

$F(1, 0, 0)$

$$\begin{pmatrix} a_{\bar{i}j}+a_{j\bar{k}}-a_{ki} \\ b_{jk}+b_{\bar{k}i}+b_{i\bar{j}} \\ c_{k\bar{i}}+c_{ij}-c_{\bar{j}k} \end{pmatrix}$$

TABLE 8

$F(0, 1, 1)$

$$\begin{pmatrix} a_{i\bar{j}}+a_{\bar{j}k}-a_{\bar{k}\bar{i}} \\ b_{\bar{j}\bar{k}}+b_{k\bar{i}}+b_{\bar{i}j} \\ c_{\bar{k}i}+c_{\bar{i}\bar{j}}-c_{j\bar{k}} \end{pmatrix}$$

It remains to notice that each term of $\tilde{R}^1=\sum_{\alpha,\beta,\gamma}R^1(\alpha, \beta, \gamma)$ appears in our tables (and therefore in $\tilde{R}^1$) exactly twice: once with plus and another time with minus. It may be of interest to notice that we could do with just twelve minuses in all eight tables.

10. An estimate for the number of terms (of multiplications) in the algorithm. We leave as an exercise for a reader to verify our estimates for the numbers of aggregates and families, or just to check the whole algorithm which is formally presented in the next section and differs only slightly from the one defined by our tables. The set $S^1(s)$ can be chosen the same for both algorithms. For example, it can be determined by the following formulae.

$$(5)\qquad \begin{aligned} &S^1(s)=S_1^1(s)\cup S_2^1(s), \qquad S_1^1(s)=\{(i, j, k), 0\leqq i\leqq j<k\leqslant s-1\},\\ &S_2^1(s)=\{(i, j, k), 0\leqq k<j\leqq i\leqq s-1\}.\end{aligned}$$

Counting the number of terms which remain after uniting and canceling we estimate that we constructed LA such that $\|\mathrm{LA}\|=8(\mathrm{s}^3-\mathrm{s})/3+24\mathrm{s}^2=(\mathrm{n}^3-4\mathrm{n})/3+6\mathrm{n}^2$.

Exercise 3. Construct a fast LA, such that $\|\mathrm{LA}\|=(n^3-4n)/3+6n^2$ by using a similar procedure and by substituting $i^*=i+1$, $j^*=j+1$, $k^*=k+1$ for $\bar{i}$, $\bar{j}$, $\bar{k}$, and $S^*=\{(i, j, k),\ i, j, k \text{ are even}\}$ for $\tilde{S}(s)$.

11. Formal description of an asymptotically fast LA.

ALGORITHM 3.

$$\begin{aligned} T^0=\sum_{(i,j,k)\in S^1(s)}\Big[&(a_{ij}+a_{jk}+a_{ki})(b_{jk}+b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk})\\ &-(a_{ij}-a_{\bar{j}k}+a_{k\bar{i}})(b_{j\bar{k}}+b_{ki}-b_{\bar{i}j})\,(-c_{\bar{k}i}+c_{i\bar{j}}+c_{ik})\\ &-(-a_{\bar{i}j}+a_{j\bar{k}}+a_{ki})(b_{jk}-b_{\bar{k}i}+b_{i\bar{j}})\,(c_{k\bar{i}}+c_{ij}-c_{\bar{j}k})\\ &-(a_{i\bar{j}}+a_{jk}-a_{\bar{k}i})(-b_{\bar{j}k}+b_{k\bar{i}}+b_{ij})\,(c_{ki}-c_{\bar{i}j}+c_{j\bar{k}})\\ &-(a_{\bar{i}j}+a_{\bar{j}\bar{k}}-a_{k\bar{i}})(-b_{j\bar{k}}+b_{\bar{k}i}+b_{\bar{i}\bar{j}})(c_{\bar{k}\bar{i}}-c_{i\bar{j}}+c_{\bar{j}k})\\ &-(-a_{i\bar{j}}+a_{\bar{j}k}+a_{\bar{k}\bar{i}})(b_{\bar{j}\bar{k}}-b_{k\bar{i}}+b_{\bar{i}j})(c_{\bar{k}_i}+c_{\bar{i}\bar{j}}-c_{j\bar{k}})\\ &-(a_{\bar{i}\bar{j}}-a_{j\bar{k}}+a_{\bar{k}i})(b_{\bar{j}k}+b_{\bar{k}\bar{i}}-b_{i\bar{j}})(-c_{k\bar{i}}+c_{\bar{i}j}+c_{\bar{j}\bar{k}})\\ &+(a_{\bar{i}\bar{j}}+a_{\bar{j}\bar{k}}+a_{\bar{k}\bar{i}})(b_{\bar{j}\bar{k}}+b_{\bar{k}\bar{i}}+b_{\bar{i}\bar{j}})(c_{\bar{k}\bar{i}}+c_{\bar{i}\bar{j}}+c_{\bar{j}\bar{k}})\Big],\end{aligned}$$

$$
\begin{aligned}
T^1 = \sum_{i,j=0}^{s-1} \Bigg\{ & a_{ij}b_{ij}\Bigg[(s-2\delta_{ij})c_{ij} + \sum_{k=0}^{s-1}{}^{*} (c_{ki}+c_{jk})\Bigg] \\
& + a_{ij}b_{\bar{i}\bar{j}}\Bigg[(s-\delta_{ij})c_{i\bar{j}} + \sum_{k=0}^{s-1}{}^{*} (-c_{\bar{k}i}+c_{jk})\Bigg] \\
& + a_{\bar{i}j}b_{i\bar{j}}\Bigg[(s-\delta_{ij})c_{ij} - \delta_{ji}c_{\bar{j}\bar{i}} + \sum_{k=0}^{\delta-1}{}^{*} (c_{k\bar{i}}-c_{\bar{j}k})\Bigg] \\
& + a_{i\bar{j}}b_{ij}\Bigg[(s-\delta_{ij})c_{\bar{i}j} - \sum_{k=0}^{s-1}{}^{*} (c_{ki}+c_{j\bar{k}})\Bigg] \\
& + a_{\bar{i}j}b_{\bar{i}\bar{j}}\Bigg[(s-\delta_{ij})c_{i\bar{j}} - \sum_{k=0}^{s-1}{}^{*} (c_{\bar{k}\bar{i}}+c_{\bar{j}k})\Bigg] \\
& + a_{i\bar{j}}b_{\bar{i}j}\Bigg[(s-\delta_{ij})c_{\bar{i}\bar{j}} - \delta_{\bar{j}\bar{i}}c_{ji} + \sum_{k=0}^{s-1}{}^{*} (c_{\bar{k}i}-c_{j\bar{k}})\Bigg] \\
& + a_{\bar{i}\bar{j}}b_{i\bar{j}}\Bigg[(s-\delta_{ij})c_{\bar{i}j} + \sum_{k=0}^{s-1}{}^{*} (-c_{k\bar{i}}+c_{\bar{j}\bar{k}})\Bigg] \\
& + a_{\bar{i}\bar{j}}b_{\bar{i}\bar{j}}\Bigg[(s-2\delta_{ij})c_{\bar{i}\bar{j}} + \sum_{k=0}^{s-1}{}^{*} (c_{\bar{k}\bar{i}}+c_{\bar{j}\bar{k}})\Bigg]\Bigg\},
\end{aligned}
$$

$$
\begin{aligned}
T^2 = \sum_{i,j=0}^{s-1} \Bigg\{ & a_{ij} \sum_{k=0}^{s-1}{}^{*} (b_{ki}+b_{jk})c_{ij} - a_{ij} \sum_{k=0}^{s-1}{}^{*} (b_{ki}+b_{j\bar{k}})c_{i\bar{j}} \\
& + a_{\bar{i}j} \sum_{k=0}^{s-1}{}^{*} (b_{jk}-b_{\bar{k}i})c_{ij} + a_{i\bar{j}} \Bigg[\sum_{k=0}^{s-1}{}^{*} (b_{k\bar{i}}-b_{\bar{j}k}) - \delta_{ji}b_{\bar{j}\bar{i}}\Bigg]c_{\bar{i}j} \\
& + a_{\bar{i}j}\Bigg[\sum_{k=0}^{s-1}{}^{*} (b_{\bar{k}i}-b_{j\bar{k}}) - \delta_{\bar{j}\bar{i}}b_{ji}\Bigg]c_{i\bar{j}} + a_{i\bar{j}} \sum_{k=0}^{s-1}{}^{*} (b_{\bar{j}\bar{k}}-b_{k\bar{i}})c_{\bar{i}\bar{j}} \\
& - a_{\bar{i}\bar{j}} \sum_{k=0}^{s-1}{}^{*} (b_{\bar{k}\bar{i}}+b_{\bar{j}k})c_{\bar{i}j} + a_{\bar{i}\bar{j}} \sum_{k=0}^{s-1}{}^{*} (b_{\bar{k}\bar{i}}+b_{\bar{j}\bar{k}})c_{\bar{i}\bar{j}}\Bigg\},
\end{aligned}
$$

$$
\begin{aligned}
T^3 = \sum_{i,j=0}^{s-1} \Bigg\{ & \sum_{k=0}^{s-1}{}^{*} (a_{ki}+a_{jk})b_{ij}c_{ij} + \Bigg[\sum_{k=0}^{s-1}{}^{*} (a_{k\bar{i}}-a_{\bar{j}k}) - \delta_{ji}a_{\bar{j}\bar{i}}\Bigg]b_{\bar{i}j}c_{i\bar{j}} \\
& - \sum_{k=0}^{s-1}{}^{*} (a_{ki}+a_{j\bar{k}})b_{i\bar{j}}c_{ij} + \sum_{k=0}^{s-1}{}^{*} (a_{jk}-a_{\bar{k}i})b_{ij}c_{\bar{i}j} \\
& + \sum_{k=0}^{s-1}{}^{*} (a_{\bar{j}\bar{k}}-a_{k\bar{i}})b_{\bar{i}\bar{j}}c_{i\bar{j}} - \sum_{k=0}^{s-1}{}^{*} (a_{\bar{k}\bar{i}}+a_{\bar{j}k})b_{\bar{i}j}c_{\bar{i}\bar{j}} \\
& + \Bigg[\sum_{k=0}^{s-1}{}^{*} (a_{\bar{k}i}-a_{j\bar{k}}) - \delta_{\bar{j}\bar{i}}a_{ji}\Bigg] b_{i\bar{j}}c_{\bar{i}j} + \sum_{k=0}^{s-1}{}^{*} (a_{\bar{k}\bar{i}}+a_{\bar{j}\bar{k}})b_{\bar{i}\bar{j}}c_{\bar{i}\bar{j}}\Bigg\},
\end{aligned}
$$

$$\sum_{i,j,k=0}^{n} a_{ij}b_{jk}c_{jk} = T^0 - T^1 - T^2 - T^3.$$

Here $n=2s$, $\bar{i}=i+s$, $\bar{j}=j+s$, $\bar{k}=k+s$, S^1 (s) is determined by (5), δ_{pq} is Kronecker's symbol:

$$\delta_{pq} = \begin{cases} 1 & \text{if } p=q, \\ 0 & \text{if } p \neq q, \end{cases}$$

the symbol $\sum_{k=0}^{s-1}{}^*$ is equivalent with the symbol $\sum_{k=0}^{s-1}$ for $i \neq j$ and with the symbol $\sum_{k=0,k\neq i}^{s-1}$ for $i = j$.

THEOREM 3. *For any even n there exists* LA *having the complexity* $(n^3-4n)/3+6n^2$.

Proof. See Algorithm 3.

12. Trilinear aggregating. In the previous sections we decomposed $\sum_{i,j,k=0}^{n-1} T(ijk)$ as a sum of $(n^3-4n)/3$ 3-aggregates and of

$$F^* = \sum_{\alpha,\beta,\gamma} \left\{ \sum_{i=0}^{s-1} T(i+\alpha s, i+\beta s, i+\gamma s) + R^2(\alpha, \beta, \gamma) \right\}.$$

The latter form F^* was in turn decomposed as a sum of $6n^2$ terms which resulted in a fast LA. In the next sections we will reduce the number of terms in F^* to $\frac{9}{2}\text{n}^2+\text{n}$. This will result in a further speed-up of the algorithms.

In order to find such a decomposition of F^*, we need to generalize our definition of aggregates so that an aggregate could be derived from any set of terms not necessarily from desirable ones. In particular, we will form aggregates from pairs of 1-resemble terms. This is a straightforward generalization of the previous definitions so that the readers may skip all this section except formula (7) unless they are interested in a more systematic study of aggregatings as a general class of transformations of trilinear forms. We will see that this class also includes the uniting of terms and the forming of 2-aggregates and 3-aggregates from desirable terms, that is the operations already studied in this paper. We will establish the relationship between the norms of two decompositions of a trilinear form before and after the aggregating of its terms. Finally, we will show that aggregating is a very general kind of transformation of any trilinear form.

Now we will generalize our definitions of resemblence, aggregates and related concepts.

DEFINITIONS. A triplet of linear forms $L^1 = L^1(A)$, $L^2 = L^2(B)$, $L^3 = L^3(C)$ is a (nonunique) representation of the term $L^1L^2L^3$. A set $\hat{T}$ of q terms $T_1, T_2, \cdots, T_q$ has a resemblence if a representation L_m^1, L_m^2, L_m^3 is given for each term T_m (for $m = 1, 2, \cdots, q$) and if $L_1^i = L_2^i = \cdots = L_q^i$ for at least one integer i such that $1 \leqq i \leqq 3$. If these equalities hold for ν integers i, each between 1 and 3 (here $0 \leqq \nu \leqq 3$), then a given set $\tilde{T}$ has ν resemblences, and $T_1, \cdots, T_q$ are called ν-resemble terms (also kin if $\nu = 2$). A set of kin terms is a family. If q triplets L_m^1, L_m^2, L_m^3, $m = 1, 2, \cdots, q$, of linear forms are given, then $T(q) = \prod_{s=1}^3 \sum_{m=1}^q L_m^s$ is a trilinear aggregate of the set $\tilde{T} = (T_1, \cdots, T_q)$, $T_m = L_m^1 L_m^2 L_m^3$ (or, for the sake of simplicity, is just a q-aggregate). The substitution of $T(q) - \sum_{t=q+1}^{q^3} T_t$ for $\sum_{m=1}^q T_m$ is trilinear aggregating of the set $\tilde{T}$. Here T_t, $t = 1, \cdots, q^3$, are all the terms $L_m^1 L_p^2 L_r^3$ such that m, p, r are integers, $1 \leqq m, p, r \leqq q$, and $T_t \neq T_s$, if $t \neq s$, $\sum_{t=1}^{q^3} T_t$ is identically $T(q)$. If $\tilde{T}$ is a set with two resemblences, then the trilinear aggregating of $\tilde{T}$ is the trilinear uniting of $\tilde{T}$. If $T(q)$ and $\sum_{m=1}^q T_m$ are identically zero, then q-aggregating is q-canceling. The inverse operation to q-aggregating, that is the substitution of $\sum_{m=1}^q T_m$ for $T(q) - \sum_{t=q+1}^{q^3} T_t$, is a trilinear q-disaggregating.

Remark 5. ν-resemblence would have become an inner property of a given set of terms, not depending on the choice of the representations of the terms, if in this definition the equalities $\alpha_1^r L_1^r = \alpha_2^r L_2^r = \cdots = \alpha_q^r L_q^r$ had been substituted for the equalities $L_1^r = L_2^r = \cdots = L_q^r$. Here the coefficients $\alpha_1^r, \alpha_2^r, \cdots, \alpha_q^r$ are any numbers. We do not need this invariant definition in the present paper.

The following obvious lemmas finally show how the aggregating of terms influences the complexity of their sum.

LEMMA 4. *Any linear combination of q kin (2-resemble) terms is identically either a term, or zero. For $\nu \geqq 2$ the sum of q given ν-resemble terms is identically* $(1/q^2)\ T(q)$.

LEMMA 5.

$$q\sum_{m=1}^{q} L_m^{\nu} L_m^{\mu} = \sum_{r=1}^{q}\left(\sum_{m=1}^{q} \varepsilon^{rm} L_m^{\nu}\right)\left(\sum_{m=1}^{q} \varepsilon^{-rm} L_m^{\mu}\right) \tag{6}$$

where $\varepsilon = \varepsilon(q)$ is a primitive q-root of unity, that is $\varepsilon^m \neq 1$ if $1 \leqq m \leqq q-1$; $\varepsilon^q = 1$. In particular, for $q = 2$,

$$2(L_1^{\nu}L_1^{\mu} + L_2^{\nu}L_2^{\mu}) = [(L_1^{\nu}+L_2^{\nu})(L_1^{\mu}+L_2^{\mu}) + (L_1^{\nu}-L_2^{\nu})(L_1^{\mu}-L_2^{\mu})]. \tag{7}$$

COROLLARY. *Let a trilinear form $\hat{F}$ include ν-resemble terms $T_1, T_2, \cdots, T_q$. Let these q terms be aggregated. Let $R_i = R_i(\hat{F})$, $i = 0, 1$, be the decompositions of $\hat{F}$ before and after the aggregating of $T_1 \cdots, T_q$. Then*

$$q-1 \leqq \|R_0\| - \|R_1\| \leqq q \quad \textit{if } \nu \leqq 2,$$

$\|R_0\| - \|R_1\|$ *may be negative if $\nu \leqq 1$, and $\|R_0\| = \|R_1\|$ if $\nu = 1$ and the aggregating is determined by* (6) *or* (7).

We will conclude this section with a simple and nonconstructive but very general result that any decomposition of any given trilinear form as a sum of terms can be obtained from a trivial decomposition of this form by a chain of aggregatings and disaggregatings.

Definitions and notation. Let $\{a_\alpha, b_\beta, c_\gamma\}$ be three given sets of independent variables. Each product $\xi a_\alpha b_\beta c_\gamma$ where ξ is a number is called an elementary term. A decomposition of a given trilinear form $\sum_m L_m^1(a)L_m^2(b)L_m^3(c)$ which consists of only elementary terms and includes no proportional terms is called trivial (trivial LA is an example of such a decomposition for the form $\sum_{i,j,k} a_{ij}b_{jk}c_{ki}$).

The two following lemmas are obvious.

LEMMA 6. *Trivial decomposition of a given trilinear form is always unique. It can be obtained by a chain of disaggregatings with subsequent aggregatings of proportional trivial terms.*

LEMMA 7. *Any two decompositions of a given trilinear form can be transformed each into another by a chain of aggregatings and disaggregatings (compare with* [6]–[8]).

Proof. Lemma 6 is immediate from Lemma 7, since for any aggregating and for any disaggregating the inverse operations are disaggregating and aggregating.

13. Further improvement of LA. In this section we will redecompose the sums T^1, T^2 and T^3 of the correction terms of Algorithm 3 to reduce even further the total number of terms. This will be obtained after several aggregatings of pairs either of 1-resemble, or of kin terms. In order to write this process in a more compact and observable way, the results of each aggregating will be represented in the form of tables analogous to ones used in § 9. Again each table consists of 3 columns. But now each row (not a column) in the table represents a term, so that each multiple of the term is written in some column of the table. Unlike § 9, the numbers of rows are either 4, or 6 for each table in this section. We will use arrows to indicate which pairs of ν-resemble terms should be aggregated. Here $1 \leqq \nu \leqq 2$. If $\nu = 1$, then the aggregating is reduced to a straightforward application of formula (7). In this section we initially expand each of T^1, T^2, T^3 as a sum of their elementary correction terms. We will start with presenting (in Tables 9–14) 24 typical elementary correction terms of the trilinear form T^1 from Algorithm 3.

Table 9 Table 10

$$\begin{tikzcd}
a_{ij}b_{ij}c_{ij} \arrow[ddd, <->, bend right] & a_{\bar i\bar j}b_{\bar i\bar j}c_{\bar i\bar j} \arrow[ddd, <->, bend left] \\
a_{ij}b_{\bar i j}c_{i\bar j} \arrow[dr, <->] & a_{\bar i\bar j}b_{i\bar j}c_{\bar i j} \arrow[dl, <->] \\
a_{\bar i j}b_{i\bar j}c_{ij} & a_{i\bar j}b_{\bar i j}c_{\bar i\bar j} \\
a_{i\bar j}b_{ij}c_{\bar i j} & a_{\bar i j}b_{\bar i\bar j}c_{i\bar j}
\end{tikzcd}$$

Table 11 Table 12

$$\begin{tikzcd}
a_{ij}b_{ij}c_{ki} \arrow[ddd, <->, bend right] & a_{\bar i\bar j}b_{\bar i\bar j}c_{\bar k\bar i} \arrow[ddd, <->, bend left] \\
-a_{ij}b_{\bar i j}c_{\bar k i} \arrow[dr, <->] & -a_{\bar i\bar j}b_{i\bar j}c_{k\bar i} \arrow[dl, <->] \\
a_{\bar i j}b_{i\bar j}c_{k\bar i} & a_{i\bar j}b_{\bar i\bar j}c_{\bar k i} \\
-a_{i\bar j}b_{ij}c_{ki} & -a_{\bar i j}b_{\bar i j}c_{\bar k\bar i}
\end{tikzcd}$$

Table 13 Table 14

$$\begin{tikzcd}
a_{ij}b_{ij}c_{jk} \arrow[d, <->, bend right] & a_{\bar i\bar j}b_{\bar i\bar j}c_{\bar j\bar k} \arrow[d, <->, bend left] \\
a_{ij}b_{\bar i j}c_{jk} & a_{\bar i\bar j}b_{i\bar j}c_{\bar j\bar k} \\
-a_{\bar i j}b_{i\bar j}c_{\bar j k} \arrow[dr, <->] & -a_{i\bar j}b_{\bar i j}c_{i\bar k} \arrow[dl, <->] \\
-a_{i\bar j}b_{ij}c_{j\bar k} & -a_{\bar i j}b_{\bar i\bar j}c_{\bar j k}
\end{tikzcd}$$

Here $i, j, k = 0, 1, \cdots, s-1$, $\bar i = i+s$, $\bar j = j+s$, $\bar k = k+s$; $i, j, k \in \mathbf{S}^1(s)$. Each row is defined by a triplet of pairs of indices and represents a trivial term.

In fact eight more tables are required to represent elementary correction terms of $\mathbf{T}^2$ and $\mathbf{T}^3$. However, we do not display these tables here, since they can be obtained from Tables 11–14 just by cyclic permutations of their columns and the variables *a, b, c*. The signs minus are not considered squares of the tables. Each such sign stands in a certain row of the table and is considered a coefficient of the term represented by this row (similarly for coefficients $\frac{1}{2}$ and $\frac{1}{4}$ which appear in subsequent tables).

At first, we will display the chains of aggregates generated by the terms presented in Tables 11–14.

Table 15 Table 16

$$\begin{tikzcd}
(a_{ij}-a_{i\bar j})\, b_{ij}\, c_{ki} \arrow[d, <->, bend right] & a_{ij}\,(b_{ij}+b_{\bar i j})\, c_{jk} \arrow[d, <->, bend left] \\
-(a_{ij}-a_{i\bar j})\, b_{\bar i j}\, c_{\bar k i} & -a_{i\bar j}\,(b_{ij}+b_{\bar i j})\, c_{j\bar k} \\
-(a_{\bar i\bar j}-a_{\bar i j})\, b_{i\bar j}\, c_{k\bar i} \arrow[d, <->, bend right] & -a_{\bar i j}\,(b_{\bar i\bar j}+b_{i\bar j})\, c_{\bar j k} \arrow[d, <->, bend left] \\
(a_{\bar i\bar j}-a_{\bar i j})\, b_{\bar i\bar j}\, c_{\bar k\bar i} & a_{\bar i\bar j}\,(b_{\bar i\bar j}+b_{i\bar j})\, c_{\bar j\bar k}
\end{tikzcd}$$

Table 17 Table 18

$$\begin{tikzcd}
\frac{1}{2}(a_{ij}-a_{i\bar j})\,(b_{ij}+b_{\bar i j})\,(c_{ki}-c_{\bar k i}) \arrow[r, <->] & \frac{1}{2}(a_{ij}-a_{i\bar j})\,(b_{ij}+b_{\bar i j})\,(c_{jk}+c_{j\bar k}) \\
\frac{1}{2}(a_{ij}-a_{i\bar j})\,(b_{ij}-b_{\bar i j})\,(c_{ki}+c_{\bar k i}) & \frac{1}{2}(a_{ij}+a_{i\bar j})\,(b_{ij}+b_{\bar i j})\,(c_{jk}-c_{j\bar k}) \\
\frac{1}{2}(a_{\bar i\bar j})-a_{\bar i j})\,(b_{\bar i\bar j}+b_{i\bar j})\,(c_{\bar k\bar i}-c_{k\bar i}) \arrow[r, <->] & \frac{1}{2}(a_{\bar i\bar j}-a_{\bar i j})\,(b_{i\bar j}+b_{\bar i\bar j})\,(c_{\bar j\bar k}+c_{\bar j k}) \\
\frac{1}{2}(a_{\bar i\bar j}-a_{\bar i j})\,(b_{\bar i\bar j}-b_{i\bar j})\,(c_{\bar k\bar i}+c_{k\bar i}) & \frac{1}{2}(a_{\bar i\bar j}+a_{\bar i j})\,(b_{\bar i\bar j}+b_{i\bar j})\,(c_{\bar j\bar k}-c_{\bar j k})
\end{tikzcd}$$

Table 19

$$\frac{1}{2}(a_{ij}-a_{i\bar j})\ (b_{ij}+b_{\bar i j})\ (c_{ki}-c_{\bar k i}+c_{jk}+c_{j\bar k})$$
$$\frac{1}{2}(a_{\bar i\bar j}-a_{\bar i j})\ (b_{\bar i\bar j}+b_{i\bar j})\ (c_{\bar k\bar i}-c_{k\bar i}+c_{\bar j\bar k}+c_{\bar j k})$$
$$\frac{1}{2}(a_{ij}-a_{i\bar j})\ (b_{ij}-b_{\bar i j})\ (c_{ki}+c_{\bar k i})$$
$$\frac{1}{2}(a_{\bar i\bar j}-a_{\bar i j})\ (b_{\bar i\bar j}-b_{i\bar j})\ (c_{\bar k\bar i}+c_{k\bar i})$$
$$\frac{1}{2}(a_{ij}+a_{i\bar j})\ (b_{ij}+b_{\bar i j})\ (c_{jk}-c_{j\bar k})$$
$$\frac{1}{2}(a_{\bar i\bar j}+a_{\bar i j})\,(b_{\bar i\bar j}+b_{\bar i\bar i})\,(c_{\bar j\bar k}-c_{\bar j k}).$$

Similar chains of aggregatings result in representing the elementary correction terms of T^2 and of T^3 in two similar tables. Two latter tables (they will be called *final for* T^2 and T^3) obviously coincide with Table 19 up to within cyclic permutations of three columns of these tables and of the variables a, b, c.

Now it is easy to verify that there are only $\frac{3}{2}n^2$ families of kin terms in the decomposition represented by Table 19. Adding the terms represented by two cyclic permutations of three columns of Table 19 we increase the number of the families by 3 times, that is, up to $\frac{9}{2}\mathrm{n}^2$. In this case all the correction terms are represented in these two tables, and in Tables 9, 10, and 19. Now we will construct a chain of aggregatings of terms represented in Tables 9 and 10, such that sum of all these terms is finally turned into a sum of terms belonging to the same $\frac{9}{2}n^2$ families. Here is the chain of tables (Tables 20–23) derived from Tables 9 and 10 by application of our chain of aggregatings. The aggregatings are defined by arrows in the tables.

TABLE 20

$\frac{1}{2}(a_{ij}-a_{i\bar{j}})$	b_{ij}	$(c_{ij}-c_{\bar{i}j})$
$\frac{1}{2}(a_{ij}-a_{i\bar{j}})$	$b_{\bar{i}j}$	$(c_{i\bar{j}}-c_{\bar{i}\bar{j}})$
$\frac{1}{2}(a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$b_{\bar{i}\bar{j}}$	$(c_{\bar{i}\bar{j}}-c_{i\bar{j}})$
$\frac{1}{2}(a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$b_{i\bar{j}}$	$(c_{\bar{i}j}-c_{ij})$

TABLE 21

$\frac{1}{2}(a_{ij}+a_{i\bar{j}})$	b_{ij}	$(c_{ij}+c_{\bar{i}j})$
$\frac{1}{2}(a_{ij}+a_{i\bar{j}})$	$b_{\bar{i}j}$	$(c_{i\bar{j}}+c_{\bar{i}\bar{j}})$
$\frac{1}{2}(a_{\bar{i}\bar{j}}+a_{\bar{i}j})$	$b_{\bar{j}\bar{j}}$	$(c_{\bar{i}\bar{j}}+c_{i\bar{j}})$
$\frac{1}{2}(a_{\bar{i}\bar{j}}+a_{\bar{i}j})$	$b_{i\bar{j}}$	$(c_{\bar{i}j}+c_{ij})$

TABLE 22

$\frac{1}{4}(a_{ij}-a_{i\bar{j}})$	$(b_{ij}-b_{\bar{i}j})$	$(c_{ij}-c_{\bar{i}j}-c_{i\bar{j}}+c_{\bar{i}\bar{j}})$
$\frac{1}{4}(a_{ij}-a_{i\bar{j}})$	$(b_{ij}+b_{\bar{i}j})$	$(c_{ij}-c_{\bar{i}j}+c_{i\bar{j}}-c_{\bar{i}\bar{j}})$
$\frac{1}{4}(a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$(b_{\bar{i}\bar{j}}-b_{i\bar{j}})$	$(c_{\bar{i}\bar{j}}-c_{i\bar{j}}-c_{\bar{i}j}+c_{ij})$
$\frac{1}{4}(a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$(b_{\bar{i}\bar{j}}+b_{i\bar{j}})$	$(c_{\bar{i}\bar{j}}-c_{i\bar{j}}+c_{\bar{i}j}-c_{ij})$

TABLE 23

$\frac{1}{4}(a_{ij}+a_{i\bar{j}}-a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$(b_{ij}-b_{i\bar{j}})$	$(c_{ij}+c_{\bar{i}j})$
$\frac{1}{4}(a_{ij}+a_{i\bar{j}}+a_{\bar{i}\bar{j}}+a_{\bar{i}j})$	$(b_{ij}+b_{i\bar{j}})$	$(c_{ij}+c_{\bar{i}j})$
$\frac{1}{4}(a_{ij}+a_{i\bar{j}}-a_{\bar{i}\bar{j}}-a_{\bar{i}j})$	$(b_{\bar{i}j}-b_{\bar{i}\bar{j}})$	$(c_{\bar{i}\bar{j}}+c_{i\bar{j}})$
$\frac{1}{4}(a_{ij}+a_{i\bar{j}}+a_{\bar{i}\bar{j}}+a_{\bar{i}j})$	$(b_{\bar{i}j}+b_{\bar{i}\bar{j}})$	$(c_{\bar{i}\bar{j}}+c_{i\bar{j}})$

We can easily verify that for each of four terms in Table 22 there is a kin term in Table 19. Now we display our final table (Table 24) for the terms derived by the chains of aggregatings from elementary correction terms of T^3. This final table coincides with Table 19 up to a cyclic permutation of columns and of the variables a, b, c.

TABLE 24

$\frac{1}{2}(a_{ki}-a_{\bar{k}i}+a_{jk}+a_{j\bar{k}})$	$(b_{ij}-b_{i\bar{j}})$	$(c_{ij}+c_{\bar{i}j})$
$\frac{1}{2}(a_{\bar{k}\bar{i}}-a_{k\bar{i}}+a_{\bar{j}\bar{k}}+a_{\bar{j}k})$	$(b_{\bar{i}\bar{j}}-b_{\bar{i}j})$	$(c_{\bar{i}\bar{j}}+c_{i\bar{j}})$
$\frac{1}{2}(a_{ki}+a_{\bar{k}i})$	$(b_{ij}-b_{i\bar{j}})$	$(c_{ij}-c_{\bar{i}j})$
$\frac{1}{2}(a_{\bar{k}\bar{i}}+a_{k\bar{i}})$	$(b_{\bar{i}\bar{j}}-b_{\bar{i}j})$	$(c_{\bar{i}\bar{j}}-c_{i\bar{j}})$
$\frac{1}{2}(a_{jk}-a_{j\bar{k}})$	$(b_{ij}+b_{i\bar{j}})$	$(c_{ij}+c_{\bar{i}j})$
$\frac{1}{2}(a_{\bar{j}\bar{k}}-a_{\bar{j}k})$	$(b_{\bar{i}\bar{j}}+b_{\bar{i}j})$	$(c_{\bar{i}\bar{j}}+c_{i\bar{j}})$.

Comparing Tables 23 and 24 we notice that for each of four terms in the former there is a kin term in the latter.

Now we will construct chains of aggregations of $8s = 4n$ desirable terms $T(i(\alpha), i(\beta), i(\gamma))$ where α, β, γ are all 0-1 triplets. These terms are also included in T^1, T^2, T^3. The chains once again will be displayed in tables (Tables 25–30). Arrows will indicate pairs of terms to be aggregated. We will start with increasing the number of desirable terms by $4s = 2n$, by adding and subtracting the sum $2\sum_{i=0}^{s-1}(T(iii)+T(\bar{i}\bar{i}\bar{i}))$ to $T^1+T^2+T^3$ (this is trilinear creating, see the definition in the previous section). Now let these $6n$ desirable terms be represented in Tables 25–27 and then aggregated.

TABLE 25	TABLE 26	TABLE 27
↱ $a_{ii}b_{ii}c_{ii}$	↱ $a_{ii}b_{ii}c_{ii}$	↱ $a_{ii}b_{ii}c_{ii}$
↳ $a_{i\bar{i}}b_{\bar{i}i}c_{ii}$	↳ $a_{\bar{i}i}b_{ii}c_{i\bar{i}}$	↳ $a_{ii}b_{i\bar{i}}c_{\bar{i}i}$
↱ $a_{\bar{i}i}b_{i\bar{i}}c_{\bar{i}\bar{i}}$	↱ $a_{i\bar{i}}b_{\bar{i}\bar{i}}c_{\bar{i}i}$	↱ $a_{\bar{i}\bar{i}}b_{\bar{i}i}c_{i\bar{i}}$
↳ $a_{\bar{i}\bar{i}}b_{\bar{i}\bar{i}}c_{\bar{i}\bar{i}}$	↳ $a_{\bar{i}\bar{i}}b_{\bar{i}\bar{i}}c_{\bar{i}\bar{i}}$	↳ $a_{\bar{i}\bar{i}}b_{\bar{i}\bar{i}}c_{\bar{i}\bar{i}}$

TABLE 28

$\frac{1}{2}(a_{ii}+a_{i\bar{i}})\ (b_{ii}+b_{\bar{i}i})\ c_{ii}$
$\frac{1}{2}\ (a_{ii}-a_{i\bar{i}})\ (b_{ii}-b_{\bar{i}i})\ c_{ii}$
$\frac{1}{2}(a_{\bar{i}\bar{i}}+a_{\bar{i}i})\ (b_{\bar{i}\bar{i}}+b_{i\bar{i}})\ c_{\bar{i}\bar{i}}$
$\frac{1}{2}(a_{\bar{i}\bar{i}}-a_{\bar{i}i})\ (b_{\bar{i}\bar{i}}-b_{i\bar{i}})c_{\bar{i}\bar{i}}$

TABLE 29

$\frac{1}{2}(a_{ii}+a_{\bar{i}i})\ b_{ii}\ (c_{ii}+c_{i\bar{i}})$
$\frac{1}{2}(a_{ii}-a_{\bar{i}i})\ b_{ii}\ (c_{ii}-c_{i\bar{i}})$
$\frac{1}{2}(a_{\bar{i}\bar{i}}+a_{i\bar{i}})\ b_{\bar{i}\bar{i}}\ (c_{\bar{i}\bar{i}}+c_{\bar{i}i})$
$\frac{1}{2}(a_{\bar{i}\bar{i}}-a_{i\bar{i}})\ b_{\bar{i}\bar{i}}\ (c_{\bar{i}\bar{i}}-c_{\bar{i}i})$

TABLE 30

$\frac{1}{2}a_{ii}\ (b_{i\bar{i}}+b_{ii})\ (c_{\bar{i}i}+c_{ii})$
$\frac{1}{2}a_{ii}\ (b_{i\bar{i}}-b_{ii})\ (c_{\bar{i}i}-c_{ii})$
$\frac{1}{2}\ a_{\bar{i}\bar{i}}\ (b_{\bar{i}i}+b_{\bar{i}\bar{i}})\ (c_{i\bar{i}}+c_{\bar{i}\bar{i}})$
$\frac{1}{2}a_{\bar{i}\bar{i}}\ (b_{\bar{i}i}-b_{\bar{i}\bar{i}})\ (c_{i\bar{i}}-c_{\bar{i}\bar{i}})$

Each term in Tables 28 and 30 has a kin (2-resemble) term in the last four rows of Tables 19 and 24 if we write $i = j$ in the latters. Similarly, we could display the final table for the terms derived by the chains of aggregatings from elementary correction terms of T^2 (this table would be analogous to Tables 19 and 24). Then comparing this table for $i = j$ and Table 29 we could verify that each term in Table 29 has kin terms among the terms of our $\frac{9}{2}n^2$ families. Only n terms $-2T(iii)$ and $-2T(\bar{i}\bar{i}\bar{i})$, $i-0, 1, \cdots, s-1$ have no kin terms among the terms of our $\frac{9}{2}n^2$ families. Thus the total number of different families in our decomposition of $\mathrm{T}^1+\mathrm{T}^2+\mathrm{T}^3$ equals $\frac{9}{2}n^2+n$, where n families consist of terms $T(iii)$ and $T(\bar{i}\bar{i}\bar{i})$, $i = 0, 1, \cdots, s-1$.

This gives the desirable improvement of LA. The complexity of the resulting LA (which will be denoted algorithm 4) equals $(n^3-4n)/3+\frac{9}{2}n^2+n = (n^3-n)/3+\frac{9}{2}n^2$. We proved the following theorem.

THEOREM 4. *For any even n there exists* $\mathrm{LA} = \mathrm{LA}(n)$ *whose complexity is* $(n^3-n)/3+\frac{9}{2}n^2$.

Exercise 4. Modify LA from Exercise 3 to reduce its complexity to $(n^3-n)/3+\frac{9}{2}n^2$.

Exercise 5. Find a formal representation for Algorithm 4 which is similar to the formal representation of Algorithm 3 in § 12 (use the final tables and Table 23).

14. The Main Theorem and illustrating tables.

MAIN THEOREM. *There exist algorithms for* MM, MI, ED, SLS *involving only* $O(N^{\beta})$ *arithmetic operations, where* $N \times N$ *is a size of quadratic matrices involved in a given problem* MM, ML, ED, SLS, *and* $\beta = \log_{48} 4721 \approx 2.780 < \log_2 7 \approx 2.807$.

The Main Theorem follows from Theorems 1 and 4 (the latter applied here for $n = 48$).

Remark 6. Since the proofs of Theorems 1 and 4 are constructive, the same is true for the Main Theorem.

Here are three tables (Tables 31–33) illustrating the results of this paper.

TABLE 31
$M(n)$ for Algorithms 2, 2a *and for the combinations of the best previously published LA including* [21], [22], [25].

n	Algorithm 2	Alborithm 2a*	The best previously published
10	725	710	721
12	1188	1170	1127
14	1813	1792	1909
16	2624	2600	2401
18	3645	3618	3375
20	4900	4870	5047
22	6413	6380	6972

* For $n \geqq 16$ the complexity of Algorithm 2a is greater than the complexity of Algorithm 4.

TABLE 32
$M(n)$ and $\log_n M(n)$ *for Algorithm* 3.

n	$M(n)$	$\log_n M(n)$
50	56600	2.7974
52	63024	2.7969
54	69912	2.7964
56	77280	2.7961
58	85144	2.7958
60	93520	2.7955
62	102424	2.7954
64	111872	2.79525
66	121880	2.79517
68	132464	2.79513
70	143640	2.79512
72	155424	2.79515
74	167832	2.7952
76	180880	2.7953
78	194584	2.7954
80	208960	2.7955
82	224024	2.7956
84	239792	2.7958
86	256280	2.7959
88	273504	2.7961
90	291480	2.7963

TABLE 33

$M(n)$ and $\log_n M(n)$ for Algorithm 4.

n	$M(n)$	$\log_n M(n)$*
12	1220	2.8599
18	3396	2.8129
20	4460	2.8050
28	10836	2.7881
36	21372	2.7821
44	37092	2.78029
46	41952	2.78017
48	47216	2.7801419
50	52900	2.78019
52	59020	2.78030
54	65592	2.78046
60	88180	2.78119
80	199440	2.7848
100	378300	2.7889
150	1262000	2.7979
200	2846600	2.8050
220	3767060	2.8063509
222	3868720	2.8075773
240	4867120	2.8095

* Compare with $\log_2 7 \approx 2.8073549$

Here is a summary of the basic steps which have been done to obtain the Main Theorem (including the results of [12], [13], [26] etc.).

1. The problems **MM, MI, ED, SLS** of any size can be reduced to constructing fast **LA** (that is **LA** with a small number of "essential multiplications") for a problem **MM** of a particular size. Any algorithm for the problems **MM, MI** can always be transformed into **LA** for **MM** having roughly the same complexity.

2. A heuristic conclusion: constructing fast **LA** could be started with studying the case of **LA** of a general size n.

3. Another heuristic conclusion: the trilinear representation is more promising for constructing fast **LA** than the bilinear one since the technique of aggregating terms is easier to apply in the trilinear case.

4. Any **LA** is a chain of transformations from the trivial one by aggregating and disaggregating terms.

5. The complexity is reduced if it is possible to aggregate 2-resemble terms (to unite kin terms).

6. The sets consisting of comparatively small number of aggregates and groups of 2-resemble terms can be created by appropriate aggregating of 0-resemble terms of the trivial **LA**.

7. The set of terms arising after appropriate aggregating of $\mathbf{n^3/2}$ pairs of 0-resemble terms of the trivial **LA** consists of roughly $n^3/2$ aggregates and λn^2 groups of 2-resemble terms, such that $\lambda \leqq 3$.

8. $\lambda \geqq \frac{9}{4}$.

9. Numerous unnecessary terms which probably cannot be partitioned into $O(n^2)$ groups of 2-resemble terms arise after aggregating triplets of terms of the trivial **LA**. However, these terms can be canceled if the triplets of terms and coefficients ±1 for a representation of each term are appropriately chosen. This results in fast **LA** having the complexity $(n^3-4n)/3+6n^2$ for any even n.

10. The algorithm can be further improved by aggregating 1-resemble terms.

Remark 7. The above listed steps 1–7 have already been considered and applied for constructing fast **LA** by Pan in 1972 (see [19]).

15. Open problems (brief discussion). In the previous sections new upper bounds on the complexity $M = M(n)$ of LA have been established. They resulted in asymptotically fast algorithms for some important problems. Now two following questions arise. How far can this or similar techniques be extended? What are the lower bounds on $M(n)$ and, more generally, on the complexity $M(m, n, p)$ of LA(m, n, p) for matrix multiplications (see Remark 1 in § 2)? An application of the active operation-basic substitution technique[1] immediately gives lower bounds $M(m, n, p) \geqq (m+n-1)p$ and, in particular, for $m = n = p$, $M(n) \geqq 2n^2 - n$. An application of this technique to a version of LA, which can be called a linear one (see e.g., [6]–[8]) to distinguish it from a bilinear one described in § 2 of this paper, and from a trilinear one described in § 3, yields the lower bounds $M(m, n, p) \geqq (m-1)(n+1) + np$, $M(n) = M(n, n, n) \geqq 2n^2 - 1$ (see [5], [30]). These lower bounds can be slightly improved for $(m, n, p) = (2, 2, p)$, $(m, n, p) = (2, 3, 3)$ (see [11], [12], [19]), and for $(m, n, p) = (2, 3, 4)$ and $m = n = p = 3$ (see [19]). Even so, the gap between the best known lower and upper bounds is enormous. The present author hopes that the technique introduced in this paper will be extended to the problems of evaluation of different sets of bilinear forms. It would be also interesting to develop this technique further to speed-up matrix multiplication. For instance, is it possible to design a faster LA by including aggregating 4-tuples, or 6-tuples, or 9-tuples of 0-resemble desirable terms into our construction? Do there exist other efficient ways to use aggregating 0-resemble terms for decomposition of a given trilinear form?

Lately combining the techniques of the present paper and of extension of algebraic fields (D. Bini, M. Capovani, G. Lotti, F. Romani, S. Winograd) with the reduction of total to partial MM (A. Schönhage) gave the complexity bounded by cN^{β}, $\beta < 2.6054$, c is an enormous constant, for $N \times N$ *MM* (V. Pan).

To further the progress in designing fast matrix multiplication algorithms, perhaps even in parallel with search for fast LA, the problem should be restated in a more practical way. Assume that the entries of given matrices are numbers given with a certain precision p in binary form. Then it would be extremely interesting to find lower and/or upper bounds on the number of bit operations involved in the evaluation (with another given precision q) of the product of two given matrices.

16. Acknowledgments. My interest in the algebraic complexity theory and particularly in the problem of matrix multiplication was encouraged in the fifties and sixties by Professor V. D. Erokhin (who is not alive now) and by Professor A. G. Vitushkin (at Moscow State University). My coming back to this field in 1977 became possible thanks to Dr. S. Winograd at the IBM T. J. Watson Research Center in Yorktown Heights. I am very grateful to all three of them for this and to the latter also for very instructive discussions and suggestions resulted in a drastic improvement of the text of my first draft of this paper.

I wish to thank Dr. N. Pippenger at the IBM Research Center for his helpful criticism of my terminology, Prof. A. B. Borodin at the University of Toronto for his helpful comments on the original version of this paper, several people for pointing out some minor errors in [20], and also Ms. C. Brown who typed the paper.

[1] This technique was introduced for establishing lower bounds on the number of arithmetic operations ± and $\dot{x}$ for polynomial evaluation in [17] (on the number of multiplications/divisions) and in [18] (on the number of additions/subtractions). It was rediscovered in the case of ± in [15] and extended in several directions in [14], [24], [28].

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*, American Elsevier, New York, 1975.

[3] R. W. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Proc. Fifth Ann. ACM Symp. on Theory of Computing, 1973, pp. 88–95.

[4] ———, *On the number of multiplications required for matrix multiplication*, this Journal, 5 (1976), pp. 624–628.

[5] ———, *On the optimal 5 (1976), evaluation of a set of bilinear forms*, Linear Algebra and Appl. 19 (1978), pp. 207–235.

[6] C. M. FIDUCCIA, *Fast matrix multiplication*, Proc. Third Ann. ACM Symp. on Theory of Computing, 1971, pp. 45–49.

[7] ———, *On obtaining upper bounds on the complexity of matrix multiplication*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 31–40.

[8] ———, *On algebraic complexity of matrix multiplication*, Ph.D. Thesis, Brown University, Providence, RI., 1973.

[9] N. GASTINEL, *Sur le calcul des produits de matrices*, Numer. Math., 17 (1971), pp. 222–229.

[10] H. F. DEGROOTE, *On varieties of optimal algorithms for the computation of bilinear mappings II. Optimal algorithms for* 2×2 *matrix multiplication*, Tech. Rep., Mathematisches Institute, Universität Tübingen, 1978.

[11] J. E. HOPCROFT AND L. R. KERR, *Some techniques for proving certain simple programs optimal*, Proc. of the 1969 Tenth Ann. Symp. on Switching and Automata Theory, 1969, pp. 36–45.

[12] ———, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[13] J. E. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplications and other bilinear forms*, this Journal, 21 (1973), 159–173.

[14] Z. M. KEDEM AND D. G. KIRKPATRICK, *Addition requirements for rational functions*, this Journal, 6 (1977), pp. 188–199.

[15] D. G. KIRKPATRICK, *On the additions necessary to compute certain functions*, Proc. 4th Ann. ACM Symp. on Theory of Computing, 1972, pp. 94–101.

[16] J. D. LADERMAN, *A noncommutative algorothm for multiplying* 3×3 *matrices using 23 multiplications*, Bull. Amer. Math. Soc., 82 (1976), pp. 126–128.

[17] V. YA PAN, *On some methods of computing polynomial values*, Problemy Kibernet., (1962), pp. 21–30. Transl. Problems of Cybernetics, edited by A. A. Lyapunov., U.S.S.R., (1962), 7, pp. 20–30, U.S. Department of Commerce.

[18] ———, *Methods for computing polynomials*, Ph.D. Thesis, Department of Mechanics and Mathematics, Moscow State University, 1964. (In Russian.)

[19] ———, *On schemes for the computation of products and inverses of matrices*, Russian Math. Surveys, 27 (1972), no. 5, pp. 249–250.

[20] ———, *Strassen's algorithm is not optimal*, Proceedings of the 19th Annual Symposium on Foundations of computer science, 1978, 166–176.

[21] R. L. PROBERT, *On the composition of matrix multiplication algorithms*, Proc. of Sixth Manitoba Conf. on Num. Math. and Computing, Congressus Numerantium 18, 1977, pp. 357–366.

[22] G. SCHACHTEL, *A non-commutative algorithm for mutiplying* 5×5 *matrices using 103 multiplications*, Information Processing Letters (1978), no. 4, pp. 180–182.

[23] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[24] ———, *Evaluation of Rational Functions*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 1–10.

[25] O. SYKORA, *A Fast Non-commutative Algorithm for Matrix Multiplication*, Lecture Notes in Computer Science, 53, Springer-Verlag, New York, 1977, pp. 504–512.

[26] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 26 (1973), pp. 184–202.

[27] S. WINOGRAD, *A new algorithm for inner product*, IEEE Trans. Computers, C-17 (1968), pp. 693–694.

[28] ———, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

[29] S. WINOGRAD, *On multiplication of* 2×2 *matrices*, Linear Algebra and Appl., 4 (1971), pp. 381–388.

[30] ———, to appear.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

ON THE POLYHEDRAL DECISION PROBLEM*

ANDREW C. YAO† AND RONALD L. RIVEST‡

Abstract. Computational problems sometimes can be cast in the following form: Given a point **x** in R^n, determine if **x** lies in some fixed polyhedron. In this paper we give a general lower bound to the complexity of such problems, showing that $\frac{1}{2}\log_2 f_s$ linear comparisons are needed in the worst case, for any polyhedron with f_s s-dimensional faces. For polyhedra with abundant faces, this leads to lower bounds nonlinear in n, the number of variables.

Key words. adversary strategy, complexity, dimension, edge, face, linear decision tree, lower bound, polyhedron

1. Introduction. Computational problems sometimes can be cast in the following form. Given n numbers $x_1, x_2, \cdots, x_n$, determine if they satisfy some fixed set of linear inequalities, i.e., if the point $\mathbf{x}=(x_1, x_2, \cdots, x_n)$ lies in some "polyhedron". For example, the problem of verifying a maximum element can be stated as "Given $x_1, x_2, \cdots, x_n$, determine if $x_1 \geqq x_i$ for all i." As another example, a version of the minimum spanning tree verification problem is the following: Given a weight function w on the set of edges in a graph G, determine if $w(T_0) \leqq w(T)$ for all spanning trees T of G (T_0 is a fixed spanning tree, and $w(T)$ is the sum of edge weights in T). The aim of this paper is to establish a general lower bound on this type of problems, in terms of some intrinsic characteristics of the polyhedron in question. In contrast to a previous result of this type (Rabin [5]), the present bound can give values larger than the number of variables.

2. Definitions and notations. Let R^n be the space of real n-tuples. A set P in R^n is a *polyhedron* if $P=\{\mathbf{x}|\mathbf{x}\in R^n,\ l_i(\mathbf{x})\leqq 0,\ i=1,2,\cdots,m\}$, where m is an integer, $\mathbf{x}=(x_1, x_2, \cdots, x_n)$, and $l_i(\mathbf{x})=\sum_{1\leqq j\leqq n} c_{ij}x_j - a_i$ for some real numbers c_{ij}, a_i. The *polyhedral decision problem* $B(P)$ is to determine whether $\mathbf{x}\in P$ for any input **x**. We are interested in the *linear decision tree model* [1], [5], [10]. An algorithm is a ternary tree with each internal node representing a test of the form "$\sum \lambda_i x_i - c : 0$", and each leaf containing a "yes" or "no" answer. For any input, the algorithm proceeds by moving down the tree, testing and branching according to the test results ($<$, $=$, or $>$), until a leaf is reached. At that point, the answer to the question "Is $\mathbf{x}\in P$?" is supplied by the leaf. The *cost* of an algorithm is the height of the tree, i.e., the maximum number of tests made for any input. The *complexity* of $B(P)$ is the minimum cost of any algorithm, and is denoted by $C(P)$.

Faces of a polyhedron. Let $P=\{\mathbf{x}|l_i(\mathbf{x})\leqq 0, i=1,2,\cdots,m\}$ be a polyhedron in R^n. To each subset H (maybe $\varnothing$) of $\{1,2,\cdots,m\}$, we define a set $F_H(P)\subseteq R^n$ by $F_H(P)=\{\mathbf{x}|l_i(\mathbf{x})<0 \text{ for each } i\in H;\ l_i(x)=0 \text{ for each } i\notin H\}$. We say that $F_H(P)$ is a *face* of dimension s if the smallest affine subspace of R^n containing $F_H(P)$ has dimension s. (An *affine subspace* is the solution to a set of inhomogeneous equations. See, for example, [6] for more discussions.) The empty face has dimension -1 by convention. Let $\mathscr{F}_s(P)$ be the set of faces of dimension s of P. Note that no two elements of $\mathscr{F}_s(P)$

* Received by the editors July 28, 1978, and in revised form May 20, 1979. This research was supported in part by National Science Foundation under Grants MCS-72-03752 A03, MCS-77-05313, and MCS-75-14294.

† Computer Science Department, Stanford University, Stanford, California 94305.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

overlap. The set of faces $\mathcal{F}_s(P)$ is independent of the choice of $l_i(x)$. That is, if $P=\{\mathbf{x}|l'_i(\mathbf{x})\leqq 0,\ i=1,2,\cdots,m'\}$, the set $\mathcal{F}_s(P)$ constructed using $\{l'_i(\mathbf{x})\}$ is the same as the one constructed using $\{l_i(\mathbf{x})\}$. For an intrinsic definition of faces, see for example [3], [8]. A face of dimension 1 is called an edge, as it is part of a line (agreeing with intuition).

Open polyhedra. A nonempty set Q in R^n is called an *open polyhedron* if $Q=\{x|l_i(x)<0,\ i=1,2,\cdots,m\}$. The concepts of faces and set of faces are defined identically as for polyhedra. More precisely, let $P=\{\mathbf{x}|l_i(\mathbf{x})\leqq 0,\ i=1,2,\cdots,m\}$, then $F_H(Q)=F_H(P)$, $\mathcal{F}_s(Q)=\mathcal{F}_s(P)$.

3. Lower bounds for polyhedral decision problems. Let T be a polygon on the plane. Suppose we are asked to decide if a given point x is inside T by making a series of tests of the form "$\boldsymbol{\lambda}\cdot\mathbf{x}-c$: 0". It is easy to see that about log v tests are necessary if T has v vertices. Our main result is the following generalization.

THEOREM 1. *Let $P=\{\mathbf{x}|l_i(\mathbf{x})\leqq 0$ for $i=1,2,\cdots,m\}$ be a polyhedron in R^n. Then for each s,*

$$2^{C(P)}\cdot\binom{C(P)}{n-s}\geqq|\mathcal{F}_s(P)|.$$

COROLLARY.

$$C(P)\geqq\tfrac{1}{2}\log_2|\mathcal{F}_s(P)|.$$

Theorem 1 relates the complexity of $B(P)$ to certain "static" combinatorial properties of the polyhedron P. Informally, if a polyhedron P has many edges (or faces), then the theorem says it is difficult to decide whether a point lies in P. The rest of this section is devoted to proving Theorem 1. Note that the corollary follows from Theorem 1 since $\binom{C(P)}{n-s}\leqq 2^{C(P)}$.

We will assume in what follows that P is of dimension n. The following informal argument demonstrates that this can be done without loss of generality. Suppose that dim $(P)=n'<n$. Let $S\subseteq R^n$ be the smallest affine subspace of R^n containing all of P; thus dim $(S)=n'$. Now every test $\sum\lambda_ix_i-c$: 0 in R^n either corresponds to a linear test $\sum\lambda'_ix'_i-c'$: 0 in S (where $\mathbf{x}'$ is, for $\mathbf{x}\in S$, $\mathbf{x}$ expressed in a basis for S), or else (if $\{\mathbf{x}\in R^n|\sum\lambda_ix_i=c\}\supseteq S$) the test $\sum\lambda_ix_i-c$: 0 is useful only for determining if $\mathbf{x}\in S$, and not for telling if $\mathbf{x}\in P$ under the assumption that $\mathbf{x}\in S$. Therefore the complexity of determining if an $\mathbf{x}\in R^n$ is in P is at least as great as the complexity of determining if an $\mathbf{x}\in S$ is in P. Since dim $(S)=$ dim (P) we are finished with our demonstration.

To prove Theorem 1 we shall adopt the "adversary approach" commonly used in deriving lower bounds for decision trees. We shall design an adversary strategy $\mathcal{A}$ which, for any algorithm, will specify the outcomes for successive queries based on the results of previous queries. The following lemma is essential to the construction of $\mathcal{A}$.

LEMMA 1. *Let $Q=\{\mathbf{x}|p_i(\mathbf{x})<0,\ i=1,2,\cdots,t\}$ be a nonempty open polyhedron, $q(\mathbf{x})=\sum_{i=1}^n\lambda_ix_i-c$ a linear form, $Q_1=Q\cap\{\mathbf{x}|q(\mathbf{x})<0\}$ and $Q_2=Q\cap\{\mathbf{x}|q(\mathbf{x})>0\}$. Then for each s, there exists a $j\in\{1,2\}$ such that Q_j is nonempty, and $|\mathcal{F}_s(Q_j)|\geqq\frac{1}{2}|\mathcal{F}_s(Q)|$.*

Proof of Lemma 1. If $Q_2=\emptyset$, then $Q\subseteq\{\mathbf{x}|q(\mathbf{x})\leqq 0\}$. Since Q is an open set, we must have $Q\subseteq\{\mathbf{x}|q(\mathbf{x})<0\}$. Therefore, $Q_1=Q$, and $j=1$ satisfies the requirements. Similarly, for the case $Q_1=\emptyset$ we can choose $j=2$. It remains to prove the lemma when both Q_1 and Q_2 are nonempty. We shall accomplish this by constructing a 1–1 mapping ψ from $\mathcal{F}_s(Q)$ into $\mathcal{F}_s(Q_1)\cup\mathcal{F}_s(Q_2)$. This then implies that $|\mathcal{F}_s(Q)|\leqq|\mathcal{F}_s(Q_1)|+|\mathcal{F}_s(Q_2)|$. We can then choose a j such that $|\mathcal{F}_s(Q_j)|\geqq\frac{1}{2}|\mathcal{F}_s(Q)|$.

Now we construct ψ. Let $F_H(Q) \in \mathscr{F}_s(Q)$. Define

$$A_1 = F_H(Q) \cap \{\mathbf{x} | q(\mathbf{x}) < 0\},$$
$$A_2 = F_H(Q) \cap \{\mathbf{x} | q(\mathbf{x}) > 0\},$$
$$A_3 = F_H(Q) \cap \{\mathbf{x} | q(\mathbf{x}) = 0\}.$$

Case (1). $A_1 \cup A_2 = \emptyset$. In this case $F_H(Q) \subseteq \{\mathbf{x} | q(\mathbf{x}) = 0\}$. Let us write $Q_1 = \{\mathbf{x} | p_i(\mathbf{x}) < 0, \quad i = 1, 2, \cdots, t+1\}$, with $p_{t+1}(\mathbf{x}) = q(\mathbf{x})$. Clearly $F_H(Q_1) = F_H(Q) \cap \{q(\mathbf{x}) = 0\} = F_H(Q)$. Define $\psi(F_H(Q)) = F_H(Q_1)$.

Case (2). $A_1 \cup A_2 \neq \emptyset$. Assume that $A_1 \neq \emptyset$; the case $A_2 \neq \emptyset$ can be similarly treated. Write as before, $Q_1 = \{\mathbf{x} | p_i(\mathbf{x}) < 0, \ i = 1, 2, \cdots, t+1\}$ with $p_{t+1}(\mathbf{x}) = q(\mathbf{x})$. Define $H' = H \cup \{t+1\}$. Clearly $F_{H'}(Q_1) = F_H(Q) \cap \{x | q(x) < 0\}$ is nonempty and is an s-dimensional face of Q_1.

Define $\psi(F_H(Q)) = F_{H'}(Q_1)$.

It remains to show that the ψ constructed is an 1–1 mapping. It is easily seen that $\psi(F_H(Q)) \subseteq F_H(Q)$. Since all the $F_H(Q)$ in $\mathscr{F}_s(Q)$ are disjoint, it follows that all the $\psi(F_H(Q))$ are disjoint, hence distinct. This completes the proof of Lemma 1. □

It would be interesting to know if the same value of j can be used for every value of s in Lemma 1.

The adversary strategy $\mathscr{A}$. The adversary $\mathscr{A}$ will specify a way to answer questions with the help of a sequence of open polyhedra $V_0, V_1, V_2, \cdots$. Initially, $V_0 = Q$ where $Q = \{\mathbf{x} | l_i(\mathbf{x}) < 0, \ i = 1, 2, \cdots, m\}$. That Q is an open polyhedron (i.e., $Q \neq \emptyset$) is a consequence of the assumption that P has dimension n (see e.g. [8, Lemma (2.3.10)]). When the jth query "$q_j(x)$: 0" is asked, $\mathscr{A}$ has constructed $V_0, V_1, \cdots, V_{j-1}$. The adversary $\mathscr{A}$ will decide the outcome and construct V_j in the following way: Let $Q_1 = V_{j-1} \cap \{\mathbf{x} | q_j(\mathbf{x}) < 0\}$, and $Q_2 = V_{j-1} \cap \{\mathbf{x} | q_j(\mathbf{x}) > 0\}$; by Lemma 1, there is an $i \in \{1, 2\}$ such that $Q_i \neq \emptyset$, and $|\mathscr{F}_s(Q_i)| \geqq \frac{1}{2} |\mathscr{F}_s(V_{j-1})|$; the adversary's answer to the jth query is then "$q_j < 0$" if $i = 1$, and "$q_j > 0$" if $i = 2$; V_j is defined to be Q_i.

Analysis of the adversary strategy. Let $q_j(x)$: 0 $(j = 1, 2, \cdots, t)$ be the entire sequence of queries asked by the algorithm faced with outcomes determined by $\mathscr{A}$. Let $\varepsilon_j q_j(\mathbf{x}) < 0$ be the results of the queries $(\varepsilon_j = \pm 1)$. Then,

$$(1) \qquad V_t = \{\mathbf{x} | l_i(\mathbf{x}) < 0, i = 1, 2, \cdots, m, \ \varepsilon_j q_j(\mathbf{x}) < 0, j = 1, 2, \cdots, t\} \neq \emptyset$$

and

$$|\mathscr{F}_s(V_t)| \geqq \frac{1}{2} |\mathscr{F}_s(V_{t-1})| \geqq \frac{1}{2^2} |\mathscr{F}_s(V_{t-2})| \geqq \cdots \geqq \frac{1}{2^t} |\mathscr{F}_s(V_0)|, \text{ i.e.,}$$

$$(2) \qquad |\mathscr{F}_s(V_t)| \geqq \frac{1}{2^t} |\mathscr{F}_s(Q)|.$$

For each $\mathbf{x} \in V_t$, the same leaf in the tree T is reached and the algorithm must say "yes, $\mathbf{x} \in P$". Since the algorithm only knows that $\mathbf{x} \in \{\mathbf{x} | \varepsilon_j q_j(\mathbf{x}) < 0, j = 1, 2, \cdots, t\}$, we have

$$\{\mathbf{x} | \varepsilon_j q_j(\mathbf{x}) < 0, j = 1, 2, \cdots, t\} \subseteq P.$$

As Q is the "largest" open set contained in P, we have

$$\{\mathbf{x} | \varepsilon_j q_j(\mathbf{x}) < 0, j = 1, 2, \cdots, t\} \subseteq Q$$
$$= \{\mathbf{x} | l_i(\mathbf{x}) < 0, i = 1, 2, \cdots, m\}.$$

Therefore, (1) can be written as

$$V_t = \{\mathbf{x} \mid \varepsilon_j q_j(\mathbf{x}) < 0, j = 1, 2, \cdots, t\}. \tag{3}$$

As there are only t linear functions in (3), there can be at most $\binom{t}{n-s}$ s-dimensional faces of V_t. Therefore,

$$\binom{t}{n-s} \geqq |\mathscr{F}_s(V_t)|. \tag{4}$$

Equations (2) and (4) lead to

$$2^t \cdot \binom{t}{n-s} \geqq |\mathscr{F}_s(V_t)|. \tag{5}$$

As the left-hand side of (5) is an increasing function of t, and $C(P) \geqq t$, we have proved Theorem 1. □

4. Remarks. General discussions on the maximum number of faces that a polyhedron can have are given in [3] and [7]. As there can be $\approx \binom{m}{n-1}$ edges for certain polyhedra defined by m inequalities, the corollary to Theorem 1 establishes a lower bound of order $n \log m$ for, say $m > n^2$, to the corresponding polyhedral decision problem.

It would be interesting to find a "natural" problem in concrete computational complexity for which the bound of Theorem 1 yields a nontrivial (i.e., nonlinear) lower bound. In this regard we mention that, originally, it was hoped that the present approach would lead to an $\Omega(n^2 \log n)$ lower bound to the complexity of the all-pair shortest paths problem. That bound would follow if the *triangular polyhedron* $P^{(n)}$ in $R^{\binom{n}{2}}$, defined as $\{\mathbf{x} \mid \mathbf{x} = (x_{ij} \mid 1 \leqq i < j \leqq n);\ x_{ik} \geqq 0,\ x_{ij} + x_{jk} \geqq x_{ik}$ for all $1 \leqq i < k \leqq n$ and $1 \leqq j \leqq n\}$ (we define $x_{ii} = 0$ and $x_{ij} = x_{ji}$, if $i > j$), has at least $\exp(cn^2 \log n)$ edges[1]. However, it has recently been shown by Graham, Yao, and Yao [2] that $P^{(n)}$ has less than $\exp(cn^2)$ edges, with the implication that only a cn^2 lower bound can be obtained in this approach.

One candidate for the application of Theorem 1 is the problem of constructing optimal alphabetic trees [4], for which the best algorithm known has an $O(n \log n)$ running time. For a start, what is the number of edges in the polyhedron corresponding to deciding if a complete balanced tree is an optimal alphabetic tree? Another candidate is the verification problem for minimum spanning trees mentioned in the Introduction. It seems difficult, however, to obtain a nonlinear bound in this case, since the number of edges involved is no more than $\exp(cn \log^* n)$ (because the problem can be solved in $O(n \log^* n)$ by Tarjan's result [9]).

[1] See [11] for a proof of this statement. We remark that it was incorrectly stated in [11] that $P^{(n)}$ has provably $\exp(cn^2 \log n)$ edges.

REFERENCES

[1] D. P. DOBKIN, R. J. LIPTON AND S. P. REISS, *Excursions into geometry*, Computer Science Dept. Tech. Rep. 71, Yale University, New Haven, CT, 1976.

[2] R. L. GRAHAM, A. C. YAO AND F. F. YAO, *Information bounds are weak in the shortest distance problem*, Computer Science Dept. Rep. STAN-CS-78-670, Stanford University, Stanford, CA, 1978, J. Assoc. Comput. Mach., to appear.

[3] B. GRÜNBAUM, *Convex Polytopes*, Interscience, New York, 1967.

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

[5] M. O. RABIN, *Proving simultaneous positivity of linear forms*, J. Comput. Systems Sci., 6 (1972), pp. 639–650.

[6] O. SCHREIER AND E. SPERNER, *Modern Algebra and Matrix Theory*, translated by M. David and M. Hausner, Chelsea, New York, 1959.

[7] R. STANLEY, *The upper bound conjecture and Cohn–Macaulay rings*, Studies in Applied Math., 54 (1975), pp. 135–142.

[8] J. STOER AND C. WITZALL, *Convexity and Optimization in Finite Dimension I*, Springer Verlag, New York, 1970.

[9] R. E. TARJAN, *Applications of path compressions on balanced trees*, Computer Science Dept. Rep. STAN-CS-75-512, Stanford University, Stanford, CA, 1975.

[10] A. C. YAO, *On the complexity of comparison problems using linear functions*, Proc. 16th IEEE Ann. Symp. on Switching and Automata Theory, Berkeley, CA, 1975, pp. 85–89.

[11] A. C. YAO, D. M. AVIS AND R. L. RIVEST, *An* $\Omega(n^2 \log n)$ *lower bound to the shortest paths problem*, Proc. 9th ACM Annual Symposium on Theory of Computing, Boulder, CO, 1977, pp. 11–17.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0010 $01.00/0

PATH SYSTEMS: CONSTRUCTIONS, SOLUTIONS AND APPLICATIONS*

EITAN M. GURARI† AND OSCAR H. IBARRA‡

Abstract. We investigate the use of path systems in automata theory and computational complexity. A new framework is developed which brings together the main constructions of path systems corresponding to machine models as well as the main algorithms for solving such path systems. Applications to resource-bounded computation are given.

Key words. path system, computational complexity, resource-bounded computation, Turing machine, auxiliary pushdown automaton, auxiliary stack automaton, parallel Turing machine, alternating Turing machine, recursive Turing machine, restricted nondeterminism

1. Introduction. The notion of a path system was first introduced by Cook [4] in order to prove some results concerning certain types of pushdown machines. The main theorem in [4] is the following: Any language in LOG(*CFL*) (= the class of languages accepted by nondeterministic two-way multihead pushdown automata operating in polynomial time [17]) has deterministic tape complexity $(\log n)^2$. In another paper [5], Cook showed that the set SP of codings of solvable path systems is in $\mathcal{P}$ (=the class of languages accepted by deterministic polynomial time-bounded Turing machines) and any language in $\mathcal{P}$ is log-tape reducible to SP. Thus, SP is log-tape complete for $\mathcal{P}$. Recently, Sudborough [17] has described a path system problem which is log-tape complete for LOG(*CFL*).

In this paper, we investigate the use of path systems in automata theory and computational complexity. We develop a new framework which brings together the main constructions of path systems corresponding to machine models as well as the main algorithms for solving such path systems. This new treatment allows us to give unified proofs of well known results concerning resource-bounded computation as well as prove new theorems which sharpen and/or generalize these results. Examples of new theorems are the following:

THEOREM A. *Let $L(n) \geqq \log n$ and L be a language accepted by an $L(n)$-tape bounded nondeterministic (deterministic) auxiliary pushdown automaton* [3] *whose pushdown store makes at most $R(n) > 0$ reversals on inputs of length n. Then L can be accepted by an $L(n)\log(R(n))$-tape bounded nondeterministic (deterministic) Turing machine.*

THEOREM B. *Let L be a language accepted by a recursive Turing machine* [16] *M which makes at most $C(n) > 0$ recursive calls and has width $L(n) \geqq \log n$ (i.e., the storage space used per level of call is at most $L(n)$). If M is deterministic, then L has deterministic tape complexity minimum$\{L(n)+C(n), L(n)\log(C(n))\}$. If M is nondeterministic, then L has nondeterministic tape complexity $L(n)\log(C(n))$ (respectively, determinstic tape complexity $L(n)[L(n)+\log(C(n))]$).*

2. Parameterized path systems. We first recall the notion of a path system as defined by Cook [4].

* Received by the editors April 18, 1978 and in revised form April 23, 1979. This research was supported in part by the National Science Foundation under Grants DCR 75-17090 and MCS 78-01736.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota. Now at Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, Wisconsin 53201.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

DEFINITION. A *path system* is a 4-tuple $G = \langle N, T, S, R\rangle$, where N is a *finite* set (of *nodes*), $T \subseteq N$ is a set of *terminal nodes*, $S \subseteq N$ is a set of *source nodes*, and R is a *3-place relation* on N.

The *admissible* nodes of G are the least set N_a such that (i) $T \subseteq N_a$ and (ii) if β, γ are in N_a and (α, β, γ) is in R, then α is in N_a. G is *solvable* if and only if at least one admissible node is a source node. Solvability can also be defined in terms of digraphs and trees (see [4]).

In this paper, we consider a new formulation of a path system which we call parameterized path system (PPS). A single PPS defines a (possibly infinite) family of path systems uniformly parameterized in terms of a new set P. A member of the family can be specified by a tuple (x, α), where x is in P and α (the source node) is in N. Thus, R is now a relation on $P \times N \times N \times N$, where P and N are, in general, infinite. (Note that S is no longer needed.) For convenience, we also add to the definition of a PPS a new relation $R_1 \subseteq P \times N \times N$. The next definition makes these ideas precise.

DEFINITION. *A parameterized path system* (abbreviated *PPS*) is a 5-tuple $G = \langle P, N, T, R_1, R_2\rangle$, where P and N are countable sets of *parameters* and *nodes*, respectively, $T \subseteq N$ is a set of *terminal nodes*, R_1 and R_2 are *relations* with $R_1 \subseteq P \times N \times N$ and $R_2 \subseteq P \times N \times N \times N$.

We assume without loss of generality that P and N are sets of strings. For α in $P \cup N$, let $|\alpha|$, called the *size* of α, denote the length of α. We shall only deal with computable PPS's in that we assume the existence of the following algorithms:

(1) $P(\alpha)$, $N(\alpha)$, $T(\alpha)$. Given α, $P(\alpha)$, $N(\alpha)$, and $T(\alpha)$ return true or false depending on whether or not α is in P, N, and T, respectively.

(2) $R_1(x, \alpha, \beta)$. Given x in P and α, β in N, $R_1(x, \alpha, \beta)$ returns true or false depending on whether or not (x, α, β) is in R_1.

(3) $R_2(x, \alpha, \beta, \gamma)$. Given x in P and α, β, γ in N, $R_2(x, \alpha, \beta, \gamma)$ returns true or false depending on whether or not $(x, \alpha, \beta, \gamma)$ is in R_2.

(4) Routines needed for generating the elements of N systematically:

 (a) FIRST(m). Given a positive integer m, FIRST(m) returns the first node α in the ordering such that $|\alpha| \geqq m$. If no such node exists, FIRST(m) returns some distinguished symbol.

 (b) NEXT(α). Given α in N, NEXT(α) returns the node next to α in the ordering. If no next node exists, NEXT(α) returns some distinguished symbol.

We assume that each of the algorithms above can be implemented on a deterministic Turing machine with a two-way read-only input (with end-markers) and one read-write storage tape. For example, in the case of $R_2(x, \alpha, \beta, \gamma)$, the input is of the form $x \# \alpha \# \beta \# \gamma$ (# is some delimiter). The "true" or "false" output is indicated by the machine entering an accepting or nonaccepting state, respectively. The machine may also have a one-way write-only output tape as for example in the implementations of FIRST(m) and NEXT(α).

In this paper, we only study PPS's with the property that $P(\alpha)$, $N(\alpha)$, $T(\alpha)$, NEXT(α) have space complexity $O(l)$ and $R_1(x, \alpha, \beta)$, $R_2(x, \alpha, \beta, \gamma)$ have space complexity $O(l + \log|x|)$, where $l = \max\{|\alpha|, |\beta|, |\gamma|\}$. Similarly, FIRST($m$) has space complexity $O(|\text{FIRST}(m)|)$. These bounds are easily verified for all PPS's (except one) described in the paper. The proofs (except for one case, shown in Lemma 1, § 1) are therefore omitted.

DEFINITION. Let $G = \langle P, N, T, R_1, R_2\rangle$ be a PPS and (x, α) be in $P \times N$. A *solution tree for* (x, α) is a tree F_α^x (see Fig. 1) with root node α and leaves in T, and has the following property: If node β_1 has a single son β_2 then (x, β_1, β_2) is in R_1. If node γ_1 has two sons γ_2 and γ_3 then either $(x, \gamma_1, \gamma_2, \gamma_3)$ is in R_2 or $(x, \gamma_1, \gamma_3, \gamma_2)$ is in R_2.

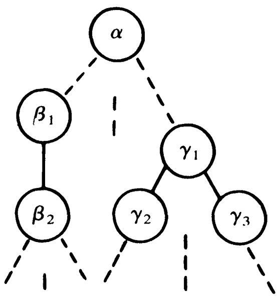


FIG. 1. *A solution tree* F_α^x.

Notation. Let $G=\langle P, N, T, R_1, R_2\rangle$ be a PPS and (x, α) be in $P\times N$. Let F_α^x be a solution tree for (x, α). Then

(a) WIDTH(F_α^x) = maximum size of a node in F_α^x;

(b) NODES(F_α^x) = number of nodes in F_α^x;

(c) LEAVES(F_α^x) = number of leaves in F_α^x;

(d) DEPTH(F_α^x) = number of levels in F_α^x, where the root α is at level 1.

3. An example. We now give an example to illustrate the formal PPS definition just given. The example concerns deterministic auxiliary pushdown automata (DAPA's) and nondeterministic auxiliary pushdown automata (NAPA's) [3]. A DAPA (NAPA) is a deterministic (nondeterministic) Turing machine with an auxiliary pushdown store.

Let M be a DAPA (NAPA). Without loss of generality we may assume that in every move, M pops the topmost symbol of the pushdown store, pushes exactly one symbol on top of the pushdown store or rewrites the topmost symbol (possibly by the same symbol). We shall refer to these moves as popping, pushing and rewriting moves. We also assume that in any computation, M executes at least one pushing move and the storage head does not write blanks. Furthermore, we assume that M accepts by entering a fixed state f with the input head on the left endmarker, the pushdown store containing only Z_0 (= the initial pushdown symbol) and the storage head on the leftmost nonblank symbol of the read-write tape.

A *partial configuration* (or simply, *p-configuration*) of M is a 3-tuple of the form $\alpha=(Z, i, uqv)$. α represents the situation in which M on some input x in $¢\Sigma^*\$$ is in state q, its input head is on the ith position, its storage tape contains uv with the storage head on the leftmost symbol of v, and the topmost symbol of the pushdown store is Z. For any p-configuration α, symbol(α) denotes the pushdown symbol of α. We shall construct from M a PPS G. To motivate the construction, we include the following brief discussion.

A computation of M on a given input x in $¢\Sigma^*\$$ can be described by a *time-space profile* (or simply, *profile*) which is a graph defined as follows (see Fig. 2): Coordinate (t, s), $t, s\geqq 1$, contains the p-configuration $\alpha=(Z, i, uqv)$ if and only if just before the tth move in the computation, the pushdown store had length s and M was in p-configuration α. We say that such an α is at *time t and distance s* and write time$(\alpha)=t$ and distance $(\alpha)=s$. The initial and final (i.e., accepting) p-configurations are $(Z_0, 1, q_0B)$ and $(Z_0, 1, fw)$, respectively, where B is the blank symbol and w is some nonnull storage string without blanks. Clearly, the input string and the profile fully describe a computation of M. Let α and β be p-configurations in the profile. There is a *path* from α to β if time$(\beta)\geqq$ time(α). The *length of the path*, denoted by length(α, β), is time$(\beta)-$time (α).

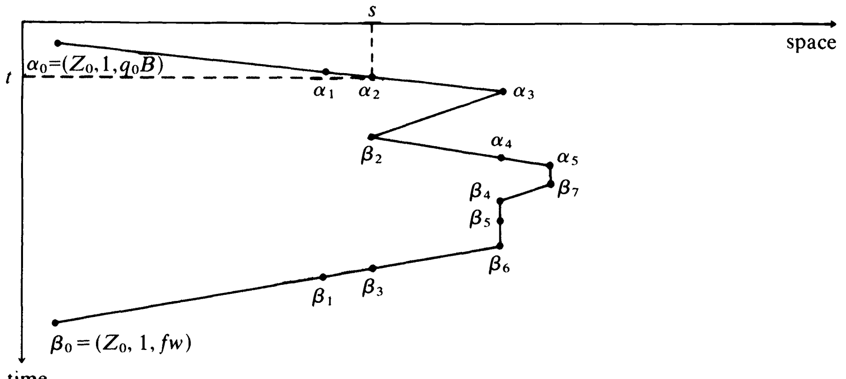


FIG. 2. *Profile of an accepting computation.*

Let α and β be p-configurations in the profile (not necessarily distinct). (α, β) is called a *pair at distance* s (or simply, *pair* if s is understood) if (i) distance$(\alpha)=$ distance$(\beta)=s$; (ii) there is a path from α to β; (iii) if a p-configuration γ is in the path from α to β, then distance$(\gamma)\geqq s$. If the length of the path from α to β is greater than 0, then (α, β) is called a *proper pair*. In Fig. 2, (α_0, β_0), (α_1, β_1), (α_2, β_2), (α_2, β_3), (β_2, β_3), (α_4, β_4), (α_4, β_5), (α_4, β_6), (β_4, β_5), (β_4, β_6), (β_5, β_6), (α_5, β_7) are proper pairs. The pairs $(\alpha_0, \alpha_0), \cdots, (\alpha_5, \beta_5), (\beta_0, \beta_0), \cdots, (\beta_7, \beta_7)$ are not proper pairs. A pair (α, α) is called *terminal* if (i) length $(\alpha, \alpha)=0$; (ii) there is a path from α to some p-configuration β with distance$(\beta)<$distance(α) and every p-configuration δ on this path has distance$(\delta)\leqq$ distance(α); (iii) there is a path from some p-configuration γ to α with distance$(\gamma)<$ distance(α) and every p-configuration δ on this path has distance$(\delta)\leqq$distance(α). In Fig. 2, the only terminal pairs are (α_3, α_3), (α_5, α_5) and (β_7, β_7).

We are now ready to define the PPS G corresponding to M. $G=\langle P, N, T, R_1, R_2\rangle$, where $P=¢\Sigma^*\$$, $N=\{(\alpha, \beta)|\alpha$ and β are p-configurations of $M\}\cup\{\nabla\}$, where ∇ is a new symbol (thus, each (α, β) in N is a possible pair in the profile of some computation), and $T=\{(\alpha, \alpha)|\alpha$ is a p-configuration$\}$ (each (α, α) in T is a possible terminal pair in the profile of some computation). R_1 and R_2 are defined as follows:

(1) For every x in P and nonnull storage string w without blanks, let $(x, \nabla, ((Z_0, 1, q_0B), (Z_0, 1, fw)))$ be in R_1.

(2) For every x in P and (α, β), (α', β') in N, let $(x, (\alpha, \beta), (\alpha', \beta'))$ be in R_1 if there exist nonnegative integers t_1, t_2 and a (possibly null) pushdown string y such that

(i) M on input x in p-configuration α can, in t_1 pushing and/or rewriting moves, enter α' with symbol(α) replaced by the string $y\cdot$ symbol(α'); and

(ii) M on input x in p-configuration β' and topmost pushdown string $y\cdot$ symbol(β') can, in t_2 popping and/or rewriting moves, enter β with $y\cdot$ symbol(β') replaced by symbol(β).

In Fig. 2, $(x, (\alpha_0, \beta_0), (\alpha_2, \beta_3))$, $(x, (\alpha_1, \beta_1), (\alpha_2, \beta_3))$, $(x, (\alpha_4, \beta_6), (\alpha_5, \alpha_5))$, $(x, (\alpha_4, \beta_4), (\alpha_5, \beta_7))$, $(x, (\alpha_5, \beta_7), (\alpha_5, \alpha_5))$, $(x, (\alpha_5, \beta_7), (\beta_7, \beta_7))$ are examples of triples in R_1. For the case when M is a NAPA, we require $t_1, t_2\leqq 1$.

(3) For every x in P and (α, β), (α', β'), (α'', β'') in N, let $(x, (\alpha, \beta), (\alpha', \beta'), (\alpha'', \beta''))$ be in R_2 if $\alpha=\alpha'$, $\beta'=\alpha''$ and $\beta=\beta''$. (Thus, R_2 allows for "splitting" of pairs.) In Fig. 2 $(x, (\alpha_2, \beta_3), (\alpha_2, \beta_2), (\beta_2, \beta_3))$ is in R_2.

The following lemma summarizes the properties of G that we will need in § 5.

LEMMA 1. *Let M be a NAPA (DAPA) and $G=\langle P, N, T, R_1, R_2\rangle$ be the PPS corresponding to M as described above. There are positive constants c_1 and c_2 with the following properties (x in $¢\Sigma^*\$$):*

(i) *If M is a NAPA, then M accepts x within storage space s and time t if and only if there is a solution tree for (x, ∇) with at most $c_1 t$ nodes and WIDTH at most $c_2(s+\log|x|)$.*

(ii) *If M is a NAPA, then M accepts x within storage space s and r pushdown reversals (r odd) if and only if there is a solution tree for (x, ∇) with at most $(r+1)/2$ leaves ($=$ terminal nodes) and WIDTH at most $c_2(s+\log|x|)$.*

(iii) *If M is a DAPA, then M accepts x within storage space s and r pushdown reversals (r odd) if and only if there is a solution tree for (x, ∇) with at most $2r+1$ nodes and WIDTH at most $c_2(s+\log|x|)$.*

(iv) *R_1, R_2, etc. have space complexity $O(l+\log|x|)$ for M a NAPA or a DAPA.*

Proof. (i) and (ii) are obvious from the construction of G. Now suppose that M is a DAPA and M accepts x within storage space s and r pushdown reversals. By (ii) there is a solution tree F_∇^x with at most $(r+1)/2$ leaves and WIDTH at most $c_2(s+\log|x|)$. Let this tree be minimal in the sense that no other solution tree for (x, ∇) with at most $(r+1)/2$ leaves and WIDTH at most $c_2(s+\log|x|)$ has fewer nodes. Then in F_∇^x there are no nodes (α_1, β_1), (α_2, β_2), (α_3, β_3) such that $(\alpha_{i+1}, \beta_{i+1})$ is the *only* son of (α_i, β_i), $i=1, 2$ (see Fig. 3(a)). Otherwise, F_∇^x is not minimal since we can delete node (α_2, β_2) and obtain a smaller solution tree (Fig. 3(b)). The upper bound of $2r+1$ on the number of nodes of F_∇^x can now easily be shown by induction on r. The converse of (iii) is trivial.

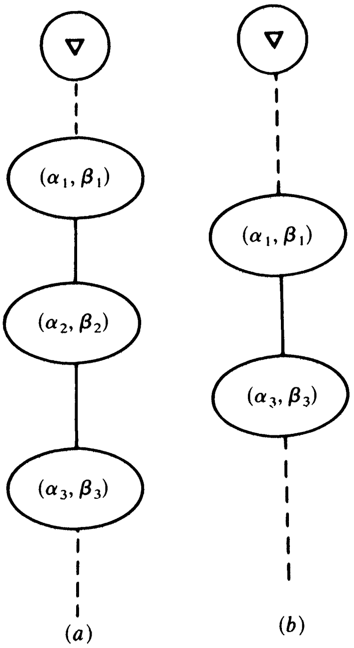


FIG. 3

We now prove (iv). We show that the bound holds for R_1 and M a DAPA. All other cases are obvious. So let x be in P and (α, β), (α', β') be in N. Clearly, $(x, (\alpha, \beta), (\alpha', \beta'))$ is in R_1 if and only if there exist two sequences of p-configurations $\alpha_0, \alpha_1, \cdots, \alpha_k$ and $\beta_m, \beta_{m-1}, \cdots, \beta_0$ satisfying

(a) $\alpha_0=\alpha$, $\alpha_k=\alpha'$;

(b) if $k>0$ then M on input x in p-configuration α_i enters, in one pushing or rewriting move, p-configuration α_{i+1}, $0\leqq i<k$;

(c) $\beta_m=\beta$, $\beta_0=\beta'$;

(d) if $m > 0$ then M on input x in p-configuration β_i enters, in one popping or rewriting move, p-configuration β_{i+1}, $0 \leqq i < m$. If the move is popping, then there exists an integer j such that

(1) $0 \leqq j < k$;

(2) $\text{distance}(\beta_{i+1}) = \text{distance}(\alpha_i)$, i.e., $\text{distance}(\beta') - \text{distance}(\beta_{i+1}) = (\text{distance }(\alpha') - \text{distance}(\alpha)) - (\text{distance}(\alpha_j) - \text{distance}(\alpha))$;

(3) $\text{symbol}(\beta_{i+1}) = \text{symbol}(\alpha_j)$; and

(4) M on input x in p-configuration α_j makes a pushing move.

Let $l_1 = |\alpha'|$ and $l_2 = |\beta|$. Since M does not write blanks on its storage tape, $|\alpha_i| \leqq l_1 + \log |x|$ for $0 \leqq i \leqq k$ and $|\beta_j| \leqq l_2 + \log |x|$ for $0 \leqq j \leqq m$. It follows that k is no greater than $c^{l_1 + \log|x|}$, where c is a constant which depends only on M. Hence, the sequence $\beta_m, \cdots, \beta_0$ corresponds to at most $c^{l_1 + \log|x|}$ popping moves while between any two such moves there are at most $c^{l_2 + \log|x|}$ rewriting moves. Therefore, $m \leqq c^{l_1 + l_2 + 2\log|x|}$. Let $l = \max\{|(\alpha, \beta)|, |(\alpha', \beta')|\}$. Obviously, $l \geqq \max\{l_1, l_2\}$. Then to determine if $(x, (\alpha, \beta), (\alpha', \beta'))$ is in R_1, one need only find the first $k \leqq c^{l + \log|x|}$ for which there exist $m \leqq c^{2(l + \log|x|)}$ and two sequences of p-configurations $\alpha_0, \cdots, \alpha_k$ and $\beta_m, \cdots, \beta_0$ satisfying conditions (a)–(d) above. Conditions (a) and (b) can easily be checked by just simulating k steps of M on input x starting in p-configuration α_0. To check conditions (c) and (d), the value of distance $(\alpha') - \text{distance}(\alpha)$ must first be computed. Then m moves of M on input x starting in p-configuration β_0 are simulated. To find symbol (β_{i+1}) for a p-configuration β_{i+1} which is entered directly after a popping move, M is simulated on input x starting in p-configuration α_0 until a p-configuration α_j satisfying conditions (1)–(4) of (*d*) is encountered. It follows that R_1 has space complexity $O(l + \log |x|)$. □

4. Algorithms for solving PPS's. In this section, we describe a number of algorithms for solving PPS's. From these algorithms, we derive upper bounds on the space and time complexity for determining the existence of solution trees having specified properties. The computing models that we use are:

(1) deterministic auxiliary pushdown automaton (DAPA) [3].

(2) nondeterministic auxiliary pushdown automaton (NAPA) [3].

(3) deterministic auxiliary stack automaton (DASA) [9]—a deterministic Turing machine (DTM) with an auxiliary stack tape. A stack is like a pushdown store except that the stack head can go inside the store in a read-only mode.

(4) parallel Turing machine (parallel TM) [11]—a parallel TM M is a nondeterministic Turing machine (NTM) with an associated function h: {states of M} → $\{\wedge, \vee\}$. With each partial configuration $\alpha = (i, uqv)$ of M, we associate a Boolean value B_α in $\{0, 1\}$, defined with respect to an input x in $\text{¢}\Sigma^*\$$, recursively as follows:

(i) $B_\alpha = 1$ if α is an accepting configuration (i.e., state(α) is an accepting state);

(ii) $B_\alpha = 0$ if α is a rejecting configuration (i.e., state(α) is a rejecting state);

(iii) If α is not an accepting or rejecting configuration and M in configuration α on input x can, in one step, enter configurations $\beta_1, \cdots, \beta_k$ whose associated Boolean values are defined then

$$B_\alpha = \begin{cases} B_{\beta_1} \wedge \cdots \wedge B_{\beta_k}, & \text{if } h(\text{state}(\alpha)) = \wedge, \\ B_{\beta_1} \vee \cdots \vee B_{\beta_k}, & \text{if } h(\text{state}(\alpha)) = \vee; \end{cases}$$

(iv) Otherwise, B_α is undefined.

We assume that accepting and rejecting configurations are halting configurations. M is said to *accept* x provided $B_{\alpha_0}=1$, where $\alpha_0=$ initial partial configuration $=(1, q_0B)$. Thus, an accepting computation of M on x can be represented by a *computation tree* whose nodes are pairs of the form (α, B_α) satisfying: (a) $(\alpha_0, 1)$ is the root; (b) the leaves are of the form $(\alpha, 1)$ or $(\alpha, 0)$ depending on whether α is an accepting or rejecting configuration; (c) if (α, B_α) is not a leaf and M in configuration α on input x can, in one step, enter configurations $\beta_1, \cdots, \beta_k$, then $(\beta_1, B_{\beta_1}), \cdots, (\beta_k, B_{\beta_k})$ are sons of (α, B_α) for some $B_{\beta_1}, \cdots, B_{\beta_k}$ in $\{0, 1\}$ and $B_\alpha = B_{\beta_1} \wedge \cdots \wedge B_{\beta_k}$ if $h(\text{state}(\alpha)) = \wedge$ and $B_\alpha = B_{\beta_1} \vee \cdots \vee B_{\beta_k}$ if $h(\text{state}(\alpha)) = \vee$.

The algorithms are of three types: balanced divide-and-conquer (DASA), filial divide-and-conquer (DAPA, NAPA, DASA, parallel TM) and dynamic programming (DASA).

4.1. Balanced divide-and-conquer. We begin with the following lemma which is similar to a result of Lewis, Stearns and Hartmanis [13] concerning derivations in context-free grammars. (See also [8, Lemma 11.1].)

LEMMA 2. *Let F_α be a binary tree with root α and NODES$(F_\alpha) = n \geqq 2$. Then there is a subtree F_β, with root β (see Fig. 4), such that NODES$(F_\beta) = m$ and $(1/4)n \leqq m \leqq (3/4)n$.*

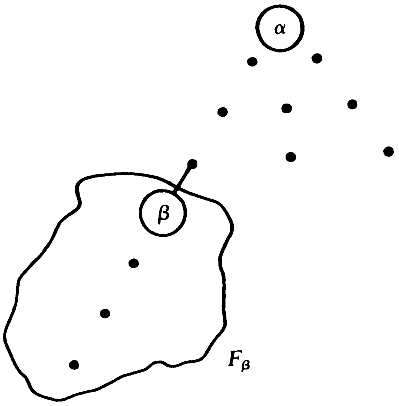


FIG. 4. *F_α has n nodes. F_β has m nodes, $(1/4)n \leqq m \leqq (3/4)n$.*

Proof. For $n<4$, the proof is obvious. So assume that $n \geqq 4$ and consider a tree F_α with NODES$(F_\alpha)=n$. Let ρ be a node in F_α such that F_ρ(= subtree formed by ρ) has NODES$(F_\rho) \geqq (3/4)n$. (Initially, we can take ρ to be α.) Then one of the following holds:

(1) ρ has one son, say γ. Then NODES$(F_\gamma) \geqq (1/4)n$. (Otherwise, $(3/4)n \leqq$ NODES$(F_\rho)=$ NODES$(F_\gamma)+1$, which implies $n<2$, a contradiction.)

(2) ρ has 2 sons. Clearly, at least one of them, say γ, has the property that NODES$(F_\gamma) \geqq (1/2)[\text{NODES}(F_\rho)-1] \geqq (1/2)[(3/4)n-1] \geqq (1/4)n$ (since $n \geqq 4$).

In either case, ρ has a son γ for which NODES$(F_\gamma) \geqq (1/4)n$. If NODES$(F_\gamma) \leqq (3/4)n$, we are done. Otherwise, the process can be repeated (with $\rho = \gamma$) until a node with the desired property is found. □

Our first theorem was motivated by similar results concerning the space complexity of the membership problem for context-free grammars [13], the complexity of recognizing solvable path systems [4], and the complexity of tape- and time-bounded auxiliary pushdown machines [7], [14]. A related problem was investigated in [18].

THEOREM 1. *Let $G=\langle P, N, T, R_1, R_2\rangle$ be a PPS. We can construct a DASA M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree for (x, α) with WIDTH at most l. If there exists a solution tree F^x_α with $WIDTH(F^x_\alpha)\leqq l$ and $NODES(F^x_\alpha)\leqq n$ then M uses at most storage space $O(l+\log|x|)$ and stack space $O(l\log n)$.*

Proof. Let us assume that there exists a solution tree F^x_α for (x, α) in $P\times N$ and $\text{NODES}(F^x_\alpha)=n$. Then, by Lemma 2, there exists a solution tree F^x_γ (for some γ in N) such that (i) F^x_γ is a subtree of F^x_α (see Fig. 5) and (ii) $m=\text{NODES}(F^x_\gamma)$ satisfies $(1/4)n\leqq m\leqq(3/4)n$.

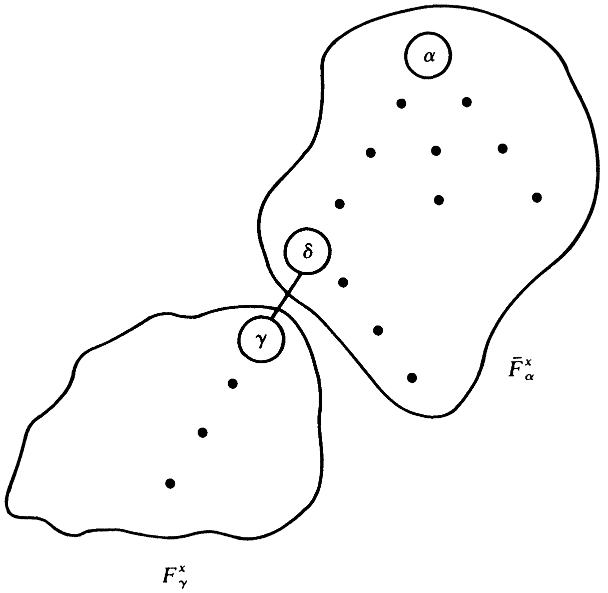


FIG. 5. *Decomposition of F^x_α into F^x_γ and $\bar{F}^x_\alpha$.*

Thus, a solution tree F^x_α with n nodes exists if and only if F^x_α can be decomposed into two trees F^x_γ and $\bar{F}^x_\alpha$ (for some γ in N) such that F^x_γ satisfies (i) and (ii), and $\bar{F}^x_\alpha$ is the tree with $\bar{m}=n-m$ nodes obtained by deleting the subtree F^x_γ from F^x_α. Clearly, $(1/4)n\leqq\bar{m}\leqq(3/4)n$. The idea of decomposing a solution tree into two smaller trees will be used in the recursive procedure that we now describe.

The procedure uses a global stack, S, which is initially empty. (As we shall see, S is, in general, not a pushdown store.) The recursive procedure uses a procedure $C(x, \alpha)$ which returns a value of true if α is a terminal or if (x, α, β_1) is in R_1 for some β_1 in the stack S or if $(x, \alpha, \beta_1, \beta_2)$ is in R_2 for some β_1 and β_2 in S. Otherwise, $C(x, \alpha)$ returns false. The recursive procedure is defined as follows.

```
procedure $A(x, \alpha, t, l, \beta)$:
    ||PUSH($\beta$) pushes $\beta$ on top of $S$ while POP($S$) pops the topmost symbol of $S$||
    PUSH($\beta$)
    if $C(x, \alpha)$ then [POP($S$); return ("true")]
    if $t=0$ then [POP($S$); return ("false")]
    for each $\gamma$ in $N$ such that $|\gamma|\leqq l$ do
```

```
        if $A(x, \gamma, t-1, l, \#)$ and $A(x, \alpha, t-1, l, \gamma)$ then [POP($S$); return ("true")]
    end
    POP($S$)
    return ("false")
end
```

Whenever $A(x, \alpha, t, l, \beta)$ is called, it determines the existence of a solution tree F_α^x with WIDTH$(F_\alpha^x) \leqq l$ and DEPTH$(F_\alpha^x) \leqq t$, where $T \cup S$ is used instead of T. To do so, A first checks whether $C(x, \alpha)$ is true. If this is not the case and $t > 0$ then A cycles through the γ's which are in N, $|\gamma| \leqq l$, and recursively calls $A(x, \gamma, t-1, l, \#)$ and $A(x, \alpha, t-1, l, \gamma)$ to determine the existence of trees F_γ^x and $\bar{F}_\alpha^x$ making up F_α^x. The parameter β is used to simplify the discussion below. $\#$ is a symbol not in N.

Given x in P, α in N and positive integer l, to determine the existence of a solution tree F_α^x with WIDTH$(F_\alpha^x) \leqq l$ and NODES$(F_\alpha^x) \leqq n$, we need only find the first $t \leqq \lceil \log_{4/3} n \rceil$ such that $A(x, \alpha, t, l, \#)$ returns the value true. Note that a maximum depth of recursion $\lceil \log_{4/3} n \rceil$ is sufficient for determining the existence of a solution tree F_α^x with NODES$(F_\alpha^x) \leqq n$. The reason for this is that, by Lemma 2, at the tth level of recursion, the solution tree has no more than $(3/4)^t n$ nodes. Thus, by the time, the $\lceil \log_{4/3} n \rceil$th level is reached the solution tree has at most 1 node.

A nonrecursive procedure implementing the procedure $A(x, \alpha, t, l, \beta)$ can easily be written using standard techniques (see, e.g., [1]). The implementation uses a pushdown store, say Q, in which are stored the data used by each call of A which has not yet been satisfied. However, since x and l are global values, they need not be stored in the pushdown store. The same is true for t which measures the depth of recursion. Thus, the pushdown store Q holds pairs of the form (α, β), α and β in $N \cup \{\#\}$, $|\alpha|, |\beta| \leqq l$. We also note that at any time, the information needed in the global stack S is already contained in the pushdown store Q. So we can use Q for S. Now, while information is added (deleted) only on (from) the top of Q, the procedure $C(x, \alpha)$ may need to read information inside Q. Thus, Q is, in general, a stack. From the preceding discussion, we see that Q will contain at most $O(\log n)$ nodes of size at most l. It follows that the space needed in Q is at most $O(l \log n)$. A storage space of $O(l + \log |x|)$ is sufficient to check R_1, R_2, etc. Moreover, the space needed to store the value of the first $t \leqq \lceil \log_{4/3} n \rceil$ such that $A(x, \alpha, t, l, \#)$ returns "true" is at most $\log(\lceil \log_{4/3} n \rceil)$. By Lemma 3 below, we can assume that $n \leqq 2^{c^l}$ for some constant c. Hence, the space necessary to store t is at most $O(l)$. Thus, a DASA M with the desired property can be constructed. □

LEMMA 3. *Let F_α^x be a minimal solution tree for (x, α) such that WIDTH$(F_\alpha^x) \leqq l$. (Here, minimal means that F_α^x has the least number of nodes among all solution trees for (x, α) with WIDTH $\leqq l$.) Then DEPTH$(F_\alpha^x) \leqq c^l$ and NODES$(F_\alpha^x) \leqq 2^{c^l}$ for some constant c.*

Proof. Since F_α^x is minimal, DEPTH$(F_\alpha^x) \leqq c^l$, where c is the least positive integer such that $c^l >$ number of nodes in N with size $\leqq l$. Hence, NODES$(F_\alpha^x) \leqq 1 + 2 + \cdots + 2^{c^l - 1} \leqq 2^{c^l}$. □

COROLLARY 1. *Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS. We can construct a DASA M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree for (x, α) with WIDTH at most l. If there exists a solution tree F_α^x with WIDTH$(F_\alpha^x) \leqq l$, LEAVES$(F_\alpha^x) \leqq r$ and DEPTH$(F_\alpha^x) \leqq d$ then M uses at most storage space $O(l + \log |x|)$ and stack space $O(l[\log d + \log r])$.*

Proof. If LEAVES$(F_\alpha^x) \leqq r$ and DEPTH$(F_\alpha^x) \leqq d$, then NODES$(F_\alpha^x) \leqq dr$. The result follows from Theorem 1. □

From Lemma 3 and Corollary 1 we have

COROLLARY 2. *Let $G = \langle P, N, T, R_1, R_2\rangle$ be a PPS. We can construct a DASA M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree for (x, α) with WIDTH at most l. If there exists a solution tree F_α^x with WIDTH$(F_\alpha^x) \leqq l$ and LEAVES$(F_\alpha^x) \leqq r$ then M uses at most storage space $O(l + \log |x|)$ and stack space $O(l[l + \log r])$.*

4.2. Filial divide-and-conquer. It seems very difficult to improve the bound of $O(l \log n)$ in Theorem 1, even if we were to replace the stack by another TM storage tape and make the machine nondeterministic. Such an improvement would mean a sharpening of the currently best known bound of $(\log n)^2$ for the nondeterministic tape complexity of context-free languages [13]. However, Corollary 2 can be improved if the machine is nondeterministic.

THEOREM 2. *Let $G = \langle P, N, T, R_1, R_2\rangle$ be a* PPS. *We can construct a NAPA M which when given x in P and α in N can determine the existence of a solution tree for (x, α). If there exists a solution tree F_α^x with WIDTH$(F_\alpha^x) \leqq l$ and LEAVES$(F_\alpha^x) \leqq r$ then M has an accepting computation in which it uses at most storage space $O(l + \log |x|)$ and pushdown space $O(l \log r)$.*

Proof. Follows from the procedure described below and Lemma 4. □

```
procedure $B(x, \alpha)$:
    ||$S$ is a pushdown store which is initially empty. PUSH($\beta$) pushes $\beta$ on top of $S$
    while POP($S$) pops and returns the top element of $S$||
    $Z \leftarrow \alpha$;
    repeat
      case
        1. $Z$ is terminal: if $S = \varnothing$ then return ("true") else
                                   $Z \leftarrow$ POP($S$)
        2. $Z$ is nonterminal: nondeterministically do (i) or (ii)
           (i) Guess a node $\beta$; if $R_1(x, \alpha, \beta)$ then $Z \leftarrow \beta$ else return
               ("false")||Guess that $\alpha$ has 1 son||
           (ii) Guess nodes $\beta$ and $\gamma$; if $R_2(x, \alpha, \beta, \gamma)$ or $R_2(x, \alpha, \gamma, \beta)$ then
               [PUSH($\beta$); $Z \leftarrow \gamma$] else return ("false") ||Guess that $\alpha$ has 2 sons||
      end
    forever
end
```

Clearly, the pushdown space needed by the procedure on input (x, α) is WIDTH$(F_\alpha^x) \cdot t$, where t = maximum number of nodes stacked in S. Theorem 2 then follows from the following lemma.

LEMMA 4. *Let $G = \langle P, N, T, R_1, R_2\rangle$ be a PPS and (x, α) be in $P \times N$. If a solution tree F_α^x exists, then procedure B has an accepting computation (i.e., a computation with output "true") in which the pushdown S never contains more than $\lfloor \log_2 LEAVES(F_\alpha^x)\rfloor$ nodes.*

Proof. Let r = LEAVES(F_α^x). The argument is an induction on r. If $r = 1$, then the procedure need only use cases 1 and 2(i) and no node is stacked. Hence, the lemma is true for $r = 1$. Now assume that $r > 1$ and the lemma is true for all solution trees with less than r leaves. Consider a solution tree F_α^x with r leaves. Let β be the node nearest the root, α, which has 2 sons. (β exists since $r > 1$. See Fig. 6.) Let γ and δ be the sons of β and let F^x_γ and F^x_δ be the subtrees formed by γ and δ, respectively. Let r_1 and r_2 be the number of leaves of F^x_γ and F^x_δ, respectively. Clearly, $r_1 \neq 0$, $r_2 \neq 0$ and $r_1 + r_2 = r$. Moreover, either $r_1 \leqq r/2$ or $r_2 \leqq r/2$. Without loss of generality assume that F^x_γ has $r_1 \leqq r/2$ leaves. Looking now at procedure B, we see that before node β is reached, S is empty. When node β is reached, the procedure can nondeterministically stack node δ and process subtree F^x_γ. By induction hypothesis, no more than $1 + \lfloor \log_2 r_1 \rfloor \leqq 1 + \lfloor \log_2 r/2 \rfloor = \lfloor \log_2 r \rfloor$ nodes are stacked in S when F^x_γ is being processed. After processing F^x_γ, the procedure can then pop node δ and process subtree F^x_δ. Again, by induction hypothesis, processing of F^x_δ needs no more than $\lfloor \log_2 r_2 \rfloor \leqq \lfloor \log_2 r \rfloor$ nodes to be stacked in S. The result follows. □

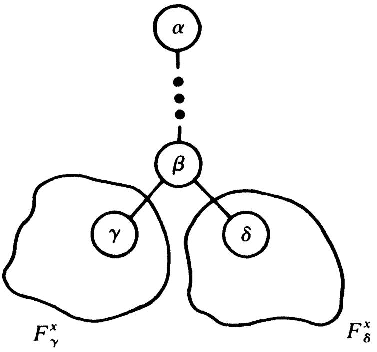


FIG. 6. *F^x_α with r leaves.*

Notation. Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS. For x in P, positive integer l and α in N such that $|\alpha| \leqq l$, define the set of strings $H(x, \alpha, l) = \{\beta | \beta \text{ in } N, |\beta| \leqq l \text{ and } (x, \alpha, \beta) \text{ in } R_1\} \cup \{\beta \# \gamma | \beta, \gamma \text{ in } N, |\beta|, |\gamma| \leqq l \text{ and } (x, \alpha, \beta, \gamma) \text{ in } R_2\}$, where # is a new symbol. Assume that the elements of $H(x, \alpha, l)$ are ordered lexicographically so that it makes sense to refer to the jth element in the set. Define the function $S(x, \alpha, l, i, j)$ as follows: For x in P, α in N, $i = 1$ or 2, and positive integers l and j, let

$$S(x, \alpha, l, i, j) = \begin{cases} \beta, & \text{if } i = 1 \text{ and the } j\text{th element in } H(x, \alpha, l) \text{ is } \beta \text{ or} \\ & \beta \# \gamma \text{ for some } \gamma, \\ \gamma, & \text{if } i = 2 \text{ and the } j\text{th element in } H(x, \alpha, l) \text{ is } \beta \# \gamma \\ & \text{for some } \beta, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that $S(x, \alpha, l, 1, j)$ is undefined if and only if $H(x, \alpha, l)$ has no jth element.

If F^x_α is a solution tree for (x, α), let $\text{CHOICE}(F^x_\alpha) = \max\{\text{cardinality of } H(x, \beta, l) | \beta \text{ is a node in } F^x_\alpha\}$.

Our next result concerns solution trees with restrictions on $\text{CHOICE}(F^x_\alpha)$ and $\text{DEPTH}(F^x_\alpha)$.

THEOREM 3. *Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS. We can construct a DASA (respectively, DAPA) M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree for (x, α) with WIDTH at most l. If there exists a solution tree F^x_α with $WIDTH(F^x_\alpha) \leqq l$, $CHOICE(F^x_\alpha) \leqq b$ and $DEPTH(F^x_\alpha) \leqq d$ then M uses at most storage space $O(l + \log |x|)$ and stack space $O(d \log b)$ (respectively, pushdown store space $O(d[\log b + h])$ for some h). The case of DAPA assumes the. existence of partial functions f and g computable in $O(l + \log |x|)$-storage space and a positive constant h such that if $S(x, \beta, l, i, j) = \gamma$ then $f(x, \beta, l, i, j)$ is defined, $|f(x, \beta, l, i, j)| \leqq h$ and $g(x, \gamma, l, i, j, f(x, \beta, l, i, j)) = \beta$. The range of f is Δ^* for some alphabet Δ. (Intuitively, f is a function of x, β, l, i and j such that if $S(x, \beta, l, i, j) = \gamma$, then* *the value of f can be used by g to construct β out of x, γ, l, i and j. Moreover, the value of f requires at most storage space h. Thus, $f(\cdot)$ is a short encoded memory of β. Trivial functions are: $f(x, \beta, l, i, j) = \beta$ and $g(x, \gamma, l, i, j, \beta) = \beta$ if $|\beta|$, $|\gamma| \leqq l$ and undefined otherwise. In this case, $h = l$. Hence, we may assume that $h \leqq l$ in the general case.)*

Proof. Clearly, a solution tree F^x_α with WIDTH$(F^x_\alpha) \leqq l$, CHOICE $(F^x_\alpha) \leqq b$ and DEPTH$(F^x_\alpha) \leqq d$ exists if and only if $H(x, \alpha, l)$ has at most b elements and one of the following holds:

(i) α is a terminal node;

(ii) there is a node β in N such that (x, α, β) is in R_1 and (x, β) has a solution tree F^x_β with WIDTH$(F^x_\beta) \leqq l$, CHOICE$(F^x_\beta) \leqq b$ and DEPTH$(F^x_\beta) \leqq d-1$;

(iii) there are nodes β and γ in N such that $(x, \alpha, \beta, \gamma)$ is in R_2 and (x, β), (x, γ) have solution trees F^x_β, F^x_γ with WIDTH(F^x_β), WIDTH$(F^x_\gamma) \leqq l$, CHOICE(F^x_β), CHOICE$(F^x_\gamma) \leqq b$ and DEPTH(F^x_β), DEPTH$(F^x_\gamma) \leqq d-1$.

We shall describe a recursive procedure based on the above observation. The procedure uses a global stack Q (initially empty) and two procedures: a function SON(x, Z, l, i, j) and a subroutine SON$^{-1}(x, Z, l)$. These procedures are defined as follows (Z is a variable and val(Z) denotes its value):

(1) SON(x, Z, l, i, j). Suppose val$(Z) = \alpha$. Then SON returns "true" if $S(x, \alpha, l, i, j) = \beta$ and, as a side-effect, sets the value of Z to β. In addition, the triple $(i, j, f(x, \alpha, l, i, j))$ is stored on top of the global stack Q. (For notational convenience, define $f(\cdot) =$ null string if M is a DASA.) If $S(x, \alpha, l, i, j)$ is undefined, SON returns "false" without changing the value of Z and the contents of Q.

The computation of SON(x, Z, l, i, j) involves generating strings of length $\leqq 2l+1$. Each string generated is checked if it is of the form β with $|\beta| \leqq l$ (or of the form $\beta \# \gamma$ with $|\beta|$, $|\gamma| \leqq l$). For such a string, the procedure then determines whether (x, α, β) is in R_1 (or $(x, \alpha, \beta, \gamma)$ is in R_2). The jth such string can be found (if it exists) using a counter requiring space $O(\log b)$. Now $j \leqq b \leqq c^l$ and f, R_1, R_2, etc., are computable in space $O(l + \log |x|)$. It follows that SON has space complexity $O(l + \log |x| + \log b) = O(l + \log |x|)$.

(2) SON$^{-1}(x, z, l)$. Suppose val$(Z) = \beta$. Then SON^{-1} when called returns with the value of Z set to α, where α is the "historical" father of a particular instance of β.

The computation of SON^{-1} (x, Z, l) depends on the type of M being constructed:

(a) If M is a DASA, α is obtained by retracing the path from the root to node β. The information needed to determine this path is on the global stack Q in the form of a string of triples: $(i_1, j_1, f(\cdot))$ $(i_2, j_2, f(\cdot)) \cdots (i_k, j_k, f(\cdot))(i_{k+1}, j_{k+1}, f(\cdot))$. These triples were stored by procedure SON and in particular, $(i_{k+1}, j_{k+1}, f(\cdot))$ was stored when the value of Z was changed from α to β. When α is found, $(i_{k+1}, j_{k+1}, f(\cdot))$ is deleted from the stack. It is straightforward to verify that SON^{-1} has storage space complexity $O(l + \log |x|)$.

(b) If M is a DAPA, then α is obtained by simply deleting the topmost triple $(i_{k+1}, j_{k+1}, f(\cdot))$ from the global stack and computing $g(x, \beta, l, i_{k+1}, j_{k+1}, f(\cdot))$. Again, it is clear that SON^{-1} has storage space complexity $O(l + \log |x|)$.

The recursive procedure is given by the following program.

```
procedure $D(x, Z, s, t, l)$:
  if val$(Z)$ is a terminal node then return ("true")
  for $j \leftarrow 1$ to $s$ do
    if not SON$(x, Z, l, 1, j)$ then return ("false")      ||Suppose $\alpha$ is the value of $Z$. If $H(x, \alpha, l)$ does not have a $j$th element then return false||
    if $D(x, Z, s, t-1, l)$ then                           ||Now $Z$ has value $\beta$, where $\beta$ is the first son of $\alpha$, and a valid $F^x_\beta$ exists||
      [call SON$^{-1}(x, Z, l)$                             ||Put $\alpha$ back to $Z$||
      if not SON$(x, Z, l, 2, j)$ then return ("true")     ||$(x, \alpha, \beta)$ is in $R_1$, i.e., $\alpha$ has no second son||
      if $D(x, Z, s, t-1, l)$ then                         ||Now $Z$ has value $\gamma$, where $\gamma$ is the second son of $\alpha$, and a valid $F^x_\gamma$ exists||
         [call SON$^{-1}(x, Z, l)$; return ("true")]
      else call SON$^{-1}(x, Z, l)$]                        ||A valid $F^x_\gamma$ does not exist. Put $\alpha$ back to $Z$||
    else call SON$^{-1}(x, Z, l)$                           ||A valid $F^x_\beta$ does not exist. Put $\alpha$ back to $Z$||
  end
  return ("false")
end
```

Whenever $D(x, Z, s, t, l)$ is called, it determines the existence of a solution tree F^x_α with WIDTH$(F^x_\alpha) \leqq l$, CHOICE$(F^x_\alpha) \leqq s$ and DEPTH$(F^x_\alpha) \leqq t$, where $\alpha = \text{val}(Z)$. Hence, to determine the existence of a solution tree F^x_α with the desired properties, we find s and t for which $t \log s$ is the smallest and $D(x, Z, s, t, l)$ with Z set to α returns "true". Clearly, D can be implemented using a DASA or a DAPA depending on whether the global stack Q is operated as described in case (a) or (b) in the computation of SON^{-1}. A pushdown store to implement the recursion is not needed since x, Z, s and l are global and need not be stacked. The parameter t which measures the maximum depth of recursion is also not stacked, while the value of the local variable j is already recorded in the global stack, Q. Clearly, if there exists a solution tree F^x_α with WIDTH$(F^x_\alpha) \leqq l$, CHOICE$(F^x_\alpha) \leqq b$ and DEPTH$(F^x_\alpha) \leqq d$ then Q needs at most space $O(d \log b)$ in the case of a DASA and $O(d[\log b + h])$ in the case of a DAPA. Now b, $d \leqq c^l$ for some constant c. Hence the space required to store s and t is at most $O(l)$. Since SON and SON^{-1} have space complexity $O(l + \log |x|)$, M needs at most storage space $O(l + \log |x|)$. □

COROLLARY 3. *Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS. We can construct a DAPA M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree for (x, α) with WIDTH at most l. If there exists a solution tree F^x_α with WIDTH$(F^x_\alpha) \leqq l$ and DEPTH$(F^x_\alpha) \leqq d$ then M uses at most storage space $O(l + \log |x|)$ and pushdown store space $O(ld)$.*

Proof. If F^x_α is a solution tree with WIDTH$(F^x_\alpha) \leqq l$, then CHOICE$(F^x_\alpha) \leqq c^l$ for some constant c. Then in Theorem 3, let $b = c^l$ and define the functions f and g by: For all x in P, β and γ in N, $i = 1, 2$, and positive integers l and j, $f(x, \beta, l, i, j) = \beta$ and $g(x, \gamma, l, i, j, \beta) = \beta$ if $|\beta|$, $|\alpha| \leqq l$ and undefined otherwise. □

The next result is concerned with solving PPS's by parallel TM's.

THEOREM 4. *Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS. We can construct a parallel TM*

M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree F_α^x with $WIDTH(F_\alpha^x) \leqq l$ within storage space $O(l+\log|x|)$.

Proof. The operation of M is described by the following program.

```
Phase 1. Write $\alpha$ on the storage tape and proceed to phase 2.
Phase 2. Let $\rho$ be the node on the storage tape. If $\rho$ is a terminal node, then accept;
         otherwise, go to phase 3.
Phase 3. Execute (1) or (2)
         (1) Nondeterministically write $\#\beta$ to the right of $\rho$, where $\#$ is a
             delimiter and $\beta$ is some node. Check if $(x, \rho, \beta)$ is in $R_1$. If so, replace
             $\rho\#\beta$ by $\beta$ and go to phase 2; otherwise, reject.
         (2) Nondeterministically write $\#\beta\#\gamma$ to the right of $\rho$, where $\beta$ and $\gamma$
             are nodes. Check if $(x, \rho, \beta, \gamma)$ is in $R_2$. If so, go to phase 4; otherwise,
             reject.
Phase 4. Assume that this phase is always entered in a disti guished state $p$. Then
         enter state $p_1$ or state $p_2$.
         If in state $p_1$, replace $\rho\#\beta\#\gamma$ by $\beta$ and go to phase 2.
         If in state $p_2$, replace $\rho\#\beta\#\gamma$ by $\gamma$ and go to phase 2.
```

Construct a NTM M from the above algorithm. We may assume that M always halts. (M need only check that phase 2 is entered at most c^l times, where c is the least positive integer such that $c^l \geqq$ number of nodes in N with size $\leqq l$.) Let K be the state set of M. Note that p is in K. Then M becomes the desired parallel TM by defining the function h as follows: $h(p) = \wedge$ and $h(s) = \vee$ for each s in $K - \{p\}$. □

4.3. Dynamic programming.

Definition. Let $G = \langle P, N, T, R_1, R_2 \rangle$ be a PPS and (x, α) be in $P \times N$. A *solution graph* for (x, α) is a finite rooted directed acyclic graph G_α^x where

(1) α (the root node) has no incoming edges:

(2) Every node β is either terminal or there is at least one edge leading out from β.

(3) If β has only one direct successor, say γ, then (x, β, γ) is in R_1.

(4) If β has two direct successors, say γ and δ, then $(x, \beta, \gamma, \delta)$ or $(x, \beta, \delta, \gamma)$ is in R_2.

An example of a solution graph G_α^x is shown in Fig. 7.

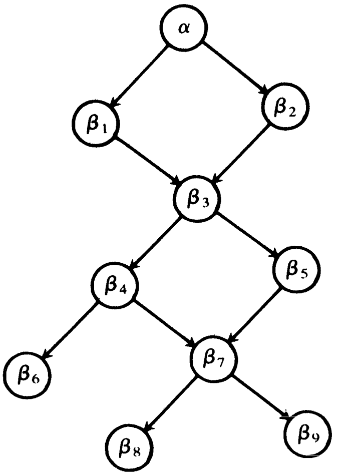


Fig. 7. *A solution graph G_α^x.*

Clearly, by duplicating nodes, we can convert any solution graph G_α^x into a solution tree F_α^x. Thus, we have the following proposition.

PROPOSITION. *Let G be a PPS. (x, α) has a solution tree if and only if it has a solution graph.*

THEOREM 5. *Let $G = \langle P, N, T, R_1, R_2\rangle$ be a PPS. We can construct a DASA M which when given x in P, α in N and positive integer l as inputs can determine the existence of a solution tree F^x_α with $WIDTH(F^x_\alpha) \leqq l$ within storage space $O(l + \log|x|)$ and time $O(c^{l+\log|x|})$ for some constant c. Moreover, M has nonerasing auxiliary stack tape.*

Proof. By the proposition, it is sufficient to determine for (x, α) in $P \times N$ whether or not a solution graph G^x_α exists. We describe an algorithm similar to one in [5]. The algorithm consists of first generating and inserting in the stack all nodes in T of size at most l. A storage space of $O(l)$ is used in the generation. Then for each node β not in the stack, $|\beta| \leqq l$, the stack is examined for a node γ such that (x, β, γ) is in R_1 or for nodes γ and δ such that $(x, \beta, \gamma, \delta)$ is in R_2. In either case, β is then inserted in the stack. A storage space of $O(l + \log|x|)$ is needed here. The process of inserting nodes in the stack is continued until no new node can be put in. Note that the number of nodes of size at most l that can be inserted in the stack is $O(c_1^l)$ for some c_1. Then the algorithm returns "true" if and only if α appears in the stack. It is clear that the algorithm can be implemented on a nonerasing DASA which operates within $O(l + \log|x|)$ storage space and $O(c^{l+\log|x|})$ time. □

5. Applications. In this section, we show how PPS's can be used to unify the proofs of several well-known results concerning resource-bounded computation. In addition, we obtain new theorems which sharpen and/or generalize previously known results.

DEFINITION. Let $L(n)$ be a function on the positive integers. A NTM (DTM, DAPA, NAPA, DASA, parallel TM) is *$L(n)$-tape bounded* if every input of length n that is accepted has some accepting computation which uses no more than $L(n)$ space on its read-write storage tape. (Note that $L(n)$ does not include the space used in the pushdown store or the stack.) $L(n)$ is *tape constructible* if there is an $L(n)$-tape bounded DTM U which halts for all inputs and which has the property that for each n, every input of length n enables U to scan exactly $L(n)$ cells on its storage tape.

Convention. There are other resource bounds (e.g., time, number of reversals, number of nondeterministic moves, etc.) that are considered in the paper. As in the above definition, these bounds only apply to inputs that are accepted. When the device is nondeterministic and has several resource constraints we require that the bounds hold simultaneously for at least one accepting computation. All the functional bounds in the paper ($L(n)$, $T(n)$, $R(n)$, $C(n)$, and $D(n)$) are functions on the positive integers. Throughout, we assume that $L(n) \geqq \log n$. To simplify proofs, we also assume that $L(n)$ is tape constructible for results involving deterministic simulators. This assumption can be removed since we can modify the simulators to iterate the computations for $L(n) = 1, 2, 3, \cdots$ until an accepting configuration is reached. Note, however, that the modified machines may not halt for inputs that are not accepted.

5.1. Auxiliary pushdown automata and Turing machines. Our first theorem concerns languages accepted by $L(n)$-tape bounded NAPA's which operate within $T(n)$ time.

THEOREM 6. *Let L be accepted by an $L(n)$-tape bounded NAPA M_1 which operates within $T(n)$ time. Then L can be accepted by an $L(n)$-tape bounded DASA M_2 whose stack uses no more than $L(n)\log(T(n))$ space.*

Proof. Given M_1, construct a PPS G as described in § 3. By Lemma 1(i) there are positive constants c_1 and c_2 such that M_1 accepts an input x in $¢\Sigma^*\$$ within $L(n)$ space and $T(n)$ time ($n = |x|$) if and only if there exists a solution tree for (x, ∇) with at most $c_1T(n)$ nodes and WIDTH at most $c_2(L(n) + \log n)$. Let M be the DASA corresponding to G of Theorem 1 (balanced divide-and-conquer). We now describe the operation of M_2.

M_2 with input x first divides the storage tape into two tracks. On track 1 it writes the string $\nabla \# 1^{L(n)}$. Then M_2 uses the second track of the storage tape in simulating the computation of M on x, ∇ and the positive integer represented by $1^{L(n)}$. From Theorem 1 (balanced divide-and-conquer) and the assumption that $L(n) \geqq \log n$, it follows that M_2 can be constructed to be $L(n)$-tape bounded and its stack uses no more than $L(n) \log (T(n))$ space. □

COROLLARY 4. *Let L and M_1 be as in Theorem 6. Then L can be accepted by an* $L(n) \log (T(n))$-tape *bounded DTM* M_2.

Corollary 4 has been reported earlier by Monien [14]. His proof is a direct DTM construction. Though the basic idea in his proof resembles the $(\log n)^2$ construction of Lewis, Stearns and Hartmanis [13], it is quite hard to follow.

If in Theorem 6, we require M_2 to be a DAPA, we have the following result.

THEOREM 7. *Let L be accepted by an $L(n)$-tape bounded NAPA M_1 which operates within $T(n)$ time. Then L can be accepted by an $L(n)$-tape bounded DAPA M_2 whose pushdown store uses no more than $L(n)T(n)$ space.*

Proof. The proof is similar to that of Theorem 6, this time using Corollary 3 (filial divide-and-conquer). Note that a solution tree with $O(T(n))$ nodes has depth $O(T(n))$. □

Recently, Harju [7] claimed a result stronger than Theorem 7: M_2 need only use $L(n) \log (T(n))$ pushdown space.

We can also prove the following corollary using Lemma 3, Corollary 3 (filial divide-and-conquer) and Theorem 5 (dynamic programming).

COROLLARY 5. *Let L be accepted by an $L(n)$-tape bounded NAPA. Then*

(i) *L can be accepted by an $L(n)$-tape bounded DAPA whose pushdown store uses no more than $c^{L(n)}$ space for some constant c.*

(ii) *L can be accepted by an $L(n)$-tape bounded nonerasing DASA which operates within $c^{L(n)}$ time for some constant c.*

From Corollary 5(i), we get the following result first proved by Cook [3].

COROLLARY 6. *A language L is accepted by an $L(n)$-tape bounded NAPA if and only if it is accepted by an $L(n)$-tape bounded DAPA.*

The next two corollaries are immediate from Corollary 4. They were first shown by Lewis, Stearns and Hartmanis [13] and Savitch [15], respectively.

COROLLARY 7. *If L is a context-free language (i.e., accepted by a real-time pushdown automaton), then L has deterministic tape complexity at most* $(\log n)^2$.

COROLLARY 8. *Let L be accepted by an $L(n)$-tape bounded NTM which operates within $T(n)$ time. Then L has deterministic tape complexity* $L(n) \log (T(n))$.

As another application of PPS's, we prove another result of Cook [3], [5]. The proof follows the ideas in [5].

THEOREM 8. *L is accepted by a $c^{L(n)}$-time bounded DTM (c a postive constant) if and only if it is accepted by an $L(n)$-tape bounded DAPA.*

Proof. The "if" part follows from Corollary 5(ii). Now suppose L is accepted by a $c^{L(n)}$-time bounded DTM M. We may assume that M is single-tape (i.e., the input tape is also the read-write storage tape) with a left endmarker. We also assume that M does not write blanks, has no stationary moves, and only reverses its head on the left endmarker or on the blank to the right of the nonblank portion of the tape. Finally, we assume that M only accepts when its head is on the left endmarker in state f. (Note that if M does not have the assumed properties, we can modify M to satisfy the desired properties. The time complexity will change from $c^{L(n)}$ to $d^{L(n)}$ for some $d \geqq c$.) Clearly, the computation of M will have the pattern shown in Fig. 8.

Let K, Σ, Γ be the state set, input alphabet, and storage tape alphabet of M, respectively, with Γ containing the blank symbol, B, and the left endmarker, ¢. Define a PPS $G = \langle P, N, T, R_1, R_2\rangle$ by: $P = ¢\Sigma^*$, $N = \{(f, ¢, 1)\} \cup \{(q, a, i, j) \mid q \text{ in } K, a \text{ in } \Gamma, i \text{ and } j$ positive integers$\}$ and $T = \{(q_0, ¢, 1, 1)\}$. Intuitively, (q, a, i, j) represents the situation in which M is in state q with its storage head scanning symbol a in position i (the left endmarker is in position 1), and M is on the jth sweep. Note that $|(q, a, i, j)| = O(\log (i+j))$. In Fig. 8, k = the number of sweeps made is even, and $k \leqq c_1^{L(n)}$ for some c_1. Also, the storage space used is $s \leqq c_2^{L(n)}$ for some c_2. Clearly, a 4-tuple $\alpha = (q, a, i, j)$ appearing in the pattern is uniquely defined by at most 2 other 4-tuples. Thus, in Fig. 8, α_1 is uniquely defined by α_2 and α_3. On the other hand, β_1 is uniquely defined by only β_2. We shall define the relations R_1 and R_2 in such a way that the computation of M can be simulated in reverse, i.e., the root of the solution tree will be $(f, ¢, 1)$ and the leaves will be $(q_0, ¢, 1, 1)$. For convenience, we define R_1 and R_2 by cases. Assume that the input is $x = ¢a_2 \cdots a_n$.

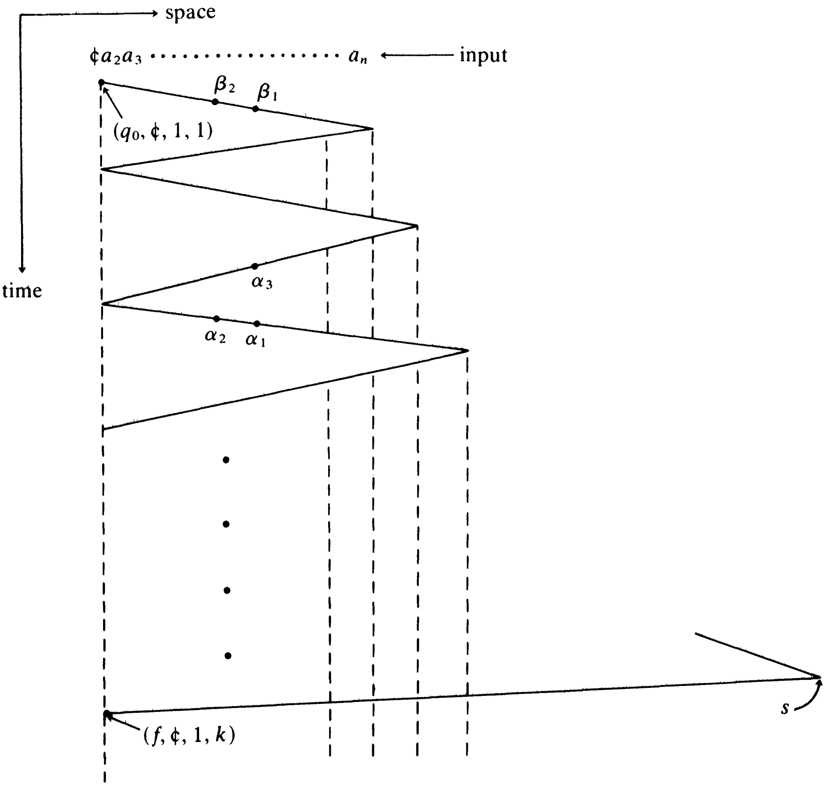
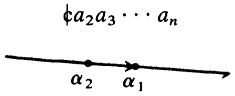


FIG. 8. *An accepting computation of M.*

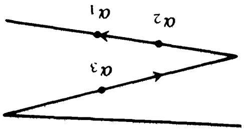
Case 0. Last sweep: For each positive integer k, let $(x, (f, ¢, 1), (f, ¢, 1, k))$ be in R_1.

Case 1. First sweep, i.e., $j = 1$:



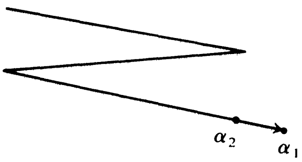
(x, α_1, α_2) is in R_1 if $\alpha_1 = (q_1, a_{i+1}, i+1, 1)$, $\alpha_2 = (q_2, a_i, i, 1)$, $1 \leqq i < n$, M in state q_2 reading a_i moves right in state q_1. $(a_1 = ¢.)$

Case 2. $2j+1$st sweep, $j \geqq 1$:



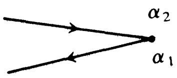
$(x, \alpha_1, \alpha_2, \alpha_3)$ is in R_2 if $\alpha_1 = (q_1, a, i+1, 2j+1)$, $\alpha_2 = (q_2, b, i, 2j+1)$, $\alpha_3 = (q_3, c, i+1, 2j)$, $a \neq B$, $b \neq B$, M in state q_3 reading c rewrites c by a and moves left, and M in state q_2 reading b moves right in state q_1.

Case 3. Entering the leftmost blank:



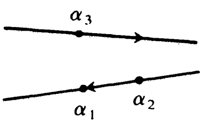
(x, α_1, α_2) is in R_1 if $\alpha_1 = (q_1, B, i+1, 2j-1)$, $\alpha_2 = (q_2, b, i, 2j-1)$, $j \geqq 1$, $b \neq B$, M in state q_2 reading b moves right in state q_1. If $j = 1$, then b is the last symbol of $x = ¢a_2 \cdots a_n$.

Case 4. Change in sweep number occuring on the right:



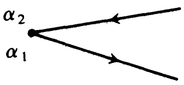
(x, α_1, α_2) is in R_1 if $\alpha_1 = (q_1, B, i, 2j)$ and $\alpha_2 = (q_1, B, i, 2j-1)$, $j \geqq 1$.

Case 5. $2j$th sweep, $j \geqq 1$:



$(x, \alpha_1, \alpha_2, \alpha_3)$ is in R_2 if $\alpha_1 = (q_1, a, i, 2j)$, $\alpha_2 = (q_2, b, i+1, 2j)$, $\alpha_3 = (q_3, c, i, 2j-1)$, $a \neq B$, $c \neq B$, M in state q_3 reading c rewrites c by a and moves right, M in state q_2 reading b moves left in state q_1.

Case 6. Change in sweep number occurring on the left:



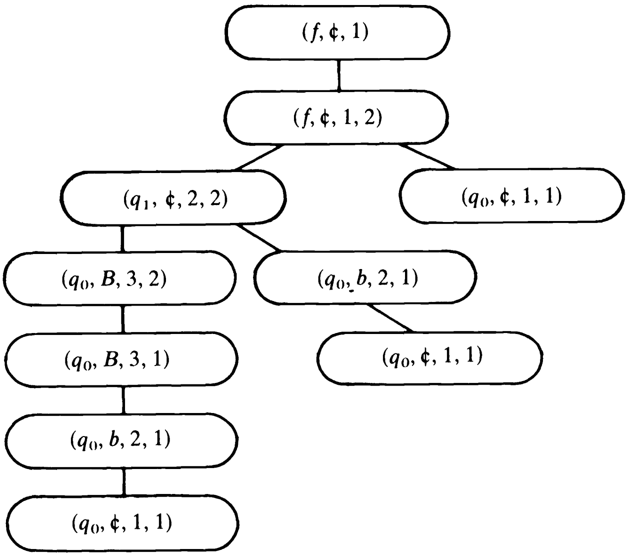
(x, α_1, α_2) is in R_1 if $\alpha_1 = (q_1, ¢, 1, 2j+1)$ and $\alpha_2 = (q_1, ¢, 1, 2j)$, $j \geqq 1$.

Now $x = ¢a_2 \cdots a_n$ is accepted by M in k sweeps and space s if and only if there is a solution tree for $(x, (f, ¢, 1))$ all of whose nonroot nodes are of the form (q, a, i, j) with $i \leqq s$ and $j \leqq k$. Thus, a solution tree for $(x, (f, ¢, 1))$ has maximum node size $O(\log(s+k)) = O(L(n))$. The result now follows from Lemma 3 and Corollary 3 (filial divide-and-conquer). □

Note that the "only if" part of the proof of Theorem 8 fails if M is nondeterminstic. This is because the PPS G may have a solution tree which does not correspond to any computation of M. For example, consider the NTM M which when in state q reading a may rewrite a by a' and enter state q' while moving in direction D, where the possible assignments for q, a, a', q' and D are given in Fig. 9(a). Clearly, $x = ¢b$ is not accepted by M. However, the PPS G has a solution tree for $(¢b, (f, ¢, 1))$. See Fig. 9(b).

q	a	q'	a'	D
q_0	¢	q_0	¢	right
q_0	b	q_0	b	right
q_0	b	q_1	c	right
q_0	B	q_1	c	left
q_1	c	f	c	left

(a)



(b)

FIG. 9

The next theorem involves $L(n)$-tape bounded NAPA's with reversal-bounded pushdown store.

THEOREM 9. *Let L be accepted by an $L(n)$-tape bounded NAPA which makes at most $R(n)$ pushdown reversals on inputs of length n. Then*

(i) *L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)[L(n)+\log(R(n))]$ space;*

(ii) *L can be accepted by an $L(n)$-tape bounded NAPA whose pushdown store uses no more than $L(n)\log(R(n))$ space.*

Proof. This follows from Lemma 1(ii), Corollary 2 (balanced divide-and-conquer) and Theorem 2 (filial divide-and-conquer). □

The bound in Theorem 9(i) can be tightened if the machine is deterministic:

THEOREM 10. *Let L be accepted by an $L(n)$-tape bounded DAPA which makes at most $R(n)$ pushdown reversals on inputs of length n. Then L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)\log(R(n))$ space.*

Proof. The proof is similar to that of Theorem 6 using Lemma 1(iii). □

The next corollary follows from Theorems 9–10 and the observation that a machine with multiple (input) heads can be simulated by a one-headed machine with a $\log n$-tape bounded read-write storage tape.

COROLLARY 9.

(i) *Let L be accepted by a nondeterministic two-way multihead pushdown automaton M [3, 4] which makes at most $R(n)$ pushdown reversals on inputs of length n. Then L can be accepted by a $(\log n)(\log n+\log(R(n)))$-tape bounded DTM M_1 and by a $(\log n)(\log(R(n)))$-tape bounded NTM M_2.*

(ii) *If in (i), M is deterministic, then the DTM M_1 can be constructed to be $(\log n)(\log(R(n)))$-tape bounded.*

Next, we consider an $L(n)$-tape bounded NAPA whose input head and storage head make at most $R(n)$ reversals. The pushdown store is unrestricted. For such a machine, the complexity of simulation is smaller than that of Theorem 9(i). We need the following lemma which is of independent interest.

LEMMA 5. *Let L be accepted by an $L(n)$-tape bounded NAPA M whose input head and storage head make at most $R(n)$ reversals on inputs of length n. Then L can be accepted by an $L(n)$-tape bounded NAPA which operates within $R(n)[n+L(n)]$ time.*

Proof. Assume without loss of generality that M moves one of its heads (input or storage) just before accepting. Also assume that if M does not move its storage head from a given square, the content of that square is not changed. Let Σ and Γ be the input and storage alphabets of M, respectively, $\Sigma \cap \Gamma = \varnothing$. (For notational convenience, assume that the input endmarkers, ¢ and \$, are in Σ and the blank symbol B is in Γ). Let $D=\{-1, 0, +1\}$ and $\Delta = \Sigma \times D \times \Gamma \times \Gamma \times D$. Define the language (over Δ) $L' = \{(a_1, i_1, b_1, c_1, j_1) \cdots (a_k, i_k, b_k, c_k, j_k) | k \geqq 1, (i_t, j_t) \neq (0,0)$ for $1 \leqq t \leqq k$, there are states $q_1, \cdots, q_{k+1}$ with q_1 the initial state and q_{k+1} an accepting state and pushdown strings $\alpha_1 = Z_0$ (the initial pushdown symbol), $\cdots, \alpha_{k+1}$ such that for $1 \leqq t \leqq k$, M in state q_t with pushdown contents α_t and its input and storage heads on symbols a_t and b_t, respectively, eventually moves its input head in direction i_t, rewrites b_t by c_t, moves the storage head in direction j_t, and enters state q_{t+1} with α_{t+1} the resulting pushdown contents$\}$. Clearly, L' can,be accepted by a one-way pushdown automaton M'. Since every one-way pushdown automaton can be transformed to be real-time [8], we may assume that M' is real-time. We will now construct from M' an $L(n)$-tape bounded NAPA M'' which will accept L within time $R(n)[n+L(n)]$.

M'', when given input x in $¢(\Sigma - \{¢, \$\})^*\$$, an initially blank storage tape and Z_0 on its pushdown store, simulates the computation of M' while checking that the input is consistent. This is accomplished as follows:

Assume that at some point in the simulation, the input and storage heads of M'' are on symbols a_t and b_t, respectively. Also assume that the current state, q_t, of M' is stored in the finite contol of M''. Then M'' carries out the following steps simultaneously: (i) M nondeterministically guesses a triple (i_t, c_t, j_t) such that $(i_t, j_t) \neq (0,0)$; (ii) moves the input head in direction i_t; (iii) rewrites the storage tape symbol b_t by c_t; (iv) moves the storage head in direction j_t; (v) determines the move of M' when in state q_t and input $(a_t, i_t, b_t, c_t, j_t)$; and (vi) updates the pushdown and state q_t accordingly.

M'' accepts the input if and only if M' enters an accepting state. Clearly, M'' accepts L. Now if a string x, $n = |x|$, is accepted by M (the original NAPA), then the number of nonstationary moves of M is $m \leqq R(n)[n+L(n)]$. By the definition of L', there is some string y in L' that codes the computation of M and $|y| = m$. The one-way pushdown automaton M' accepting L' is real-time. From the description of M'' the simulation of the computation of M' on string y can be done without loss of time. Hence, M'' on input x accepts in time m. It follows that M'' operates within time $R(n)[n+L(n)]$. □

From Lemma 5 and Theorems 6 and 7, we have

THEOREM 11. *Let L be accepted by an $L(n)$-tape bounded NAPA whose input head and storage head make at most $R(n)$ reversals on inputs of length n. Then*

(i) *L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)[\log(R(n)) + \log(n+L(n))]$ space;*

(ii) *L can be accepted by an $L(n)$-tape bounded DAPA whose pushdown store uses no more than $L(n)R(n)[n+L(n)]$ space.*

Suppose that in an $L(n)$-tape bounded NAPA, we replace the auxiliary pushdown store by an unrestricted read-write storage tape. Then we obtain a nondeterministic TM with a read-only input tape and two read-write storage tapes, one of which is $L(n)$-tape bounded. Call this new machine an *$L(n)$-tape bounded nondeterministic auxiliary storage* TM (or *$L(n)$-tape bounded* NATM, for short). Then we have the following result which generalizes Theorem 4.2 of Greibach [6].

THEOREM 12. *The following statements are equivalent.*

(i) *L is accepted by an $L(n)R(n)$-tape bounded NTM.*

(ii) *L is accepted by an $L(n)$-tape bounded NATM whose auxiliary storage makes at most $R(n)$ reversals on inputs of length n.*

Proof. The proof follows the lines in [6].

(i) implies (ii). Let L be accepted by a NTM M with a two-way read-only input and one $L(n)R(n)$-tape bounded storage tape. A partial configuration of M is a tuple $\alpha = (i, uqv)$ which represents the situation in which M (on some input) is in state q, its input head is on the ith position, and its storage tape contains uv with the storage head on the leftmost symbol of v. Clearly, $L(n)R(n)$-space is sufficient to store a partial configuration. We now describe the computation of an $L(n)$-tape bounded NATM M' accepting L whose auxiliary storage makes at most $R(n)$ reversals. Given an input ¢x\$, M' simulates the computation of M as follows:

M' starts off by guessing a sequence of partial configurations that M goes through on input ¢x\$. This sequence is written on the auxiliary storage with #'s separating the configurations (see Fig. 10). Each α_i is broken up into $R(n)$ segments, each of which requires $L(n)$ space. M' then checks that $\alpha_1\#\alpha_2\# \cdots \#\alpha_k$ is an accepting sequence of configurations of M on input ¢x\$. It does this by making $R(n)$ passes over $\alpha_1\# \cdots \#\alpha_k$.

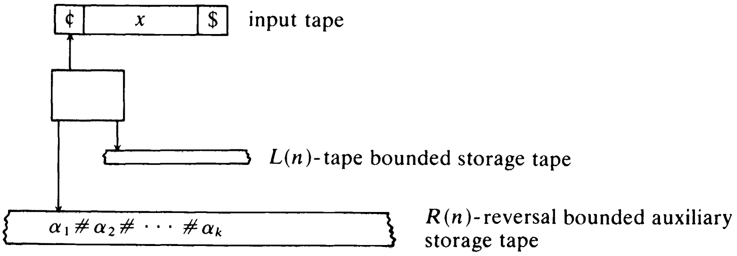

¢
x
\$
input tape
$L(n)$-tape bounded storage tape
$\alpha_1\#\alpha_2\# \cdots \#\alpha_k$
$R(n)$-reversal bounded auxiliary storage tape

FIG. 10. *NATM M'*.

On the jth pass, M' checks that the jth segments of the configurations are consistent. The $L(n)$-tape bounded storage tape is used as a scratch tape for this purpose. It is clear that M' can be constructed to be an $L(n)$-tape bounded NATM whose auxiliary storage is $R(n)$-reversal bounded.

(ii) implies (i). Suppose that L is accepted by an $L(n)$-tape bounded NATM M whose auxiliary storage is $R(n)$-reversal bounded. We may assume without loss of generality that M's auxiliary storage head makes no stationary moves. Moreover, we may assume that the auxiliary storage has left and right endmarkers and reversals can only occur on these endmarkers. (The number of squares between the endmarkers is fixed during a given computation.) Clearly, a computation of M on a given input can be described by a profile of partial configurations as shown in Fig. 11.

In Fig. 11, each α_j is of the form (Z, i, uqv). (Z, i, uqv) represents the situation in which M (on some input) is in state q, its input head is on position i, the $L(n)$-tape bounded storage tape contains uv with the storage head on the leftmost symbol of v, and the auxiliary storage is scanning symbol Z. Call the "crossing sequence" $\langle \alpha_1, \alpha_2, \cdots, \alpha_{2k-1}, \alpha_{2k} \rangle$ a *cut* in the profile. The $L(n)R(n)$-tape bounded NTM M' accepting L operates by constructing (in succession) the cuts appearing in the profile of an accepting computation of M. For example, in Fig. 11, $\langle \alpha'_1, \alpha'_2, \cdots, \alpha'_{2k-1}, \alpha'_{2k} \rangle$ can easily be constructed by M knowing $\langle \alpha_1, \alpha_2, \cdots, \alpha_{2k-1}, \alpha_{2k} \rangle$. Since a cut contains at most $R(n)$ partial configurations, each of which requires at most $L(n)$ space, $L(n)R(n)$ is sufficient to record a cut. We omit the details. □

5.2. Parallel Turing machines. We now turn to some applications of PPS's to parallel Turing machines. The model of a parallel TM was introduced in [11] as a

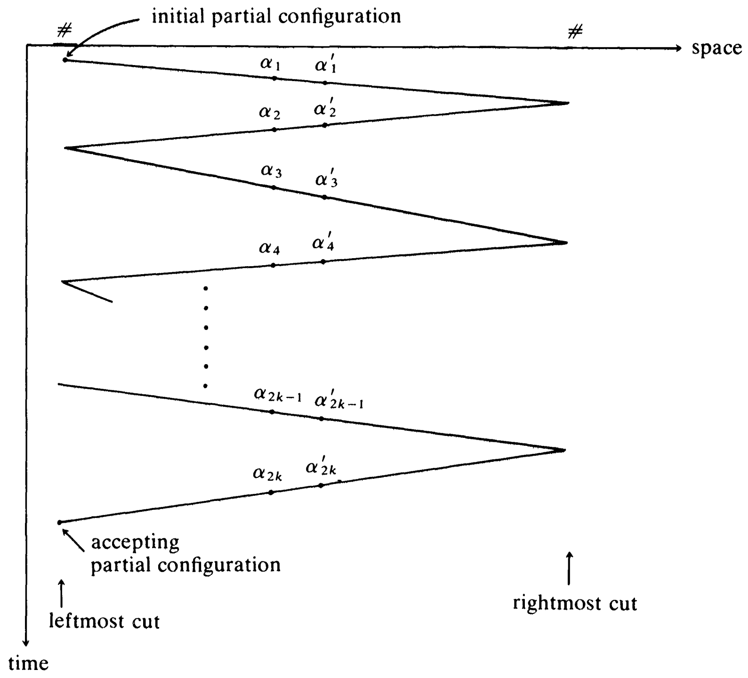


FIG. 11. *Profile of a computation of M.*

generalization of nondeterministic TM (see § 4 for definition). Several results concerning the relationships between time- and tape-bounded deterministic and parallel computation were derived in [11]. We observe that most of these results can easily be shown using PPS. To illustrate, we give a proof of the following result which combines Theorems 3 and 4 of [11].

THEOREM 13.

(i) *If L is accepted by an $L(n)$-tape bounded parallel TM, then L can be accepted by a $c^{L(n)}$-time bounded DTM for some constant c.*

(ii) *If L is accepted by a $c^{L(n)}$-time bounded DTM for some constant c, then L can be accepted by an $L(n)$-tape bounded parallel TM.*

Proof. (i) Let L be accepted by an $L(n)$-tape bounded parallel TM M. Without loss of generality, we may assume that in every step, M has at most 2 choices of next moves. Construct a PPS $G=\langle P, N, T, R_1, R_2\rangle$, where $P=\text{¢}\Sigma^*\$$, $N=\{(\alpha, B_\alpha)|\alpha$ is a partial configuration of M, $B_\alpha=0, 1\}$, $T=\{(\alpha, 1)|\alpha$ is an accepting configuration$\}\cup\{(\alpha, 0)|\alpha$ is a rejecting configuration$\}$, $R_1=\{(x, (\alpha_1, B_{\alpha_1}), (\alpha_2, B_{\alpha_2}))|x$ in $\text{¢}\Sigma^*\$$, (α_i, B_{α_i}) in N, $B_{\alpha_1}=B_{\alpha_2}$, M in configuration α_1 on input x can (in one step) only enter configuration $\alpha_2\}$, and $R_2=\{(x, (\alpha_1, B_{\alpha_1}), (\alpha_2, B_{\alpha_2}), (\alpha_3, B_{\alpha_3}))|x$ in $\text{¢}\Sigma^*\$$, (α_i, B_{α_i}) in N, $\alpha_2\neq\alpha_3$, M in configuration α_1 can (in one step) enter configuration α_2 or α_3, $B_{\alpha_1}=B_{\alpha_2}\wedge B_{\alpha_3}$ if $h(\text{state}(\alpha_1))=\wedge$ and $B_{\alpha_1}=B_{\alpha_2}\vee B_{\alpha_3}$ if $h(\text{state}(\alpha_1))=\vee\}$. It is clear that x is accepted by M if and only if there is a solution tree for $(x, (\alpha_0, 1))$, where $\alpha_0=(1, q_0B)$ is the initial partial configuration of M. Part (i) then follows from Theorem 5 (dynamic programming).

(ii) Now suppose L is accepted by a $c^{L(n)}$-time bounded DTM M. Assume that M has the properties stated in the proof of Theorem 8. Construct the PPS $G=\langle P, N, T, R_1, R_2\rangle$ corresponding to M as described in the proof of Theorem 8. G has the property that a string $x = ¢a_2 \cdots a_n$ (of length n) is accepted by M within time $c^{L(n)}$ if and only if $(x, (f, ¢, 1))$ has a solution tree with WIDTH $O(L(n))$. The result now follows from Theorem 4 (filial divide-and-conquer). □

COROLLARY 10. *A language L is accepted by an $L(n)$-tape bounded DAPA if and only if it is accepted by an $L(n)$-tape bounded parallel TM.*

Proof. This follows from Theorems 8 and 13. □

5.3. Alternating Turing machines. The definition of acceptance by a parallel TM requires that each path in the computation tree leads to an accepting or rejecting configuration. This requirement can lead to some inconsistency with NTM conventions. Consider, e.g., an ordinary NTM M in which the initial partial configuration $\alpha_0 = (1, q_0B)$ on any input x in $¢\Sigma^*\$$ has exactly two choices of next configurations: α_0 and $\alpha_1 = (1, q_1B)$, where q_1 is an accepting state. Then M considered as a nondeterministic TM accepts all strings. However, M considered as a parallel TM with h defined by: $h(q_0) = h(q_1) = \vee$ accepts the empty set. The inconsistency can be avoided by using the following more natural definition of acceptance:

Let M be a parallel TM and x be in $¢\Sigma^*\$$. Define a *computation tree* for x as a tree whose nodes are partial configurations satisfying :(i) the root node is α_0; (ii) the leaves are accepting configurations; (iii) if α is a node which is not a leaf and M in configuration α on input x can, in one step, enter configurations $\beta_1, \cdots, \beta_k$, then α has sons $\beta_1, \cdots, \beta_k$ provided $h(\text{state}(\alpha)) = \wedge$; if $h(\text{state}(\alpha)) = \vee$, then α has exactly one son, β_i, $1 \leqq i \leqq k$. A string x is accepted by M if x has a computation tree. A parallel TM which accepts this way is called an *alternating* TM in [2], [12]. Again, we assume without loss of generality that in every step, M has at most 2 choices of next moves.

The class of languages accepted by $L(n)$-tape bounded alternating TM's remains the same since Theorem 13 also holds for alternating TM's [2]. The proof of part (ii) clearly holds. For part (i), the appropriate *PPS* is $G = \langle P, N, T, R_1, R_2 \rangle$, where $P = ¢\Sigma^*\$$, $N = \{\alpha | \alpha$ is a partial configuration of $M\}$, $T = \{\alpha | \alpha$ is an accepting partial configuration$\}$, $R_1 = \{(x, \alpha_1, \alpha_2) | x$ in $¢\Sigma^*\$$, α_i in N, M in configuration α_1 on input x can (in one step) enter configuration α_2, and either α_2 is unique or $h(\text{state}(\alpha_1)) = \vee\}$, and $R_2 = \{(x, \alpha_1, \alpha_2, \alpha_3) | x$ in $¢\Sigma^*\$$, α_i in N, $\alpha_2 \neq \alpha_3$, M in configuration α_1 on input x can (in one step) enter configuration α_2 or α_3, and $h(\text{state}(\alpha_1)) = \wedge\}$. Note that a solution tree $F^x_{\alpha_0}$, where α_0 is the initial partial configuration, corresponds exactly to a computation tree for x.

An alternating TM M is $C(n)$-*conjunction bounded* ($D(n)$-*disjunction bounded*) if every input of length n that is accepted has a computation tree in which the number of $\wedge$-nodes ($\vee$-nodes) is at most $C(n)(D(n))$. A node α is a $\wedge$-node ($\vee$-node) if $h(\textit{state}(\alpha)) = \wedge\ (\vee)$. Clearly, if M is $L(n)$-tape bounded and $C(n)$-conjunction bounded, then any input x of length n that is accepted has a computation tree that has at most $C(n)+1$ leaves. It follows that the PPS G corresponding to M as defined above has a solution tree $F^x_{\alpha_0}$ for (x, α_0) with at most $C(n)+1$ leaves. If, in addition, M is $T(n)$-time bounded, then the computation tree for x (and hence the solution tree $F^x_{\alpha_0}$) has depth at most $T(n)$.

Parts (i) and (ii) of the next result then follow from Corollary 2 (balanced divide-and-conquer) and Theorem 2 (filial divide-and-conquer) while part (iii) follows from Corollary 1 (balanced divide-and-conquer).

THEOREM 14. *Let L be accepted by an $L(n)$-tape bounded and $C(n)$-conjunction bounded alternating TM M. Then*

(i) *L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)[L(n) + \log (C(n))]$ space;*

(ii) *L can be accepted by an $L(n)$-tape bounded NAPA whose pushdown store uses no more than $L(n) \log (C(n))$ space.*

(iii) *If, in addition, M is $T(n)$-time bounded, then L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)[\log (T(n)) + \log (C(n))]$ space.*

In case the alternating *TM M is* $L(n)$-tape bounded, $C(n)$-conjunction bounded and $D(n)$-disjunction bounded, then the solution tree $F^x_{\alpha_0}$ has at most $2C(n)+D(n)+1$ nodes. The next result then follows from Theorem 1 (balanced divide-and-conquer).

THEOREM 15. *Let L be accepted by an $L(n)$-tape bounded, $C(n)$-conjunction bounded and $D(n)$-disjunction bounded alternating TM. Then L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n) \log (C(n)+D(n))$ space.*

COROLLARY 11. *Let L be accepted by an $L(n)$-tape bounded NTM which makes at most $D(n)$ nondeterministic moves on inputs of length n. Then L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n) \log (D(n))$ space.*

A slightly weaker form of Corollary 11 has been observed earlier by Kintala and Fischer [10] without proof.

If the alternating TM M is $L(n)$-tape bounded and $T(n)$-time bounded, then the PPS G corresponding to M (as described above) has a solution tree $F^x_{\alpha_0}$ with CHOICE $(F^x_{\alpha_0}) \leqq 2$ and DEPTH$(F^x_{\alpha_0}) \leqq T(n)$. Then the functions f and g of Theorem 3 (filial divide-and-conquer) can be defined as follows. If $S(x, (k, uvw), l, i, j) = (k', uaqbw)$ for some symbols a and b, then $f(x, (k, uvw), l, i, j) = (k'-k, v)$ and $g(x, (k', uaqbw), l, i, j, (k'-k, v)) = (k, uvw)$. Now abs$(k'-k) \leqq 1$ and $|v| \leqq$ fixed constant. Hence, whenever $f(\cdot)$ is defined, $|f(\cdot)| \leqq$ fixed constant. Thus, we have

THEOREM 16. *Let L be accepted by an $L(n)$-tape bounded and $T(n)$-time bounded alternating TM. Then L can be accepted by an $L(n)$-tape bounded DAPA whose pushdown store uses no more than $T(n)$ space.*

5.4. Recursive Turing machines. Finally, we look at applications of PPS's to recursive Turing machines [16]. A recursive TM is an ordinary TM which can call itself. We refer the reader to [16] for formal definitions and motivations. Here, we just give a brief description.

A recursive TM, M, has a finite control attached to a two-way write-only *input tape*, a two-way read-only *input-parameter tape*, a one-way write-only *output-parameter tape*, and a two-way read-write *storage tape* (see Fig. 12). The finite control has six distinguished states: ⟨start⟩, ⟨call⟩, ⟨return-yes⟩, ⟨return-no⟩, ⟨received-yes⟩, ⟨received-no⟩. Initially, there is only one copy of M at level 1. The input string and input-parameter string (assumed null at level 1) are placed on their respective tapes delimited by endmarkers ¢ and \$. The heads on these tapes are initially positioned on the left endmarker, ¢. The output-parameter tape contains ¢ and the storage tape is completely blank. The machine begins in state ⟨start⟩ and operates like an ordinary nonrecursive TM with a one-way write-only output tape. (Thus, the moves of M are determined by the state and the symbols scanned on the input, input-parameter and storage tapes.)

The "active" machine, at level k (initially, $k = 1$), can make a recursive call by writing the right delimiter \$ on its output-parameter tape and entering state ⟨call⟩. When this happens, a new copy of M, at level $k+1$, is created with the output-parameter tape of level k becoming the input-parameter tape of the new machine. A new input tape containing the original input string, a new output-parameter tape containing ¢ and a new blank read-write storage tape are created for the machine at level $k+1$. The new copy at level $k+1$ becomes the active machine and begins its computation in state ⟨start⟩ with the input, input-parameter, and output-parameter heads on ¢. The machine at level k is now "waiting". If the machine at level $k+1$ enters the state ⟨return-yes⟩ or ⟨return-no⟩, then the machine at this level disappears and the waiting machine at level k resumes its computation in state ⟨received-yes⟩ or ⟨received-no⟩, respectively, with its output-parameter tape reinitialized to ¢. We assume without loss of generality that the heads on the storage and output-parameter tapes do not write blanks.

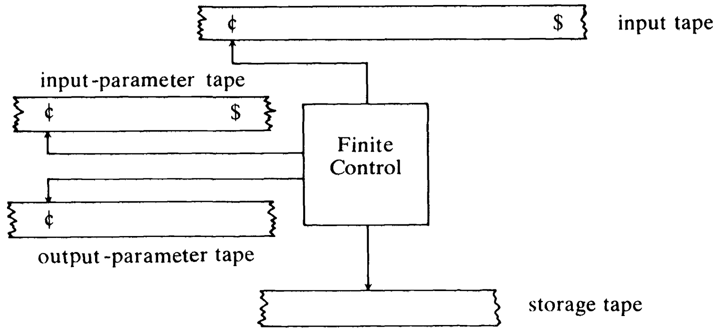


FIG. 12. *A recursive TM, M.*

An input tape x in $¢\Sigma^*\$$ is *accepted* or *rejected* if M eventually enters the state ⟨return-yes⟩ or ⟨return-no⟩, respectively, at level 1. (Note that M on some inputs may never enter ⟨return-yes⟩ or ⟨return-no⟩. These inputs are neither accepted nor rejected. Thus M as defined is a recognition device, see [16].) M can be *deterministic* or *nondeterministic.* In case M is nondeterministic we require that there is no input for which there are two computations which accept and reject the input, respectively. In fact, we shall assume that there is no combination of input and parameter which can result in both a "yes" and a "no" answer to a recursive call (see [16]).

Following [16], define a *level instantaneous description* (or simply *level* ID) of M as a 4-tuple $\alpha = (i, uqv, w\!\uparrow\! y, z)$. α represents the configuration of M (at some level) in which the input head is on position i, the storage tape contains uv with the storage head on the leftmost symbol of v, the state is q, the input-parameter tape contains wy with its head on the left-most symbol of y, and the output parameter tape contains z. The *width* of α is the length of uv plus the length of z.

DEFINITION. Let M be a recursive TM. M is *$L(n)$-width bounded* if every input of length n has an accepting or rejecting computation in which each level ID encountered during the computation has width at most $L(n)$. Similarly, M is *$D(n)$-depth bounded* (*respectively, $C(n)$-call bounded*) if every input of length n has an accepting or rejecting computation in which the depth of recursion (respectively, the number of recursive calls) is at most $D(n)$ (respectively, $C(n)$).

The following result was shown in [16].

THEOREM 17. *Let L be accepted by a deterministic (respectively, nondeterministic) $L(n)$-width bounded and $D(n)$-depth bounded recursive TM M. Then L can be accepted by an $L(n)D(n)$-tape bounded DTM (respectively, NTM) M'.*

Proof [16]. The construction of M' from M is straightforward. M' simply keeps track of the level ID's as they are created and updates them accordingly. Since M has simultaneous width $L(n)$ and depth $D(n)$, a storage space of $O(L(n)D(n))$ is sufficient. □

Now consider the case when the recursive TM M is $L(n)$-width bounded and $C(n)$-call bounded. Clearly, the above construction applied to this case yields an M' which still has a worst case storage bound of $O(L(n)C(n))$. (This will happen, e.g., when $D(n)=C(n)$.) We can prove a better simulation result.

THEOREM 18. *Let L be accepted by a deterministic $L(n)$-width bounded and $C(n)$-call bounded recursive TM M. Then L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)\log(C(n))$ space and by an $L(n)$-tape bounded DASA whose stack uses no more than $C(n)$ space.*

Proof. Let α and β be level ID's of M and x in $¢\Sigma^*\$$ be an input. Write $\alpha \overset{*}{\underset{x}{\Rightarrow}} \beta$ if α can enter β after 0 or more steps and state$(\beta) = \langle$return-yes$\rangle$ or $\langle$return-no$\rangle$ or $\langle$call$\rangle$. The initial level ID (at level 1) is $\alpha_0 = (1, \langle\text{start}\rangle B, \upharpoonright ¢\$, ¢)$, where B is the blank symbol. Without loss of generality, assume that M in level ID α_0 on its first move enters level ID $\hat{\alpha}_0 = (1, \langle\text{call}\rangle\hat{B}, \upharpoonright ¢\$, ¢\$)$, where $\hat{B}$ is a fixed nonblank symbol. Thus, $\alpha_0 \overset{*}{\underset{x}{\Rightarrow}} \hat{\alpha}_0$ for all input x.

Define a PPS $G = \langle P, N, T, R_1, R_2\rangle$ as follows: $N = \{(h, \alpha, \beta) | \alpha$ and β are level ID's and $h =$ yes or no$\}$, $P = ¢\Sigma^*\$$, $T = \{(h, \alpha, \beta) | (h, \alpha, \beta)$ in N, state$(\beta) = \langle$return-$h\rangle\}$, $R_1 = \emptyset$, $R_2 = \{(x, (h_1, \alpha_1, \beta_1), (h_2, \alpha_2, \beta_2), (h_3, \alpha_3, \beta_3)) | x$ in $¢\Sigma^*\$$, (h_j, α_j, β_j) in N, $h_1 = h_3$, $\alpha_j \overset{*}{\underset{x}{\Rightarrow}} \beta_j$, $\beta_1 = (i, u\langle\text{call}\rangle v, w\upharpoonright y, z\$)$, $\alpha_2 = (1, \langle\text{start}\rangle B, \upharpoonright z\$, ¢)$, $\alpha_3 = (i, u\langle\text{received-}h_2\rangle v, w\upharpoonright y, ¢)\}$.

Clearly, M accepts x in at most $C(n)$ recursive calls ($n = |x|$) if and only if $(x, (\text{yes}, \alpha_0, \hat{\alpha}_0))$ has a solution tree with at most $2C(n)+1$ nodes (hence, depth at most $C(n)+1$). Also, from the definition of R_2 and the fact that h_2 has only two possible values (yes and no), CHOICE$(F^x_\alpha) \leqq 2$ for any solution tree F^x_α. The result then follows from Theorem 1 (balanced divide-and-conquer) and Theorem 3 (filial divide-and-conquer). □

For nondeterministic recursive TM's we have

THEOREM 19. *Let L be accepted by a nondeterministic $L(n)$-width bounded and $C(n)$-call bounded recursive TM M. Then*

(i) *L can be accepted by an $L(n)$-tape bounded DASA whose stack uses no more than $L(n)[L(n)+\log(C(n))]$ space;*

(ii) *L can be accepted by an $L(n)$-tape bounded NAPA whose pushdown store uses no more than $L(n)\log(C(n))$ space.*

Proof. The proof is similar to that of Theorem 18. This time, the PPS $G = \langle P, N, T, R_1, R_2\rangle$ is defined as follows: $N = \{(h, \alpha) | \alpha$ is a level ID of M and $h =$ yes or no$\}$, $P = ¢\Sigma^*\$$, $T = \{(h, \alpha) | (h, \alpha)$ in N, state$(\alpha) = \langle$return-$h\rangle\}$, $R_1 = \{(x, (h_1, \alpha_1), (h_2, \alpha_2)) | h_1 = h_2$, state$(\alpha_1) \neq \langle$call$\rangle$, M in level ID α_1 on input x can, in one step, enter level ID $\alpha_2\}$, $R_2 = \{(x, (h_1, \alpha_1), (h_2, \alpha_2), (h_3, \alpha_3)) |\ h_1 = h_3$, $\alpha_1 = (i, u\langle\text{call}\rangle v, w\upharpoonright y, z\$)$, $\alpha_2 = (1, \langle\text{start}\rangle B, \upharpoonright z\$, ¢)$, $\alpha_3 = (i, u\langle\text{received-}h_2\rangle v, w\upharpoonright y, ¢)\}$. Then M accepts x in at most $C(n)$ recursive calls if and only if $(x, (\text{yes}, \alpha_0))$ has a solution tree with at most $C(n)+1$ leaves. From Corollary 2 (balanced divide-and-conquer) and Theorem 2 (filial divide-and-conquer) we have the result. □

Combining Theorems 17, 18, and 19 we get

THEOREM 20. *Let L be accepted by an $L(n)$-width bounded, $C(n)$-call bounded and $D(n)$-depth bounded recursive TM M.*

(i) *If M is deterministic, then L has deterministic tape complexity* $\min\{L(n)+C(n), L(n)\log(C(n)), L(n)D(n)\}$.

(ii) *If M is nondeterministic, then L has deterministic tape complexity* $L(n)[L(n)+\log(C(n))]$ *(respectively, nondeterministic tape complexity* $\min\{L(n)\log(C(n)), L(n)D(n)\}$).

We conclude with the following corollary which was also shown in [16] by a different technique.

COROLLARY 12. *Let L be accepted by a nondeterministic $L(n)$-width bounded and $D(n)$-depth bounded recursive TM. Then L has deterministic tape complexity $L^2(n)D(n)$.*

Proof. Immediate from Theorem 19(i) and the observation that $C(n) \leqq c^{L(n)D(n)}$ for some constant c. □

Acknowledgment. We would like to thank the referee for organizational suggestions and detailed comments which improved the presentation of our results.

REFERENCES

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

[2] A. CHANDRA AND L. STOCKMEYER, *Alternation*, Proc. of the Seventeenth Annual Symposium on Foundations of Computer Science, IEEE, (New York), 1976, pp. 98–108.

[3] S. COOK, *Characterizations of pushdown machines in terms of time-bounded computers*, J. Assoc. Comput. Mach., 18 (1971), pp. 4–18.

[4] ———, *Path systems and language recognition*, Proc. of the Second Annual ACM Symposium on Theory of Computing (New York), 1970, pp. 70–72.

[5] ———, *An observation on time-storage trade off*, Proc. of the Fifth Annual ACM Symposium on Theory of Computing (New York), 1973, pp. 29–33.

[6] S. GREIBACH, *Visits, crosses, and reversals for nondeterministic off-line machines*, Information and Control, 36 (1978), pp. 174–216.

[7] T. HARJU, *A simulation result for auxiliary pushdown automata*, Tech. Rep. 4 (1977), Dept. of Mathematics, Univ. of Turku, Finland.

[8] J. HOPCROFT AND J. ULLMAN, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[9] O. IBARRA, *Characterizations of some tape and time complexity classes of Turing machines in terms of multihead and auxiliary stack automata*, J. Comput. System. Sci., 2 (1971), pp. 88–117.

[10] C. KINTALA AND P. FISCHER, *Computations with a restricted number of nondeterministic moves*, Proc. of the Ninth Annual ACM Symposium on Theory of Computing (New York), 1977, pp. 178–185.

[11] D. KOZEN, *On parallelism in Turing machines*, Proc. of the Seventeenth Annual Symposium on Foundations of Computer Science, IEEE (New York), 1976, pp. 89–97.

[12] R. LADNER, R. LIPTON AND L. STOCKMEYER, *Alternating pushdown automata*, Proc. of the Nineteenth Annual Symposium on Foundations of Computer Science, IEEE (New York), 1978, pp. 92–106.

[13] P. LEWIS, R. STEARNS AND J. HARTMANIS, *Memory bounds for recognition of context-free and context-sensitive languages*, Proc. of Sixth Annual Symposium on Switching Circuit Theory and Logical Design, IEEE (New York), 1965, pp. 191–202.

[14] B. MONIEN, *Relationships between pushdown automata and tape-bounded Turing machines*, Proc. of Symposium on Automata, Languages and Programming, 78-Rocquencourt, France, M. Nivat, ed., North Holland Elsevier, Amsterdam, July 1972, pp. 575–583.

[15] W. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[16] ———, *Recursive Turing machines*, International J. Comput. Math., 6 (1977), pp. 3–31.

[17] I. SUDBOROUGH, *Time and tape bounded auxiliary pushdown automata*, Proc. of the Sixth International Symposium on the Mathematical Foundations of Computer Science, Czechoslovokia, Springer-Verlag, September 1977.

[18] ———, *Relating open problems on the tape complexity of context-free languages and path systems problems*, Proc. of the Twelfth Annual Johns Hopkins Conference on Information Sciences and Systems, March 1978.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

CODE MOTION*

JOHN H. REIF†

Abstract. Code motion is a program optimization concerned with the movement of code as far as possible out of control cycles into new locations where the code may be executed less frequently. This paper describes methods for approximating certain functions which ensure that the relocated code may be computed properly and safely, inducing no errors of computation.

The effectiveness of code motion depends on the goodness of our approximation to these functions, as well as on tradeoffs between (1) the *primary goal* of moving code out of control cycles and (2) the *secondary goal* of providing that the values resulting from the execution of relocated code are utilized.

Two versions of code motion are formulated: the first emphasizes the primary goal, whereas the other insures that the second goal is not compromised. Algorithms are presented for both formulations of code motion; the algorithm for the first version of code motion is restricted to reducible flow graphs, but the other runs on *all* flow graphs. Both of our algorithms run in almost linear time. Previous algorithms for similar formulations of code motion have time cost lower bounded in the worst case by the length of the program text times the number of nodes of the control flow graph.

Key words. code optimization, flow graph, data flow analysis, code movement

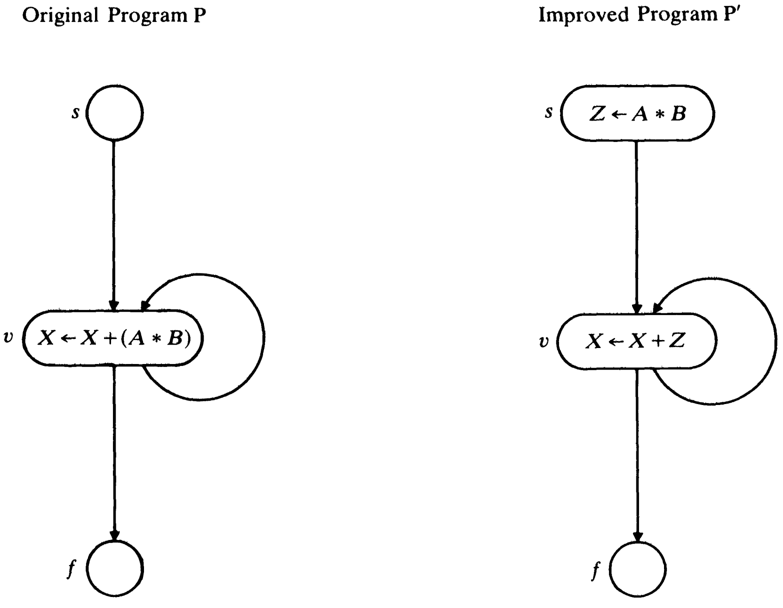
1. Introduction. We assume here the global flow model such as described in [2], [3]. Let $G = (V, E, s)$ be the *control flow graph* of program P to which we wish to apply code motion. Nodes in the set V correspond to linear blocks of code, edges in E specify possible control flow immediately between these blocks, and all flow of control begins at the start node $s \in V$. We also distinguish the *final node* $f \in V$ at which all flow of control ends. Every execution of the program P corresponds to a path in G, though some control paths may not correspond to possible executions of P. The essential parameters of the model are $n = |V|$, $m = |E|$, and $l =$ the number of text expressions[1] (each block in V is assumed to contain at least one text expression, so $n \leqq l$). We assume bit vectors of length l may be stored in a constant number of words and we have the usual logical and arithmetic operations on bit vectors, as well as an operation which shifts a bit vector to the left up to the first nonzero element. (This operation is generally used for normalization of floating point numbers; here it allows us to determine the position of the first nonzero element of the bit vector in a constant number of steps.) An algorithm runs in *almost linear* number of steps relative to this model if it requires $O((m+l)\alpha(m+l))$ bit vector and elementary operations, where α is the extremely slow-growing function of [22] (α is related to a functional inverse of Ackermann's function).

Consider a text expression t located at node loc(t) in V. To effect code motion (see also [5], [3], [6], [7] for descriptions of code motion optimizations) on the computation associated with t, we relocate the computation to a node movept(t) by deleting t from the text of node loc(t) (t may be replaced by a simpler instruction, say a load instruction) and installing an appropriate text expression t' (not necessarily lexically identical to the string t) at movept(t). On execution of the modified program P' the result of the computation at movept(t) might be stored in a special register or memory location to be retrieved when the execution reaches node loc(t). See Fig. 1 for a simple example of code motion.

* Received by the editors July 22, 1977, and in revised form May 9, 1979.

† Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts. Now at Computer Science Department, University of Rochester, Rochester, New York 14627. This research was supported in part by the Naval Electronics System Command Contract N00039-76-C-0168 and Rome Air Development Center Contract F30602-76-C-0032.

[1] A text expression t may be considered an index into the text of a block of code in V.



Original Program P

Improved Program P′

FIG. 1. *A simple example of code motion.*

To insure that P' is semantically equivalent to the original program P, we require that if node w is the movept of t, then:

R1. All control paths from the start node s to loc(t) contain node w.

R2. The computation is possible at node w; i.e., all quantities required for the computation must be properly defined at node w.

R3. The computation must be *safe* at w; thus if an error occurs in a particular excecution of P', an error must also have occurred in the corresponding execution of the original program P.

Observe that the nodes satisfying R1 form a chain, called a *dominator chain*, from s to loc(t). The *birth point* of text expression t is the first node on this chain that satisfies R2. We show in [17] that if the program P is interpreted within the arithmetic domain, the problem of computing birth points exactly is recursively unsolvable, so we must be content with computable approximations. Reif [17], and Reif and Tarjan [19] give algorithms for computing such approximations. The algorithm of [19] requires an almost linear number of bit vector operations for all control flow graphs to compute an approximation BIRTHPT(t) to the true birth point. We assume BIRTHPT(t) is contained on all paths from the start node s to loc(t).

The first node on the dominator chain from BIRTHPT(t) to loc(t) which satisfies restriction R3 is called the *safe point* of t. Section 3 discusses an approximation to the safe point, called SAFEPT, which may be computed in an almost linear number of bit vector operations, given an efficient test for safety of code motion (we rely on a global flow algorithm by Tarjan [24] for this). Unfortunately, known algorithms (including Tarjan's) for testing safety of code motion are efficient only on a restricted class of flow graphs which are called *reducible* (see [9]).

Let us continue the formulation of the code motion problem. We add a further restriction:

R4. The movept of t must not be contained on a control cycle avoiding loc(t).

Let M_1 consist of all nodes occurring on the dominator chain from SAFEPT(t) to loc(t) that satisfy R4. We choose movept(t) from the nodes in M_1 based on the following goals:

G1. movept(t) is to be located on as few control cycles as possible.

G2. As few control paths as possible may contain movept(t) and reach the final node f in V without passing through loc(t) (we assume f is reachable from all nodes).

The above goals conflict, for to satisfy G1 we would choose movept(t) earlier in the dominator ordering than we would if we were to also satisfy goal G2.

We consider two formulations of code motion. In the first formulation we stress G1 and in the other we stress G2. Let M_2 be the set of nodes in M_1 which also satisfy the restriction:

R5. All control paths from the movept of t to the final node f must contain loc(t).

Note that M_2 is not empty since loc$(t) \in M_2$. For $i = 1, 2$ let

M_i' = those nodes in M_i which satisfy R4 and are contained in the minimum number of control cycles

and let movept$_i(t)$ be the latest node in M_i' relative to the dominator ordering of G

More general formulations of code motion have been described in [3], [4], [5], [6], [7], [16], including the movement of code to several nodes (rather than to a single node), the movement of code to nodes occurring after (rather than before) loc(t) in the dominator ordering of F, and code movement combined with common subexpression elimination. Previous formulations of code motion [6], [7] similar to ours require $\Omega(l)$ (the "big omega" notation denotes a lower bound in the worst case; see [14]) operations per node in the flow graph, or a total worst-case time cost of $\Omega(l \cdot n)$.

The next section defines the relevant digraph terminology. Section 3 presents an algorithm for computing SAFEPT, using Tarjan's algorithm for testing safety of code movement. Section 4 reduces the first version of code motion to the computation of SAFEPT and a pair of functions $C1$ and $C2$ related to the cycle structure of flow graphs. We show that the function $C1$ suffices to solve the second type of code motion; in this formulation we avoid testing for safety of code motion. Sections 5 and 6 present algorithms for computing the functions $C1$ and $C2$ over certain domains in an almost linear number of bit vector operations. The algorithm for computing $C2$ requires a special function DDP; in § 7 an algorithm, restricted to reducible flow graphs, is presented which computes DDP in $O(m\alpha(m))$ bit vector steps. We conclude in § 8 with a graph transformation (similar to those described in ([6], [2]) which improves the results obtained from the two versions of code motion and in certain cases simplifies our algorithms for computing $C1$ and $C2$.

2. Graph theoretic notions. A *digraph* $G = (V, E)$ consists of a set V of elements called *nodes* and a set E of ordered pairs of nodes called *edges*. The edge (u, v) *departs from* u and *enters* v. We say u is an *immediate predecessor* of v and v is an *immediate successor* of u. The *outdegree* of a node v is the number of immediate successors of v and the *indegree* is the number of immediate predecessors of v.

A *path from* u *to* w in G is a sequence of nodes $p = (u = v_1, v_2, \cdots, v_k = w)$ where $(v_i, v_{i+1}) \in E$ for all i, $1 \leqq i < k$.

The path p may be built by composing subpaths:

$$p = (v_1, \cdots, v_i) \cdot (v_i, \cdots, v_k).$$

The path p is a *cycle* if $u = w$. A path is *simple* if it contains no cycles.

A node u is *reachable* from a node v if either $u = v$ or there is a path from v to u.

A *flow graph* (V, E, s) is a triple such that (V, E) is a digraph and s is a distinguished node in V, the *root*, such that s has no predecessors and every node in V is reachable from s.

A digraph is *acyclic* if it contains no cycles. If u is reachable from v, u is a *decendant* of v and v is a *ancestor* of u (these relations are *proper* if $u \neq v$). Immediate successors are called siblings. An acyclic flow graph T is a *tree* if every node v other than the root has a unique immediate predecessor, the parent of v. T is *oriented* if the edges departing from each node are oriented from left to right.

The *preordering* of oriented tree T is defined by the following algorithm (see also Knuth [13]).

```
ALGORITHM A.
INPUT An oriented tree $T$ with root $s$.
OUTPUT A numbering of the nodes of $T$.
begin
  procedure PREORDER($w$):
    begin
      if $w$ is unnumbered then
        begin
          $k := k+1$;
          Let $w$ be numbered $k$;
          for all siblings of $w$ from left to right do
            PREORDER($u$);
        end;
    end;
  Initially all nodes are unnumbered;
  $k := 0$;
  PREORDER($s$);
end.
```

Given a preordering, we can (see [20]) test in constant time if any particular pair of nodes is in the ancestor relation. A *postordering* is the reverse of a preordering.

Let $G = (V, E, s)$ be an arbitrary flow graph. A *spanning tree* of G is an oriented tree ST rooted at s with node set V and edge list contained in E. The edges contained in ST are called *tree edges*, edges in E from descendents to ancestors in ST are called *cycle edges*, nontree edges in E from ancestors to their descendants in ST are *forward edges*, and edges in E between nodes unrelated in ST are *cross edges*.

A special spanning tree of G, called a *depth-first search spanning tree* is constructed by a linear time algorithm by Tarjan [20] and has the property that if the nodes are preordered by the algorithm above, then for each cross edge (u, v), v is preordered before u.

A node u *dominates* a node v if every path from the root to v includes u (u *properly dominates* v if in addition, $u \neq v$). It is easily shown that there is a unique tree DT, called the *dominator tree* of G, such that u dominates v in G iff u is an ancestor of v in DT. The parent of a node in the dominator tree is the *immediate dominator* of that node. Figure 4 illustrates the dominator tree of the flow graph of Fig. 3.

The cycle edges are partitioned by their relation in the dominator tree DT.

(a) A-cycle edges are cycle edges from a node to a proper dominator.

(b) B-cycle edges are cycle edges between nodes unrelated on the dominator tree.

G is *reducible* if each cycle p of G contains a unique node dominating all other nodes in p. Programs written in a well-structured manner are often reducible. Various characterizations of reducibility are given by Hecht and Ullman [9]; in particular they show that:

THEOREM 2. *G is reducible iff G has no B-cycle edges.*

Lengauer and Tarjan give in [15] a test for reducibility requiring an almost linear number of elementary steps.

3. Approximate safe points of code motion. Text expression t is *safe* at node w if no new errors of computation are induced when t is relocated to node w. To approximate the safe point of t we require a good method for determining if t is safe at particular nodes.

A text expression t is *dependent* on a program variable if that variable occurs within the text of t (this need not imply functional dependence). The text expression t is *dangerous* if there exists some assignment of values to the variables on which t is dependent which induce an error in the computation of t. For example, an expression with a division operation is dangerous, since an error occurs if the divisor evaluates to zero. Following Kennedy [12], we say that there is an *exposed instance* of text expression t on a simple (acyclic) control path p if there is some text expression t' located in p, with the same text string as t, and such that no variable on which t is dependent is defined at any node in p occurring after the first node of p and before $\text{loc}(t')$. For each node w let $\text{SAFE}(w)$ consist of all text expressions which are not dangerous plus all dangerous text expressions which have an exposed instance on every simple control path from w to the final node f.

THEOREM 3.1. (due to Kennedy [12]). *If w occurs on the dominator chain from* $\text{BIRTHPT}(t)$ *to* $\text{loc}(t)$ *and* $t \in \text{SAFE}(w)$ *then t is safe at node w.*

Proof. Let P' be the program derived from P by relocating the computation of t to node w. If there is an error resulting from the computation of t on control path p in the modified program, then since $t \in \mathit{SAFE}(w)$ the error would also have occurred (although somewhat later) in the execution of the original program on control path p. □.

Recall the parameters $n = |V|$, $m = |E|$, and $l =$ the number of text expressions. Tarjan [24] presents an algorithm for solving certain general path problems, and which may be used to compute SAFE in a number of bit vector operations almost linear in $m + l$ if the program flow graph is reducible. Also, Graham and Wegman [8] and Hecht and Ullman [10] give algorithms for computing SAFE with time cost often linear in $l + m$, but with worst case time cost $\Omega(l + m \cdot \log(m))$ and $\Omega(l + n^2)$, respectively.

Let $\text{loc}(t)$ be the node where text expression t is located. To approximate the safe point of t, we take $\text{SAFEPT}(t)$ to be the first node w of the dominator chain from $\text{BIRTHPT}(t)$ to $\text{loc}(t)$ such that $t \in \text{SAFE}(w)$.

Let IDOM map from nodes in $V - \{s\}$ to their immediate dominators in G. For each $w \in N$, let $\text{EARLY}(w)$ consist of those text expressions t with $\text{BIRTHPT}(t) = w$ plus, if $w \neq s$, all $t \in \text{EARLY}(\text{IDOM}(w)) - \text{SAFE}(\text{IDOM}(w))$. Let $\text{LATE}(w)$ be the set of all text expressions $t \in \text{SAFE}(w)$ such that w dominates $\text{loc}(t)$. Intuitively, if $t \in \text{EARLY}(w)$ then t is defined at w but may not be safe at w. On the other hand, if $t \in \text{LATE}(w)$ then t is safe at w but may not be defined at w.

LEMMA 3.1. $\text{SAFEPT}(t) = w$ *iff* $t \in \text{EARLY}(w) \cap \text{LATE}(w)$.

Proof. Clearly, for each node w on the dominator chain from $\text{BIRTHPT}(t)$ to $\text{loc}(t)$, $t \in \text{LATE}(w)$ iff $\text{SAFEPT}(t)$ dominates w. Hence, for each node w on the dominator chain following $\text{BIRTHPT}(t)$ to $\text{SAFEPT}(t)$, if $t \in \text{EARLY}(\text{IDOM}(w))$ then since $t \notin \text{SAFE}(\text{IDOM}(w))$, $t \in \text{EARLY}(w)$. Also for any w on the dominator chain following $\text{SAFEPT}(t)$ to $\text{loc}(t)$, $t \in \text{SAFE}(\text{IDOM}(w))$, so $t \notin \text{EARLY}(w)$. Thus $w = \text{SAFEPT}(t)$

iff w dominates $\text{SAFEPT}(t)$ and $\text{SAFEPT}(t)$ dominates w

iff $t \in \text{EARLY}(w) \cap \text{LATE}(w)$. □

Lemma 3.1 leads to a simple algorithm for computing SAFEPT. EARLY is computed by a preorder pass through the dominator tree DT and LATE is computed by a postorder (i.e., reverse of the preorder of § 2) pass through DT.

ALGORITHM B.

```
INPUT Control flow graph $G=(N, E, s)$, the set of text expressions TEXT,
  BIRTHPT, and SAFE.
OUTPUT SAFEPT.
begin
  declare LATE, EARLY as arrays of length $n=|V|$;
  declare SAFEPT as an array length $l$;
  Compute the dominator tree DT of $G$;
  Number nodes in $V$ by a preordering of DT;
  for $w := 1$ to $n$ do
    L1. EARLY($w$) := LATE($w$) := the empty set { };
  for all text expressions $t \in$ TEXT do
    L2. add $t$ to EARLY(BIRTHPT($t$)) and LATE(loc($t$));
  for $w := 2$ to $n$ do
    L3. EARLY($w$):=EARLY($w$)$\cup$(EARLY(IDOM($w$))-SAFE(IDOM($w$)));
  for $w := n$ by $-1$ to 1 do
    begin
      for all siblings $u$ of $w$ in DT do
      L4. LATE($w$)=LATE($w$)$\cup$LATE($u$);
      comment Apply Lemma 3.1;
      for all $t \in$ EARLY($w$)$\cap$LATE($w$) do
      L5. SAFEPT($t$) := $w$;
    end;
end.
```

We assume that a bit vector of length l may be stored in a constant number of words and that in a constant number of bit vector operations we may determine the first nonzero element of a bit vector (this is not an unreasonable assumption since most machines have an instruction for left-justifying a word to the first nonzero bit).

THEOREM 3.2. *Algorithm*B *is correct and requires* $O(\max\{m\alpha(m), l\})$ *elementary and bit vector operations.*

Proof. The correctness of Algorithm B follows immediately from Lemma 3.1. The dominator tree DT may be constructed by an algorithm by Lengauer and Tarjan [15] in time almost linear in $m=|E|$, (if G is reducible, an algorithm due to Hecht and Ullman [10] computes DT in a linear number of bit vector operations). Steps L1, L2, L3, L4, L5 each require a constant number of elementary and bit vector operations and are executed $O(n)$, $O(l)$, $O(n)$, $O(n)$, $O(l)$ times, respectively. Since G is a flow graph, $m \geqq n-1$. Hence, the total time cost of Algorithm B is $O(\max\{m\alpha(m), l\})$ bit vector operations. □

4. Reduction of code motion to cycle problems.

For an arbitrary flow graph $G=(V, E, s)$ and $w, x \in V$ such that w dominates x in G, let $C1_G(w, x)$ be the latest node, on the dominator chain in G from w to x, which is contained on no w-avoiding cycles. Similarly, let $C2_G(w, x)$ be the first node, on this dominator chain, which is contained on no x-avoiding cycles. See Fig. 2.

LEMMA 4.1. *For nodes* $x, y \in V$ *such that* y *dominates* x, *let* M *be the list of nodes on the dominator chain from* y *to* x *and contained on no* x*-avoiding cycles, let* w *be the first element of* M, *and let* M' = *those nodes in* M *contained in a minimal number of cycles.*

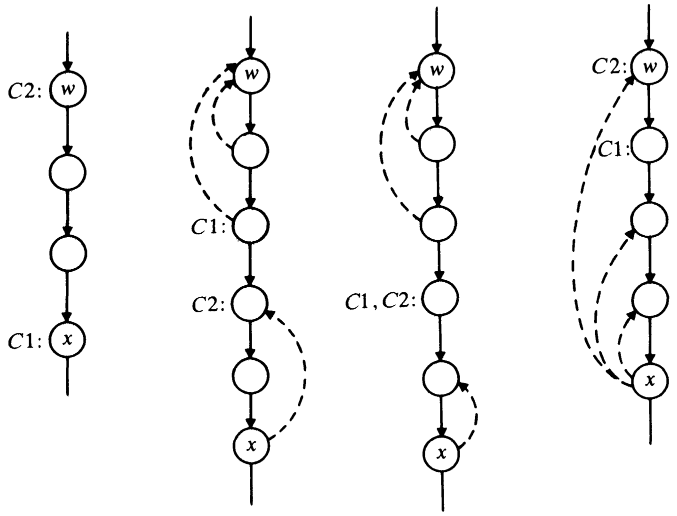


FIG. 2. *Examples of some dominator chains from w to x (with cycle edges dashed).*

Then $C1_G(w, x)$ is the latest node in M' relative to the dominator ordering of G.

Proof. Observe that $C1_G(w, x) \in M$; for otherwise $C1_G(w, x)$ is contained on a x-avoiding cycle which also contains w, a contradiction with the assumption that $w \in M$ is contained on no x-avoiding cycles.

Suppose p is a cycle containing $C1_G(w, x)$ and avoiding some $y \in M - \{C1_G(w, x)\}$. If y properly dominates $C1_G(w, x)$ then since w dominates y, p is w-avoiding, a contradiction with the assumption that $C1_G(w, x)$ is contained on no w-avoiding cycles. Otherwise, if y is properly dominated by $C1_G(w, x)$, then since y dominates w, p is x-avoiding, contradicting the assumption that $C1_G(w, x) \in M$.

Suppose $C1_G(w, x)$ properly dominates some $z \in M'$. If z is contained on no w-avoiding cycles, then z dominates $C1_G(w, x)$ by definition of $C1$, a contradiction. If z is contained on a w-avoiding cycle, then so is $C1_G(w, x)$, a contradiction. □

Let $G = (V, E, s)$ be the control flow graph. Our first variation of code movement, movept_1, may be described in terms of $C1_G$, $C2_G$, and SAFEPT.

THEOREM 4.1. *For each text expression t,*

$$\text{movept}_1(t) = C1_G(C2_G(\text{SAFEPT}(t), \text{loc}(t)), \text{loc}(t)).$$

Proof. Clearly, any node on the dominator chain from SAFEPT(t) to loc(t) satisfies R1–R3. Recall that M_1 consists of those nodes on the dominator chain from SAFEPT(t) to loc(t) which satisfy R4; i.e., they are contained on no control cycles avoiding loc(t). By definition of $C2_G$, $w = C2_G(\text{SAFEPT}(t), \text{loc}(t))$ is the first node in M_1 relative to the domination ordering in G. Hence by Lemma 4.1, $\text{movept}_1(t) = C1_G(w, \text{loc}(t))$ is the last node of M'_1 relative to the domination ordering. □

From the control flow graph $G = (V, E, s)$ we derive the *reverse control flow graph* $G = (V, E_R, f)$ which is a digraph rooted at the *final* node $f \in V$ and with edge set E_R derived from E by reversing all edges. G_R is assumed to be a flow graph; so every node is reachable in G_R from f.

LEMMA 4.2. *If x dominates y in G, y dominates z in G, and z dominates x in G_R, then y dominates x in G_R and z dominates y in G_R.*

Proof (by contradiction). Suppose there is a y-avoiding path p_1 in G_R from f to x. Since z dominates x in G_R, p_1 must contain z. The reverse of p_1, p_1^R, is a path in G. Since

x dominates y in G, there must be a y-avoiding path p_2 in G from s to x. Composing p_2 and p_1^R, we have a path in G from s to f which contains z but avoids y. But this contradicts our assumption that y dominates z in G. Hence, y dominates x in G_R. Similarly, we may easily show that z dominates y in G_R. □

THEOREM 4.2. *If w dominates x in G and x dominates w in G_R, then $C2_G(w, x) = C1_{G_R}(x, w)$.*

Proof. It is sufficient to observe by Lemma 4.2 that the dominator chain from w to x in G is the reverse of the dominator chain from x to w in G_R. The symmetries in the definition of $C2_G$ and $C1_G$ then give the result. □

Let HPT(t) be the first node on the dominator chain of G from BIRTHPT(t) to loc(t) which is dominated by loc(t) in the reverse flow graph G_R. It is useful to observe that t is safe at HPT(t) since there is an exposed instance of t on every path from HPT(t) to f. For each $w \in V$ let $H(w)$ be the first node, on the dominator chain in G from the start node s to w, which is dominated in G_R by w. H may be computed by a swift scan of the nodes in V, in preorder of the dominator tree of G by the following rule: $H(w) = H(x)$ if w dominates x in G_R, where x is the immediate dominator of w in G, and otherwise $H(w) = w$.

HPT is given from H by the following easy-to-prove lemma.

LEMMA 4.3. HPT(t) $= H$(loc(t)) *if* BIRTHPT(t) *dominates* H(loc(t)) *in G and otherwise* HPT(t) = BIRTHPT(t).

The next theorem expresses movept$_2$ in terms of $C1$ and HPT.

THEOREM 4.3. *For all text expressions t,*

$$\text{movept}_2(t) = C1_G(C1_{G_R}(\text{loc}(t), \text{HPT}(t)), \text{loc}(t)).$$

Proof. Recall that M_2 is the set of nodes $v \in M_1$ which satisfy restriction R5: That all control paths from v to f contain loc(t).

We claim that $w = C2_G(\text{HPT}(t), \text{loc}(t))$ is the first node in M_2 relative to the dominator ordering of G. Since t is safe at HPT(t), SAFEPT(t) dominates HPT(t), and so w is clearly an element of M_2. If there exists some $w' \in M_2$ which properly dominates w, then since w' satisfies restriction R5, loc(t) is contained on all paths from w' to f, which implies that HPT(t) dominates w', a contradiction.

By Theorem 4.2, $w = C2_G(\text{HPT}(t), \text{loc}(t)) = C1_{G_R}(\text{loc}(t), \text{HPT}(t))$. Hence, movept$_2(t) = C1_G(w, \text{loc}(t))$ is the last node in M_2' relative to the domination ordering. □

The next two sections describe how to compute $C1$ and $C2$ efficiently.

5. The computation of $C1$.

Let $G = (V, E, s)$ be an arbitrary flow graph with the nodes of V numbered from 1 to $n = |V|$ by a preordering of some depth-first search spanning tree ST of G. For certain $w, x \in V$ such that w dominates x in G, we wish to compute $C1_G(w, x)$; recall from § 4 that this is the last node on the dominator chain from w to x which is contained on no w-avoiding cycles.

For $w = n, n-1, \cdots, 2$ let $I(w)$ be the set of all $x \in V$ contained on a cycle of G consisting only of descendants of w in ST, and such that x is not contained in any $I(u) - \{u\}$ for $u > w$. The sets $I(n), I(n-1), \cdots, I(2)$ are related to the *intervals* of G (see [1]) and may be computed in almost linear time by an algorithm of [21].

Let IDOM(x) give the immediate dominator of node $x \in V - \{s\}$.

LEMMA 5.1 (due to Tarjan [21]). *For each $w \in V - \{s\}$ and $x \in I(w)$,* IDOM(w) *properly dominates x.*

Proof (by contradiction). Suppose the lemma does not hold; so there exists a IDOM(w)-avoiding path p from the root s to x. But by definition of $I(w)$, there exists a cycle q, avoiding all proper ancestors of w in ST and containing both w and x. Since IDOM(w) is a proper ancestor of w in ST, q avoids IDOM(w). Hence, we can construct from p and q a IDOM(w)-avoiding path from s to w, which is impossible. □

Our algorithm for computing $C1$ will construct, for each $w \in V$, a partition $PV(w)$ of the node set V. Initially, for $w = n$, $PV(w)$ consists of all singleton sets named for the nodes which they contain. For $w = n, n-1, \cdots, 2$ let $J(w)$ consist of $I(w)$ plus all nodes in V contained on a w-avoiding cycle and immediately dominated by some element of $I(w)$. Then $PV(w-1)$ is derived from $PV(w)$ by collapsing into w all sets with at least one element contained in $J(w)-\{w\}$.

For $w, x \in V$ such that w dominates x, let $g(w, x)$ be the name of the set of $PV(w)$ in which x is contained.

LEMMA 5.2. *$g(w, x)$ is an ancestor of x in* ST *and if $w > 1$,* IDOM$(g(w, x))$ *properly dominates x.*

Proof. By induction on w.

Basis step. For $w = n$, $g(w, x) = x$ and so IDOM$(g(w, x))$ = IDOM(x) properly dominates x.

Inductive step. Suppose, for some $w > 1$, the lemma holds for all $w' \geqq w$. Consider some $x \in V$ such that w dominates x.

Case 1. If $g(w-1, x) = g(w, x)$ then the lemma holds by the induction hypothesis.

Case 2. If $g(w-1, x) = w$ then in $PV(w)$, $g(w, x)$ contains some $y \in J(w)-\{w\}$. First we show that w is an ancestor of y in ST and IDOM(w) properly dominates y. If $y \in I(w)-\{w\}$, then w is an ancestor of y in ST by definition of $I(w)$, and IDOM(w) dominates y by Lemma 5.1. Otherwise, suppose $y \in (J(w)-I(w))-\{w\}$ so y is immediately dominated by some $y' \in I(w)$. Hence y' is a proper ancestor of y in ST and by definition of $I(w)$, w is an ancestor of y', so w is an ancestor of y' in ST. By Lemma 5.1, IDOM(w) properly dominates y', and hence IDOM(w) also properly dominates y.

Since the set $g(w, x)$ of $PV(w)$ contains y, $g(w, x) = g(w, y)$. By the induction hypothesis, $g(w, x) = g(w, y)$ is an ancestor of both x and y in ST. We have shown that w is an ancestor of y in ST. Since $w < g(w, x)$, w is a proper ancestor of $g(w, x)$ in ST, so w is also an ancestor of x in ST.

We claim that IDOM(w) properly dominates $g(w, x)$. If not, there would exist an IDOM(w)-avoiding path p from the root $s = 1$ to $g(w, x)$. IDOM(w) is an ancestor of w in ST and $g(w, x)$ is not an ancestor of w, so $g(w, x)$ is not an ancestor of IDOM(w) in ST. Also, since $g(w, x)$ is an ancestor of y in ST, there is a IDOM(w)-avoiding path p' of tree edges from $g(w, x)$ to y. Composing p and p', we have a IDOM(w)-avoiding path from s to $g(w, x)$, which is impossible since we have previously shown that IDOM(w) properly dominates y. Hence, IDOM(w) properly dominates $g(w, x)$. By the induction hypothesis, IDOM$(g(w, x))$ properly dominates x, and so IDOM(w) properly dominates x. □

THEOREM 5.1. *Consider any $x, w \in V$ such that w dominates x. If x is contained in no w-avoiding cycles then $g(w, x) = x$ and otherwise $g(w, x)$ is the highest ancestor of x in* ST *such that* IDOM$(g(w, x))$ *properly dominates x and all nodes, on the dominator chain following* IDOM$(g(w, x))$ *to x, are contained in w-avoiding cycles.*

Proof (sketch). If x is contained in no w-avoiding cycles in G then x cannot be contained in $I(w')$ for $w < w' \leqq x$ and so in this case $g(w, x) = x$.

Otherwise, consider the case where x is contained in some w-avoiding cycle. Suppose some node w' on the dominator chain following IDOM$(g(w, x))$ to IDOM(x) is *not* contained in a w-avoiding cycle. Then the set $g(w', x)$ of $PV(w')$ is not merged into w' in $PV(w'-1)$, so $g(w', x) = g(w'-1, x) \neq w'$. Furthermore we can show that for $y = w', w'-1, \cdots, g(w, x)+1$; $g(w', x) \notin J(y)$ so $g(w', x) = g(y, x) \neq y$. Hence

$g(w', x) = g(g(w, x), x) \neq g(w, x)$. Since $g(w, x)$ is the name of a set of $PV(w)$, $g(w, x)$ is not merged into any other set of $PV(g(w, x)), PV(g(w, x)-1), \cdots, PV(w)$, so $g(g(w, x), x) = g(w, x)$, and we have a contradiction.

Finally, suppose IDOM($g(w, x)$) is contained in some w-avoiding cycle p. Each such path p must contain a unique node w_p which dominates IDOM($g(w, x)$) and no node in p properly dominates w_p. Choose some such p with w_p as late as possible in the dominator ordering; i.e., as close as possible to IDOM($g(w, x)$). Then we can show that $g(w, x) \in J(w_p) - \{w_p\}$ and so $g(w, x)$ is merged into w_p in $PV(w_p - 1)$, which is impossible (since $g(w, x)$ is the name of a set in $PV(w)$). □

COROLLARY 5.1. *Let $w, x \in V$ such that w dominates x in G. If x is contained in no w-avoiding cycles then $Cl_G(w, x) = x$. Otherwise, $Cl_G(w, x) =$ IDOM($g(w, x)$).*

Proof. If x is contained in no w-avoiding cycles then, by definition, $Cl_G(w, x) = x$. Otherwise, suppose x is contained in some w-avoiding cycle. By Theorem 5.1, all nodes in the dominator chain following IDOM($g(w, x)$) to x are contained in w-avoiding cycles, so $Cl_G(w, x)$ properly dominates $g(w, x)$. Hence, IDOM($g(w, x)$) is the last node in the dominator chain from w to x which is contained in a w-avoiding cycle and we conclude that $Cl_G(w, x) =$ IDOM($g(w, x)$). □

We require the disjoint set operations:

(1) FIND(x) gives the name of the set currently containing node x.

(2) UNION(x, y): merge the set named x into the set named y.

The algorithm for computing Cl_G is given below.

ALGORITHM C.

INPUT Flow graph $G = (V, E, s)$ and ordered pairs $(w_1, x_1), \cdots, (w_l, x_l)$ such that each w_i dominates x_i.

OUTPUT $Cl_G(w_1, x_1), \cdots, Cl_G(w_l, x_l)$.

```
begin
  declare SET, BUCKET, FLAG to be arrays length $n = |V|$;
  Compute the depth-first spanning tree ST of $G$;
  Number the nodes in $V$ by preorder in ST;
  Compute the dominator tree DT;
  for $x := 1$ to $n$ do
    begin
      SET($x$) := $\{x\}$;
      BUCKET($x$) := the empty set { };
      FLAG($x$) := FALSE;
    end;
  for $i := 1$ to $l$ do add $x_i$ to BUCKET($w_i$);
  for $w := n$ by $-1$ to 1 do
    begin
      for all $x \in$ BUCKET($w$) do
        begin
          if FLAG($x$) then
            $Cl_G(w, x)$ := the parent of FIND($x$) in DT;
          else $Cl_G(w, x) := x$;
      if $w > 1$ then
        begin
          Compute $I(w)$ by the algorithm of [21];
          if $I(w)$ is not empty then
            begin
              for all $y \in I(w)$ do
```

```
                begin
                  $z$ := FIND($y$);
                    if NOT FLAG($z$) then
                    begin
                    $D$: for all $x \in \mathrm{IDOM}^{-1}(z)$ do
                        if FLAG($x$) then
                          UNION(FIND($x$), $w$);
                      FLAG($z$) := TRUE;
                    end;
                  if $z \neq w$ do UNION($z$, $w$);
                end;
            end;
        end;
    end;
end;
```

THEOREM 5.2. *Algorithm* C *correctly computes* $C1_G(w_1, x_1), \cdots, C1_G(w_l, x_l)$ *in time almost linear in* $m + l$.

Proof (sketch). We may show by an inductive argument that on entering the main loop on the $(n+1-w)$th iteration:

(1) FIND(x) gives $g(w, x)$;

(2) FLAG(x) iff x is contained in a w-avoiding cycle, and then apply Corollary 5.1 to show the correctness of Algorithm C.

ST, DT, and $I(n), I(n-1), \cdots, I(2)$ may be computed by the algorithms of [20], [15], [21] in time almost linear in $m = |E|$. The other steps of Algorithm C clearly require a linear number of elementary and disjoint set operations. These set operations may be implemented in almost linear time by an algorithm analyzed by Tarjan [22]. □

6. The computation of $C2$. The first formulation of code motion was shown to reduce to a number of subproblems including the calculation of the function $C2$; recall that for flow graph $G = (V, E, s)$ and each $w, x \in V$ such that w dominates x, $C2_G(w, x)$ is the first node on the dominator chain from w to x which is not contained on any x-avoiding cycles. For such w, x let a path from x to w, which avoids all proper dominators of x other than w, and which is either a simple (acyclic) path or a simple cycle (a cycle containing no other cycles as proper subsequences), be called a *dominator disjoint* (DD) *path*. Let DT be the dominator tree of G.

Our algorithm for computing $C2_G$ will require a function DDP such that for each $x \in V$, DDP(x) = x if $x = s$ or there is no DD path from IDOM(x), and otherwise DDP(x) is the first node y on the dominator chain from the root s to x such that there exists an x-avoiding DD path from IDOM(x) to y.

LEMMA 6.1. *If* DDP(x) *properly dominates* x *then all nodes on the dominator ordering from* DDP(x) to IDOM(x) *are contained on an x-avoiding cycle. Otherwise,* DDP(x) = x *and* IDOM(x) *is contained on no x-avoiding cycles.*

Proof. If DDP(x) properly dominates x, then let p be a DD path from IDOM(x) to DDP(x). ,Since DDP(x) dominates IDOM(x), there is an x-avoiding path p' from DDP(x) to IDOM(x). Hence $p \cdot p'$ is the required x-avoiding cycle.

On the other hand, suppose DDP(x) = $x \neq s$ and IDOM(x) is contained on an x-avoiding cycle q. Let q' be the subsequence of q from IDOM(x) to some node z immediately dominating x, and containing no other proper dominators on x. Then q' is a DD path, so DDP(x) properly dominates z, implying that DDP(x) $\neq x$, contradiction. □

LEMMA 6.2. *Let $z \in V$ have at least two siblings and be contained on a cycle avoiding some sibling of z in DT. Let $x_1(x_2)$ be a sibling of z with* DDP *value earliest (latest) in the dominator ordering. Then for each y which is properly dominated by z,* DDP(x_1) *is a dominator of* DDP(y); *furthermore, if $y \neq x_2$ and y is a sibling of z then* DDP$(y)=$ DDP(x_1).

Proof. Suppose z is a proper dominator of y, but DDP(y) is a proper dominator of DDP(x_1). Then DDP$(y) \neq y$ so there is a DD path p from IDOM(y) to DDP(y). Let x' be a sibling of z which is not a dominator of y. Let p' be a simple x'-avoiding path from z to y. Composing p' and p, we have an x'-avoiding DD path from z to DDP(y). But this implies that DDP(x') is a proper dominator of DDP(x_1), contradicting the assumption that x_1 has DDP value earliest in the dominator ordering. Hence, DDP(y) is dominated by DDP(x_1).

Suppose $y \neq x_2$ and y is a sibling of z. Since z is contained on a cycle avoiding some sibling of z, there must be a DD path $\bar{p}$ from z to DDP(x_1). If $\bar{p}$ avoids all siblings of z in DT, then we have our result; DDP$(y)=$ DDP(x_1). Otherwise, let $\bar{x}$ be the last node in $\bar{p}$ which is a sibling of z. Let $\bar{p}_1$ be the subsequence of $\bar{p}$ from $\bar{x}$ to z. For any $x' \in V-\{\bar{x}\}$, let p_2 be a x'-avoiding simple path from z to $\bar{x}$. Composing $\bar{p}_1$ and p_2, we have a x'-avoiding DD path from z to DDP(x_1). Hence, DDP$(x')=$ DDP(x_1). If $\bar{x}=x_2$ then $y \neq \bar{x}$ so we have DDP$(y)=$ DDP(x_1). On the other hand, if $\bar{x} \neq x_2$ then DDP$(x_2)=$ DDP(x_1). Since DDP(y) dominates DDP(x_2), we again have DDP$(y)=$ DDP(x_1). □

Let DT be the dominator tree of G with the edges oriented so that for each node $z \in V$ contained on a cycle avoiding some node immediately dominated by z, the left-most sibling of z in DT has DP value at least as late in the dominator ordering as the other siblings of z (by Lemma 6.2, the remaining siblings have the same DDP), and number V by a preordering of DT.

For each $x \in V-\{r\}$, let $K(x)$ consist of (1) the set of nodes contained on the dominator chain from DDP(x) to IDOM$(\dot{x})$ plus (2) the immediate dominator of DDP(x) if it is contained on a DDP(x)-avoiding cycle.

Let $PV'(1), PV'(2), \cdots, PV'(n)$ be a sequence of partitions of V such that:

(a) $PV'(1)$ partitions V into unit sets, each set named for the node which it contains.

(b) For $x=2, \cdots, n$ let $PV'(x)=PV'(x-1)$ if DDP$(x)=x$. Otherwise, let $PV'(x)$ be derived from $PV'(x-1)$ by collapsing each set containing an element of $K(x)-\{\text{IDOM}(x)\}$ into the set containing IDOM(x) in $PV'(x-1)$ and then renaming this set to IDOM(x).

For $w, x \in V$ such that w dominates x, let $h(w, x)$ be the name of the set containing w in $PV'(x)$.

THEOREM 6.1. *If w is contained in no x-avoiding cycles, then $h(w,x)=w$ and otherwise $h(w,x)$ is the last node on the dominator chain from w to x such that all nodes occurring up to and including $h(w,x)$ on this chain are contained on x-avoiding cycles.*

Proof. Let $(w=y_1, \cdots, y_k=x)$ be the dominator chain from w to x.

Suppose w is *not* contained on an x-avoiding cycle. Consider some node y_i on this dominator chain following w. If DDP(y_i) dominates w then by Lemma 6.1, w is contained in an x-avoiding cycle, a contradiction. Thus $w \notin K(y_i)-\{y_{i-1}\}$ and w is *not* collapsed into y_{i-1}, so $w=h(w,y_1)=\cdots=h(w,y_k)=h(w,x)$.

Otherwise, suppose w is contained on some x-avoiding cycle. Assume there is a node y_i, on the dominator chain following w to $h(w,x)$, which is *not* contained on an x-avoiding cycle. By Lemma 6.1, DDP$(y_i)=y_i$. Then $h(w,y_i)$ properly dominates y_i, so there is some $y_{j-1}=h(w,y_j)$ on the dominator chain from y_i to w such that DDP(y_j) dominates $h(w,y_i)$. By Lemma 6.1, y_i is contained on an x-avoiding cycle, a contradiction.

Finally, assume $h(w, x) \neq w$ and let y_i be the first node following $h(w, x)$ on the dominator chain from w to x. Suppose y_i is contained on an x-avoiding cycle. Then by Lemma 6.1, DDP(y_i) properly dominates y_i. Since $h(w, x) \neq w$, $h(w, x)$ is contained on an x-avoiding cycle, so $h(w, x) \in K(y_{i+1}) - \{y_i\}$ and hence $h(w, x)$ is merged into y_i, contradicting our assumption that $h(w, x)$ is the name of a set in $PV'(x)$. □

COROLLARY 6.1. *For $w, x \in V$ such that w dominates x, if w is contained on no x-avoiding cycles then $C2_G(w, x) = w$ and otherwise, $C2_G(w, x)$ is the unique node dominating x and immediately dominated by $h(w, x)$.*

Proof. The proof follows directly from Theorem 6.1.

Our algorithm for computing $C2$ will require the usual disjoint set operations UNION and FIND plus the operation RENAME(x, y), which renames the set x to y.

ALGORITHM D.

```
INPUT Flow graph $G = (V, E, s)$, DDP, and ordered pairs $(w_1, x_1), \cdots, (w_l, x_l)$
such that each $w_i$ dominates $x_i$.
OUTPUT $C2_G(w_1, x_1), \cdots, C2_G(w_l, x_l)$.
begin
  declare SET, FLAG, BUCKET to be arrays length $n = |V|$;
  Compute the dominator tree DT of $G$;
  for all $z \in V$ such that $z$ has a sibling $x$ in DT with
  DDP($x$) dominating $z$ do
    begin
      let $x'$ be the sibling of $z$ which has DDP($x'$)
      latest in the dominator ordering;
      install $x'$ as the left-most sibling of $z$;
    end;

  Number the nodes of $V$ by the preordering of the resulting oriented tree;
  for $x := 1$ to $n$ do
    begin
      SET($x$) := $\{x\}$;
      FLAG($x$) := FALSE;
      BUCKET($x$) := the empty set { };
    end;
  for $i := 1$ to $l$ do add $w_i$ to BUCKET($x_i$);
  for $x := 1$ to $n$ do
    begin
      if $x > 1$ and DDP($x$) $\neq x$ then
        begin
          $z$ := the parent of $x$ in DT;
          FLAG($z$) := TRUE;
          NEXT($z$) := $x$;
          RENAME(FIND($z$), $z$);
          $y$ := the parent of DDP($x$) in DT;
      D:  if FLAG($y$) and $y \neq z$ do
            UNION($y$, $z$);
          $u$ := FIND(DDP($x$));
          till $u = z$ do
            begin
              FLAG($u$) := TRUE;
              UNION($u$, $z$);
```

```
                $u$ := FIND(NEXT($u$));
            end;
        end;
        comment Apply Corollary 6.1;
        for all $w \in$ BUCKET($x$) do
            if FLAG($w$) then $C2_G(w, x)$ := NEXT(FIND($w$))
            else $C2_G(w, x)$ := $w$;
    end;
end;
```

THEOREM 6.2. *Algorithm* D *correctly computes* $C2_G(w_1, x_1), \cdots, C2_G(w_l, x_l)$ *in time almost linear in* $m+l$, *where* $m=|E|$.

Proof (sketch). It is possible to establish that for all $w \in V$ after the xth iteration of the main loop:

(1) NEXT(IDOM(w)) $= w$ for $w \neq s$ and w properly dominates x.

(2) The sets are just as in $PV'(x)$, with $h(w, x)$ the name of the set containing w.

(3) FLAG(w) = ŢRUE iff w is not contained in a x-avoiding cycle.

Then the correctness follows from Corollary 6.1.

We compute DT by the algorithm of [15] in time almost linear in $m+l$. The other steps of Algorithm D may easily be shown to require a linear number of elementary and disjoint set operations. Hence, by the results of [22], the total cost in elementary operations is almost linear in $m+l$. □

7. Computing DDP on reducible flow graphs. This section is concerned with the function DDP required by Algorithm D to compute $C2$. Unfortunately, we know of no algorithm which computes DDP efficiently for G nonreducible. We assume henceforth that G is reducible, so by the results of Hecht and Ullman [9], all cycle edges of G are A-cycle edges (they lead from nodes to their proper dominators). Let ST′ be the spanning tree derived from the depth first search spanning tree ST of G by reversing the edge list. The nodes of G are numbered by a preordering of ST′.

LEMMA 7.1. *If* $x > y$ *and both* x *and* y *are unrelated in DT, then any path* p *from* x *to* y *contains a dominator of* x.

Proof. It is sufficient to assume that p is simple (acyclic). Let (u, v) be the first edge through which p passes such that $v \leqq y < u$. Observe that the only edges of G in decreasing preorder are A-cycle edges, so (u, v) is an A-cycle edge and v dominates u. We claim also that v dominates x. Suppose not, so there is a v-avoiding path p' from the root s to x. Composing p' with the subsequence of p from x to u, we have a v-avoiding path from s to u, which contradicts the fact that v dominates u. Hence, v dominates x. □

We now show that in the reducible flow graph G, DD paths have a very special structure. Let $p=(x=y_0, \cdots, y_k=w)$ be a DD path from x to w passing through edges $e_1, \cdots, e_k$, where $e_i=(y_{i-1}, y_i)$.

THEOREM 7.1. e_k *is an* A-*cycle edge and* $e_1, \cdots, e_{k-1}$ *are not.*

Proof. Since p cannot contain any dominators of x other than w, y_{k-1} and x are unrelated in DT. Assume $e_k=(y_{k-1}, w)$ is *not* an A-cycle. Hence, $x > w > y_{k-1}$ and applying Lemma 7.1, $(x=y_0, \cdots, y_{k-1})$ must contain a node z which is a proper dominator of x, contradicting our assumption that p is DD.

Consider any $e_i=(y_{i-1}, y_i)$ for $1<i<k$. Since p is DD, y_i does not dominate x. Thus, there is a y_i-avoiding path p_1 from the root s to x. Also, let p_2 be the subsequence of p from x to y_{i-1}. Composing p_1 and p_2, we have a y_i-avoiding path from the root s to y_{i-1}, which implies that y_{i-1} is not dominated by y_i. Hence, none of $e_1, \cdots, e_{k-1}$ are A-cycles. □

THEOREM 7.2. *Let p be a DD path from x to w, where w properly dominates x and let z be an immediate predecessor of x in G such that z, x are unrelated in DT. Then $p' = (z, x) \cdot p$ is a DD path avoiding all siblings of z in DT.*

Proof. To show that p' is DD we need only demonstrate that w properly dominates z and p avoids z. Let $p = (x = y_0, \cdots, y_k = w)$. Since z, x are unrelated in DT and w properly dominates x, w is distinct from z.

We claim that w properly dominates z in G. Suppose not, then there must be a w-avoiding path p_1 from the root s to z. But $p_1 \cdot (z, x)$ is a w-avoiding path from the root s to x, contradicting our assumption that w properly dominates x. Hence, w properly dominates z.

Suppose p contains z, so $z = y_i$ for some $1 < i < k$. Then $(z, x = y_1, \cdots, y_i = z)$ is a cycle in G and must contain an A-cycle edge. Since z, x are unrelated in DT, this implies that for some j, $1 \leqq j \leqq i$, (y_{j-1}, y_j) is an A-cycle edge, contradicting Theorem 7.1. We conclude that p avoids z.

Hence, $p' = (z, x) \cdot p$ is DD.

Now suppose p contains a node y dominated by z. Since x, z are unrelated in DT, there must be a z-avoiding path p_2 from the root s to x. Composing p_2 and the portion of p from x to y, we have a z-avoiding path from s to y, which is impossible. Hence, $p' = (z, x) \cdot p$ avoids all siblings of z in DT. □

Let p be a DD path from x to w. Let the first edge (u, v) through which p passes, such that u is dominated by x but v is not properly dominated by x, be called the *first jump edge* of p.

THEOREM 7.3. *Let x' be a proper dominator of x. If either* (1) *$v = w$ dominates x' or* (2) *$v \neq w$ and* IDOM(v) *properly dominates x', then there exists a DD path from x' to w with first jump edge $e = (u, v)$.*

Proof. Let p_1 be a simple path from x' to x. Suppose p_1 contains some node z not dominated by x'. Then the subsequence of p_1 from z to x must contain x'. But this implies that x' occurs twice in p_1, which is impossible. Hence, all nodes in p_1 are dominated by x' and $p_2 = p_1 \cdot p$ is a DD path. Since x' properly dominates x which dominates u, x' also dominates u. If either (1) or (2) hold, then v does not properly dominate x'. Thus, the first jump edge of p_2 is $e = (u, v)$. □

ALGORITHM E.

```
INPUT A reducible flow graph $G = (V, E, s)$.
OUTPUT DDP.
begin
  declare SET, FLAG, DDP, SIBLINGS to be arrays length $n = |V|$;
  procedure EXPLORE($x$, $w$, $e$):
    begin
      comment there is a DD path from $x$ to $w$
        and $e$ is the first jump edge of $p$;
      Let $e = (u, v)$;
      for each $y \in$ SIBLINGS($x$) such that $y$, $u$ are
        unrelated in DT do
        begin
          delete $y$ from SIBLINGS ($x$);
          DDP($y$) := $w$;
        end;
      if $x \neq s$ and not FLAG($x$) then
        begin
          FLAG($x$) := TRUE;
          $x'$ := IDOM($x$);
```

```
            D: if FLAG($x'$) then
                  UNION($x$, FIND($x'$));
               if NOT $x = w$ then
                  begin
                     comment Apply Theorem 7.3;
                     if ($v = w$ dominates $x'$) OR ($v \neq w$ and
                     IDOM($v$) properly dominates $x'$) then
                        L1. EXPLORE($x'$, $w$, $e$);
                     comment Apply Theorem 7.2;
                     for all immediate predecessors $z$
                     of $x$ in $G$ such that $x$, $z$ are unrelated
                     in DT do
                        L2. EXPLORE($z$, $w$, ($z$, $x$));
                  end;
            end;
      end;
   Compute DT, the dominator tree of $G$;
   Compute ST, a depth-first spanning tree of $G$;
   Let ST′ be derived from ST by reversing the edge list;
   Number the nodes of $V$ by preorder of ST′;
   for all $x := 1$ to $n$ do
      begin
         SET($x$) := {$x$};
         FLAG($x$) := FALSE;
         DDP($x$) := $x$;
         SIBLINGS ($x$) := the siblings of $x$ in DT;
      end;
   for $w := 1$ to $n$ do
      for all $A$-cycle edges ($x$, $w$) entering $w$ do
         L3. EXPLORE($x$, $w$, ($x$, $w$));
end;
```

LEMMA 7.2. *On each execution of* EXPLORE(x, w, e), *w dominates x and there is a DD path from x to w with first jump edge e.*

Proof (by structural induction). On each initial call to EXPLORE(x, w, e) at label L3, e is a A-cycle edge (x, w) which is clearly a DD path. Suppose on any other call to EXPLORE(x, w, e) there is a DD path from x to w with first jump edge e. By Theorems 7.3 and 7.2, the recursive calls to EXPLORE at L1 and L2, respectively, also satisfy this lemma. □

It is also easy to prove by structural induction that the following holds.

LEMMA 7.3. *On each execution of* EXPLORE(x, w, e), *let y be a dominator of x contained in the set named* FIND(y). *If y has not previously been visited then* FLAG(y) = FALSE *and* FIND$(y) = y$; *otherwise,* FLAG(y) = TRUE *and* FIND(y) *is the earliest node y' on the domination chain from the root s to y such that all nodes from y' to y on this chain have been previously visited.*

Let p be a DD path from x to w with first jump edge $e = (u, v)$. For $k > 1$, the kth *jump edge* of p is recursively defined to be the $(k-1)$th jump edge (if this is defined and is not the last edge through which p passes) of the subsequence of p from v to w.

LEMMA 7.4. *For each $w, y \in V$ such that w properly dominates y, if there exists a y-avoiding DD path p from* IDOM(y) *to w, then* EXPLORE(IDOM(y), w, e) *is eventually called, where $e = (u, v)$ is the first jump edge of some such p.*

Proof (by induction on w). Suppose the lemma holds for all $w' < w$. Since $e = (u, v)$ is the first jump edge of p, IDOM(y) dominates u. If $v = w$, then (u, v) is an A-cycle edge so EXPLORE(u, w, (u, w)) is executed at label L3, and by a sequence of recursive calls to EXPLORE at label L1, we finally have a call to EXPLORE(IDOM(y), w, (u, v)). Otherwise, suppose the lemma holds for all p leading to w such that p has less than k jump edges. If p has k jump edges, then by the second induction hypothesis, EXPLORE(u, w, (u, v)) is called at label L2. Again, by a sequence of recursive calls to EXPLORE at label L1, we eventually have a call to EXPLORE(IDOM(y), w, (u, v)). □

THEOREM 7.4. *Algorithm* E *correctly computes DDP for G reducible, in time almost linear in* $m = |E|$.

Proof. The correctness of Algorithm E follows from Lemmas 7.2, 7.3, and 7.4. ST and DT may be computed (if they have not been computed previously) by the methods of [20], [15] in almost linear time. For each $x \in V$, the total cost of *all* visits to x by EXPLORE is $|\text{IDOM}^{-1}[x]| + |\text{indegree}(x)|$ in elementary and disjoint set operations. Hence, if we use a good implementation of disjoint set operations (analyzed by Tarjan [22]), the total cost of Algorithm E is almost linear in m. □

8. Niche flow graphs. Here we introduce a special class of flow graphs called niche flow graphs which in certain cases simplify the algorithms given in § 5 and § 6 for computing $C1$ and $C2$. As we shall demonstrate, the transformation of an arbitrary flow graph to a niche flow graph can be done in almost linear time; furthermore, both versions of code motion are improved by this transformation. Earnest [6] and Aho and Ullman [2] describe a similar process, where special nodes are added to the flow graph just above intervals.

Let $G = (V, E, s)$ be an arbitrary flow graph. For any $w \in V - \{s\}$ with immediate dominator IDOM(w) in G, if IDOM(w) is contained on no w-avoiding cycles then IDOM(w) is called the *niche node of* w. Intuitively, the niches nodes lie just above cycles (relative to the dominator ordering of G) and hence are good nodes to move code into. G is a *niche flow graph* if each node $w \in V - \{s\}$, with an entering A-cycle edge but *no* entering B-cycle edge, has a niche node.

If G is not a niche flow graph, then a niche flow graph G' may be derived from G by testing for each $w \in V - \{s\}$ whether w has an entering A-cycle edge and no entering B-cycle edges. If so, then add a distinct, new node $\hat{w}$ which is to be the niche of w in G', an edge from $\hat{w}$ to w, and replace each noncycle edge (x, w) entering w with a new edge $(x, \hat{w})$. The resulting flow graph G' has no more than $n = |V|$ additional nodes and edges. Since no B-cycle edges are added to G', by Theorem 2, G' is reducible if G was.

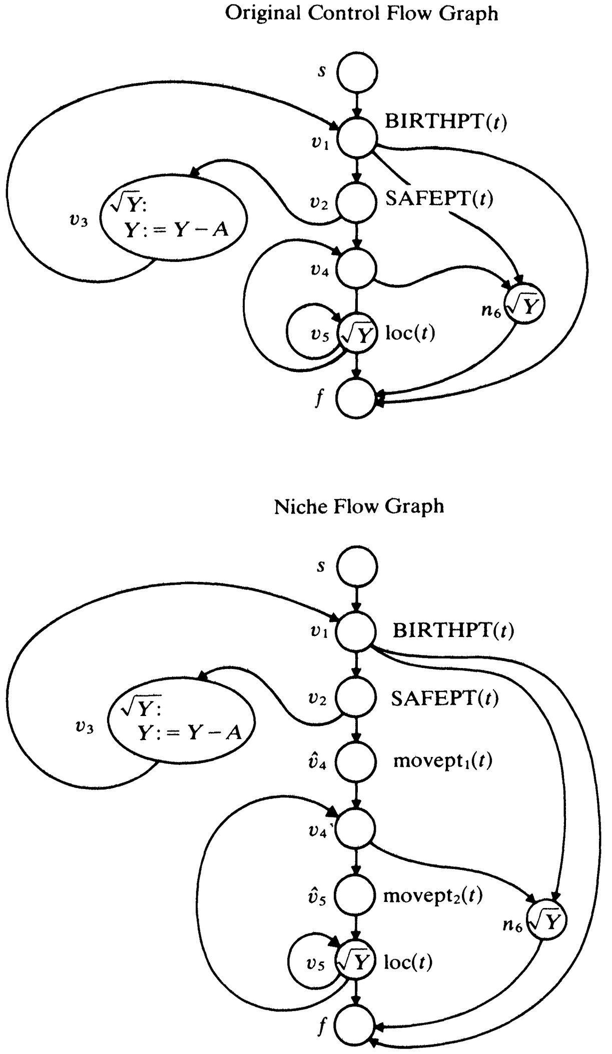
LEMMA 8.1. *If G is reducible and $y \in V - \{s\}$ is contained in an* IDOM(y)-*avoiding cycle q, then y has an entering A-cycle edge.*

Proof. Let x be the immediate predecessor of y in q. Since G is reducible, q contains a unique node z dominating all other nodes in q. But no proper dominator of y is contained in q, so $z = y$. Hence, y dominates x and (x, y) is an *A-cycle edge.* □

Let the nodes of G be numbered as in § 5 by a preordering of a depth-first search spanning tree of G.

THEOREM 8.1. *If G is a reducible niche flow graph, then for $w = n, n-1, \cdots, 2$ the partition $PV(w-1)$ is derived from $PV(w)$ by collapsing sets $I(w) - \{w\}$ into w.*

Proof. Recall that $PV(w-1)$ is defined to be derived from $PV(w)$ by collapsing into w each set z containing at least one element $y \in J(w) - \{w\}$. Suppose there is a set $z \notin I(w)$ in $PV(w)$ containing some $y \in (J(w) - I(w)) - \{w\}$. Then, by definition of $J(w)$, y is contained on a w-avoiding cycle q and IDOM(y) $\in I(w)$. But since $z \notin I(w)$, q avoids



Original Control Flow Graph

Niche Flow Graph

t is the text expression $\sqrt{Y}$ located at v_5

FIG. 3. *Transformation of a flow graph G into a niche flow graph G'.*

IDOM(y) and IDOM(y) is contained in a y-avoiding cycle q'. By Lemma 8.1, y has an entering A-cycle edge. Since G is a niche flow graph, IDOM(y) is the niche of y. But this is impossible since IDOM(y) is contained on a y-avoiding cycle q'. □

The above theorem allows us to simplify Algorithm D, which was used to compute Cl_G, in the case G is a reducible niche flow graph. In particular, the statement labeled D may be deleted from Algorithm D. Similarly, in this case the statement labeled D may be deleted from Algorithm E.

THEOREM 8.2. *If G is a reducible niche node and* DDP(x) $\neq x$, *then* $K(x)$ = *those nodes of the dominator chain from* DDP(x) *to* IDOM(x).

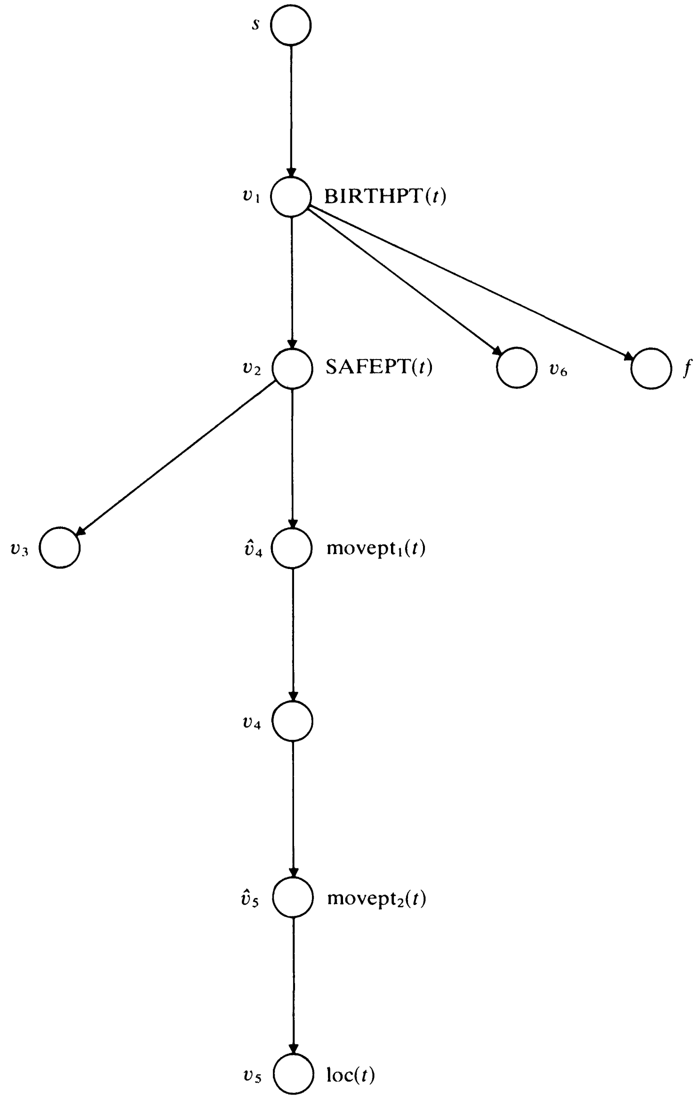


FIG. 4. *The dominator tree of the flow graph G'.*

Proof. Suppose there exists some $x \in V$ such that DDP(x) properly dominates x and IDOM(DDP(x)) is contained on a DDP(x)-avoiding cycle. Let p be the DDP path from x to DDP(x) and let p' be a simple path from DDP(x) to x. Composing p and p', we have a IDOM(DDP(x))-avoiding cycle containing DDP(x). Hence by Lemma 8.1, DDP(x) has an entering A-cycle edge. Since G is a niche flow graph, IDOM(DDP(x)) is the niche node of x. But by hypothesis, this niche node of DDP(x) is contained on a DDP(x)-avoiding cycle, which is impossible. □

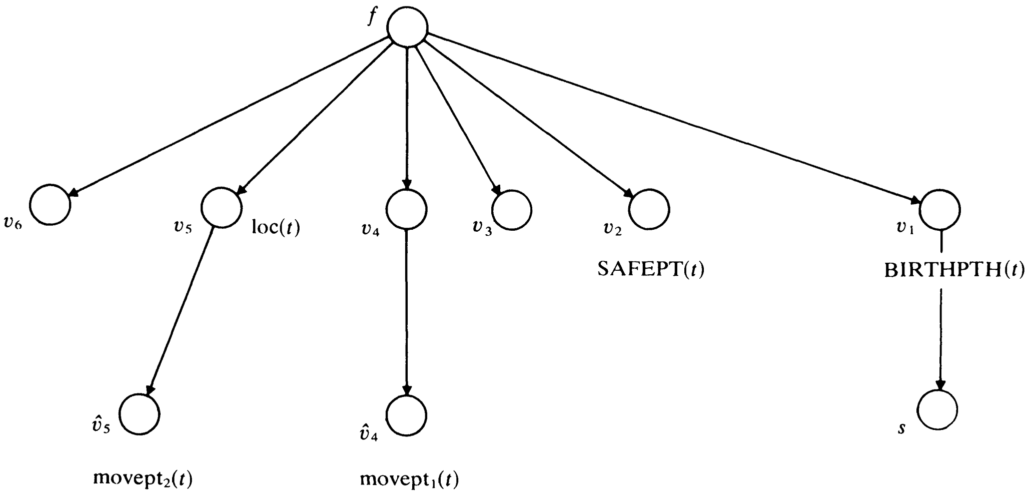


FIG. 5. *The dominator tree of the reverse of the flow graph G'.*

REFERENCES

[1] F. E. ALLEN, *Control flow analysis*, SIGPLAN Notices, 5 (1970), pp. 1–19.

[2] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing Translation and Compiling, II*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[3] ———, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977, pp. 454–466.

[4] V. A. BUSAM AND D. E. ENGLUND, *Optimization of expressions in Fortran*, Comm. ACM, 12 (1969), pp. 666–674.

[5] J. COCKE AND F. E. ALLEN, *A catalogue of optimization transformations*, Design and Optimization of Computers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, pp. 1–30.

[6] C. EARNEST, *Some topics in code optimization*, J. Assoc. Comput. Mach., 21 (1974), pp. 76–102.

[7] C. M. GESCHKE, *Global program optimizations*, Ph.D. thesis, Carnegie–Mellon Univ., Dept. of Computer Science, Pittsburgh, October 1972.

[8] S. GRAHAM AND M. WEGMAN, *A fast and usually linear algorithm for global flow analysis*, J. Assoc. Comput. Mach., 23 (1976), pp. 172–202.

[9] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[10] ———, *Analysis of a simple algorithm for global flow problems*, this Journal, 4 (1975), pp. 519–532.

[11] J. B. KAM AND J. D. ULLMAN, *Global data flow problems and iterative algorithms*, J. Assoc. Comput. Mach., 23 (1976), pp. 158–171.

[12] K. KENNEDY, *Safety of code motion*, Internat. J. Comput. Math., 3 (1971), pp. 5–15.

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

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

[15] R. LENGAUER AND R. E. TARJAN, *A fast algorithm for finding dominators in a flow graph*, Trans. Programming Languages and Systems, 1 (1979), to appear.

[16] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[17] J. H. REIF, *Combinatorial aspects of symbolic program analysis*, Ph.D. thesis, Harvard Univ. Division of Engineering and Applied Physics, Cambridge, MA, 1977.

[18] J. H. REIF AND H. R. LEWIS, *Symbolic evaluation and the global value graph*, Fourth ACM Symposium on Principles of Programming Languages, Jan., 1977; *Symbolic evaluation, Part I*, J. Assoc. Comput. Mach., submitted.

[19] J. H. REIF AND R. E. TARJAN, *Symbolic program analysis in almost linear time*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, January, 1978; this Journal, submitted.

[20] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[21] ———, *Testing flow graph reducibility*, J. Comput. System Sci., 9 (1974), pp. 355–365.

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

[23] ———, *Applications of path compression on balanced trees*, Tech. Rep. 512, Stanford Computer Science Dept., Stanford Univ., Stanford, CA, August 1975.

[24] ———, *Solving path problems on directed graphs*, Tech. Rep. 528, Stanford Computer Science Dept., Stanford Univ., Stanford, CA, October 1975.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0012 $01.00/0

ON THE COMPUTATIONAL COMPLEXITY OF PROGRAM SCHEME EQUIVALENCE*

H. B. HUNT III†, R. L. CONSTABLE‡ AND S. SAHNI¶

Abstract. The computational complexity of several decidable problems about program schemes, recursion schemes, and simple programming languages is considered. The strong equivalence, weak equivalence, containment, halting, and divergence problems for the single variable program schemes and the linear monadic recursion schemes are shown to be *NP*-complete. The equivalence problem for the Loop 1 programming language is also shown to be *NP*-complete. Sufficient conditions for a program scheme problem to be *NP*-hard are presented. The strong equivalence problem for a subset of the single variable program schemes, the strongly free schemes, is shown to be decidable deterministically in polynomial time.

Key words. computational complexity, *P*, *NP*, *NP*-complete, program scheme, recursion scheme, equivalence, containment, halting, divergence, and isomorphism

Introduction. Early work with program schemes was motivated by a quest for program optimization techniques [9], [10], [13]. Ideally one would find a class of schemes rich enough to model many interesting programs but simple enough to have decidable problems such as equivalence, halting, or divergence. No attempt was made, however, to assess the computational complexity of such decidable problems. Here, we show that a variety of such decidable problems for the single variable program schemes, the linear monadic recursion schemes, and several simple programming languages are *NP*-complete.

The remainder of this paper is divided into four sections. Section 1 contains definitions and basic properties of *p*-reducibility, program schemes, and recursion schemes. In § 2 the strong equivalence, weak equivalence, containment, halting, and divergence problems for the single variable program schemes and the linear monadic recursion schemes are shown to be *NP*-complete. We also present general sufficient conditions for a problem on the single variable program schemes to be *NP*-hard. In § 3 we consider subclasses of the single variable program schemes for which strong equivalence is decidable deterministically in polynomial time. Finally in § 4, we briefly consider the complexity of the equivalence problem for several classes of simple programming languages including the Loop 1 languages in [14].

1. Definitions.

We present definitions and basic properties of *p*-reducibility, program schemes, and monadic recursion schemes needed in §§ 2, 3, and 4. The definitions of strings, alphabets, context-free grammars, and derivations used here are from [7]. We denote the empty word by λ.

DEFINITION 1.1. $P(NP)$ is the class of all languages over $\{0, 1\}$ accepted by some deterministic (nondeterministic) polynomially time-bounded Turing machine.

DEFINITION 1.2. Let Σ and Δ be finite alphabets. Let $\mathscr{F}(\Sigma, \Delta)$ denote the set of all functions from Σ^* into Δ^* computable by some deterministic polynomially time-bounded Turing machine. Let L_1 and L_2 be subsets of Σ^* and Δ^*, respectively. We say

* Received by the editors February 14, 1975, and in final revised form May 2, 1979. This research was supported in part by the National Science Foundation under grants DCR-74-14701, GJ 35570, and DCR-75-22505.

† Department of Electrical Engineering and Computer Science, Columbia University, New York, New York 10027.

‡ Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853.

¶ Department of Computer, Information and Control Sciences, University of Minnesota, Minneapolis, Minnesota 55455.

that L_1 is *p-reducible* to L_2, written $L_1 \leqq_{\text{ptime}} L_2$, if there exists a function f in $\mathcal{F}(\Sigma, \Delta)$ such that, for all $x \in \Sigma^*$, $x \in L_1$, if and only if, $f(x) \in L_2$.

DEFINITION 1.3. A language L_0 is said to be *NP-hard* if, for all L in *NP*, $L \leqq_{\text{ptime}} L_0$. A language L_0 is said to be *NP-complete* if it is *NP*-hard and is accepted by some nondeterministic polynomially time-bounded Turing machine.[1]

DEFINITION 1.4. A Boolean form f is a *D_3-Boolean form* if f is the disjunction of clauses $C_1, \cdots, C_p$ such that each clause C_i is the conjunction of at most three literals. A Boolean form f is a *C_3-Boolean form* if f is the conjunction of clauses $C_1, \cdots, C_p$ such that each clause C_i is the disjunction of at most three literals.

PROPOSITION 1.5[3]. *The sets $\mathcal{T}_1 = \{f | f$ is a nontautological D_3-Boolean form$\}$ and $\mathcal{T}_2 = \{f | f$ is a satisfiable C_3-Boolean form$\}$ are NP-complete.*

PROPOSITION 1.6[3]. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages. If $\mathcal{L}_1$ is NP-hard and $\mathcal{L}_1$ is p-reducible to $\mathcal{L}_2$, then $\mathcal{L}_2$ is NP-hard.*

DEFINITION 1.7. Let D be a set. A *predicate on D* is a function from D into {True, False}.

We assume that the reader is familiar with the basic properties and results concerning program schemes, monadic recursion schemes, and interpretations as presented in [1], [4], [10].

Program schemes are defined as follows. Let $\mathcal{L}$, $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{P}$ be mutually disjoint sets of labels, variable symbols, function symbols, and predicate symbols, respectively. A *program scheme S* is a finite nonempty sequence of

(1) *assignment statements* of the form k . $y \leftarrow f(x_1, \cdots, x_n)$, where k in $\mathcal{L}$ is a label, f in $\mathcal{F}$ is an n-ary function symbol, and $x_1, \cdots, x_n$, y in $\mathcal{V}$ are variable symbols;
(2) *conditional statements* of the form k. *If* $P_j(x_1, \cdots, x_n)$ *then* k_1 *else* k_2, where k, k_1, and k_2 are labels, P_j in $\mathcal{P}$ is an n-ary predicate symbol, and $x_1, \cdots, x_n$ in $\mathcal{V}$ are variable symbols; and
(3) *halt statements* of the form k. *Halt*, where k is a label.

We sometimes allow *loop statements* of the form k. *Loop* as abbreviations for the statement.

$$k \text{ . } \mathbf{If}\ P_i(x_1, \cdots, x_n)\ \mathbf{then}\ k\,\mathbf{else}\ k.$$

We frequently assume that the first element of S is its initial statement and the last element of S is either a loop or halt statement.

The meaning of a program scheme S is defined in terms of interpretations. Formally, an *interpretation* I of S consists of

(1) a nonempty set D, called the *domain* of I;
(2) an assignment of an element of D to each variable symbol in $\mathcal{V}$;
(3) an assignment of a function $f^I : D^n \to D$ to every n-ary function symbol f in $\mathcal{F}$; and
(4) an assignment of a predicate $P_i^I : D^n \to D$ to every n-ary predicate symbol P_i in $\mathcal{P}$.

The definition of a *computation of* a program scheme *S under an interpretation I* can be found in [10]. The *value of S under I*, denoted by $\text{val}_I(S)$, is the final value of the distinguished output variable of S if the computation of S under I halts; and is undefined otherwise.

A *monadic recursion scheme S* is a finite list of definitional equations

$$F_1x := \mathbf{If}\ P_1x\ \mathbf{then}\ \alpha_1x\ \mathbf{else}\ \beta_1x,$$

$$F_nx := \mathbf{If}\ P_nx\ \mathbf{then}\ \alpha_nx\ \mathbf{else}\ \beta_nx$$

[1] Definition 1.3 extends the concept of *NP*-completeness to languages over arbitrary finite alphabets.

where $F_1, \cdots, F_n$ are *defined function symbols*; $P_1, \cdots, P_n$ are (not necessarily distinct) *predicate symbols*; and $\alpha_1, \beta_1, \cdots, \alpha_n, \beta_n$ are (possibly empty) strings of defined and basis symbols. A monadic recursion scheme S is said to be *linear* if at most one defined function symbol occurs in each of the strings $\alpha_1, \beta_1, \cdots, \alpha_n, \beta_n$.

The semantics of a monadic recursion scheme S is also defined in terms of interpretations. Formally, an *interpretation* I of a monadic recursion scheme S consists of

(1) a nonempty set D, called the *domain* of I;
(2) an assignment of a function $f^I : D \rightarrow D$ to every basis function symbol f is S;
(3) an assignment of a predicate $P_i^I : D \rightarrow \{\text{True, False}\}$ to every predicate symbol P_i in S; and
(4) an assignment of an element x^I of D to x.

An interpretation I of a monadic recursion scheme S, with set of basis function symbols $\mathscr{F}$, is said to be *free* if

(i) the domain D of I equals $[\mathscr{F}]^* \cdot \{x\}$; and
(ii) for all f in $\mathscr{F}$ and strings wx in $[\mathscr{F}]^* \cdot \{x\}$, $f^I(wx)$ equals the string fwx.

For any interpretation I, $(f_1 \cdots f_n x)^I = (f_1)^I(\cdots (f_n)^I(x^I) \cdots)$.

The computations of a monadic recursion scheme can be defined in terms of context-free grammars as follows. To each scheme

$$S\,.\ F_i x := \textbf{If } P_i x \textbf{ then } \alpha_i x \textbf{ else } \beta_i x \qquad (1 \leqq i \leqq n),$$

we associate a context-free grammar G_S with terminal alphabet equal to $\mathscr{F}$, nonterminal alphabet F equal to $\{F_1, \cdots, F_n\}$, and set of productions equal to $\{F_i \rightarrow \alpha_1, F_i \rightarrow \beta_i | 1 \leqq i \leqq n\}$. Let I be an interpretation. Following [4] we say that a rightmost derivation of G_S is *legal* for I if, for every step in the deviation of the form $\gamma F_i w \underset{G_S}{\Longrightarrow} \gamma \delta w$, where $\gamma, \delta \in (\mathscr{F} \cup F)^*$ and $w \in \mathscr{F}^*$, $\delta = \alpha_i$ if $P_i^I(w^I) = \text{True}$ and $\delta = \beta_i$ if $P_i^I(w^I) = \text{False}$. The computation of S under I corresponds to the unique legal derivation for I. If $F_1 \underset{G_S}{\Longrightarrow}^* w$ for $w \in \mathscr{F}^*$ by the legal derivation for I, then $\text{val}_\text{I}(\text{S}) = \text{w}^\text{I}(\text{x}^\text{I})$; otherwise, $\text{val}_I(S)$ is undefined.

Finally we assume that there is a finite alphabet Σ such that each scheme or program S is presented as a string σ_S over Σ. We say that the length of the string σ_S is the *size* of S.

DEFINITION 1.8. Let S and S' be program or monadic recursion schemes. We say that

(1) *S halts* if, for all interpretations I of S, the computation of S under I halts;
(2) *S diverges* if, for all interpretations I of S, the computation of S under I does not halt;
(3) S and S' are *strongly equivalent* if, for all interpretations I, either both of $\text{val}_I(S)$ and $\text{val}_I(S')$ are undefined, or both of $\text{val}_I(\text{S})$ and $\text{val}_I(S')$ are defined and are equal;
(4) S and S' are *weakly equivalent* if, for all interpretations I for which both of $\text{val}_I(S)$ and $\text{val}_I(S')$ are defined, $\text{val}_I(S)$ equals $\text{val}_I(S')$ and
(5) *S contains S'* if, for all interpretations I for which $\text{val}_I(S')$ is defined, $\text{val}_I(S)$ is defined and equals $\text{val}_I(S')$.

Let S and S' be program schemes. We say that

(6) S is *isomorphic to* S' if, for all interpretations I, the sequences of the instructions executed by the computations of S and S' under I are the same.

Definition 1.9[10]. Let ρ be any binary relation on the program schemes or on the monadic recursion schemes such that, for all schemes S and S',

(1) if S and S' are strongly equivalent, then $S\rho S'$; and

(2) if $S\rho S'$, then S and S' are weakly equivalent.

Then, the relation ρ is said to be a *reasonable relation.*

2. Program and recursion schemes. A variety of decidable problems on the single variable program schemes (abbreviated svp schemes) and on the linear monadic recursion schemes (abbreviated lmr schemes) are shown to be *NP*-complete. These problems include strong equivalence, weak equivalence, containment, halting, and divergence. This is accomplished in two steps. First, we show that these problems are *NP*-hard for the svp schemes. Second, we show that these problems are in *NP* for the lmr schemes.

DEFINITION 2.1. A *switching scheme S* is a monadic, loop-free, svp scheme such that each of its statements is either a conditional or a halt statement.

Our first proposition relates the tautology problem for D_3-Boolean forms to the problem of deciding, for a switching scheme S with halt statement labeled B, if the statement labeled by B is executed during some computations of S. All our *NP*-hard lower bounds follow from it.

PROPOSITION 2.2. *There exists a deterministic polynomially time bounded Turing machine M_0 such that M_0, given a D_3-Boolean form f as input, outputs a switching scheme S_f with exactly two halt statements labeled* **A** *and* **B** *such that the statement labelled* **B** *is executed during some computation of S_f, if and only if, f is not a tautology.*

Proof. We illustrate how M_0 constructs S_f from f by an example. Suppose f equals $x_1\bar{x}_2x_4 \vee x_2\bar{x}_3x_4 \vee x_1\bar{x}_4\bar{x}_5$. Then, S_f is the following:

1. **If** $P_1(x)$ **then** 2 **else** 4
2. **If** $P_2(x)$ **then** 4 **else** 3
3. **If** $P_4(x)$ **then A else** 4
4. **If** $P_2(x)$ **then** 5 **else** 7
5. **If** $P_3(x)$ **then** 7 **else** 6
6. **If** $P_4(x)$ **then A else** 7
7. **If** $P_1(x)$ **then** 8 **else B**
8. **If** $P_4(x)$ **then B else** 9
9. **If** $P_5(x)$ **then B else A**

A. Halt.
B. Halt.

We denote the set $\{S_f \mid f$ is a D_3-Boolean form; and the Turing machine M_0 of Proposition 2.2, given input f, outputs $S_f\}$ by $\mathscr{C}$.

DEFINITION 2.3. Let S be an svp scheme with exactly two halt statements labeled **A** and **B**. Let $\mathscr{A}$ and $\mathscr{B}$ be svp schemes. The program scheme $[S, \mathscr{A}, \mathscr{B}]$ is the program scheme that results from S by replacing the statement labeled **A** in S by $\mathscr{A}$ and by replacing the statement labeled **B** in S by $\mathscr{B}$, with a suitable renumbering of the statements in $\mathscr{A}$ and $\mathscr{B}$ as necessary.

For example, let S, $\mathscr{A}$, and $\mathscr{B}$ be the following:

S: 1. **If** $P_1(x)$ **then** 2 **else** 3
2. $x \leftarrow x$
A. Halt.
3. $x \leftarrow x$
B. Halt.

$\mathscr{A}$: 1. $x \leftarrow f(x)$
2. **Halt.**

$\mathscr{B}$: 1. $x \leftarrow g(x)$
2. **Halt**

Then, $[S, \mathscr{A}, \mathscr{B}]$ is the following:

```
1. **If** $P_1(x)$ **then** 2 **else** 3
2  $x \leftarrow x$
A. $x \leftarrow f(x)$
4. **Halt**.
3. $x \leftarrow x$
B. $x \leftarrow g(x)$
5. **Halt**.
```

The next theorem gives general sufficient conditions for a predicate on the svp schemes to be *NP*-hard.

THEOREM 2.4. *Let* Π *be any predicate on the svp schemes for which there exist svp schemes* $\mathscr{A}$ *and* $\mathscr{B}$ *such that, for all schemes S in* $\mathscr{C}$, $\Pi([S, \mathscr{A}, \mathscr{B}])$ *equals False if and only if the statement of S labelled* **B** *is executed during some computation of S. Then, the set* $\{S|S$ *is an svp scheme and* $\Pi(S)$ *equals False*$\}$ *is NP-hard. Moreover, if* $\mathscr{A}$ *and* $\mathscr{B}$ *are loop-free, then the set* $\{S|S$ *is a loop-free svp scheme and* $\Pi(S)$ *equals False*$\}$ *is also NP-hard.*

Proof. By Propositions 1.5 and 1.6, it suffices to show that the set $\mathscr{T}_1$ of nontautological D$_3$-Boolean forms is *p*-reducible to the set $\{S|S$ is an svp scheme and $\Pi(S)$ equals False$\}$. Let f be a D_3-Boolean form. Let S_f be the corresponding element of $\mathscr{C}$. Then, $\Pi([S_f, \mathscr{A}, \mathscr{B}])$ equals False, if and only if, the statement in S_f labeled **B** is executed during some computation of S_f. By Proposition 2.2 this is true, if and only if, f is not a tautology. Since $[S_f, \mathscr{A}, \mathscr{B}]$ is constructible from f by a deterministic polynomially time-bounded Turing machine, the theorem follows. QED

The next two corollaries yield some applications of Theorem 2.4. Henceforth, we denote the svp scheme 1. Halt. by $\mathscr{I}$.

COROLLARY 2.5. *Let* Π *be any of the following predicates on the svp schemes*:

(i) *S diverges*;
(ii) *S halts*;
(iii) *S is strongly equivalent to* $\mathscr{I}$;
(iv) *S contains* $\mathscr{I}$;
(v) $\mathscr{I}$ *contains S*;
(vi) *S is weakly equivalent to* $\mathscr{I}$; *and*
(vii) *for all reasonable relations* ρ *on the svp schemes*, $S\rho\mathscr{I}$.

Then, the set $\{S|S$ *is an svp scheme and* $\Pi(S)$ *equals False*$\}$ *is NP-hard.*

Proof. Each of the predicates in (i) through (vii) satisfies the conditions of Theorem 2.4, where the corresponding schemes $\mathscr{A}$ and $\mathscr{B}$ are as follows:

(i)	$\mathscr{A}$ is 1. Loop.	$\mathscr{B}$ is 1. Halt.
(ii)	$\mathscr{A}$ is 1. Halt.	$\mathscr{B}$ is 1. Loop.
(iii) through (viii)	$\mathscr{A}$ is 1. Halt.	$\mathscr{B}$ is 1. $x \leftarrow f(x)$ 2. Halt.

Q.E.D.

COROLLARY 2.6. *Let* ρ *be any of the following binary relations on the svp schemes: for all svp schemes S and S', SρS', if and only if,*

(i) *S is isomorphic to S'*;
(ii) *S is strongly equivalent to S'*;
(iii) *S contains S'*;
(iv) *S is weakly equivalent to S'*; *and*
(v) *for all reasonable relations* ρ_0 *on the svp schemes*, $S\rho_0S'$.

Then, the set $\{(S, S')|S$ *and S' are svp schemes and* $\sim(S\rho S')\}$ *is NP-hard. Moreover, the set* $\{(S, S')|(S$ *and S' are loop-free svp schemes and* $\sim(S\rho S')\}$ *is also NP-hard.*

Proof. The conclusions of this corollary, for the relations of (ii) through (v), follow easily from Theorem 2.4 and Corollary 2.5. Therefore, we only prove that the conclusions of this corollary hold for isomorphism. As in the proof of Theorem 2.4, we show that the set $\mathcal{T}_1$ of nontautological D_3-Boolean forms is p-reducible to the set $\{(S, S')|S$ and S' are loop-free svp schemes and S is not isomorphic to $S'\}$.

Let f be a D_3-Boolean form. Let S_f be the corresponding element of $\mathcal{C}$. Then, letting $\mathcal{B}_0$ denote the scheme

1. $x \leftarrow g(x)$
2. **Halt**.

the schemes $[S_f, \mathcal{I}, \mathcal{B}_0]$ and $[S_f, \mathcal{I}, \mathcal{I}]$ are isomorphic, if and only if, the statement in S_f labeled **B** is not executed during some computation of S_f. By Proposition 2.2 this is true, if and only if, f is a tautology. Since the schemes $[S_f, \mathcal{I}, \mathcal{B}_0]$ and $[S_f, \mathcal{I}, \mathcal{I}]$ are loop-free and are constructible from f by a deterministic polynomially time-bounded Turing machine, the corollary follows. Q.E.D.

The importance of Theorem 2.4 and Corollaries 2.5 and 2.6 lies in the weakness of the hypotheses needed to show that any predicate satisfying their conditions is NP-hard. Since no looping except possibly loop statements and only monadic functions and predicates are required, their conclusions hold for many other classes of program schemes, e.g. the monadic program schemes with nonintersecting loops, the liberal schemes, and the progressive schemes, see [10], [12]. In the remainder of this section, we show that similar results hold for the lmr schemes and that several of these NP-hard problems are NP-complete.

The effective translation of monadic svp schemes into strongly equivalent lmr schemes in [4] can easily be seen to be executable by a deterministic polynomially time-bounded Turing machine. Thus letting $\mathcal{I}'$ denote the lmr scheme

$$F_1x := \textbf{If } P_1x \textbf{ then } x \textbf{ else } x,$$

one immediate implication of Corollaries 2.5 and 2.6 is the following.

COROLLARY 2.7 (1). *Let* Π *be any of the following predicates on the lmr schemes:*

(i) *S halts*;

(ii) *S diverges*;

(iii) *S is strongly equivalent to $\mathcal{I}'$*;

(iv) *S contains $\mathcal{I}'$*;

(v) *$\mathcal{I}'$ contains S*;

(vi) *S is weakly equivalent to $\mathcal{I}'$; and*

(*vii*) *for all reasonable relations ρ on the lmr schemes, $S\rho\mathcal{I}'$. Then, the set $\{S|S$ is an lmr scheme and $\Pi(S)$ equals False$\}$ is NP-hard.*

(2) *Let ρ be any of the following binary relations on the lmr schemes: for all lmr schemes S and S', $S\rho S'$, if and only if,*

(viii) *S is strongly equivalent to S'*;

(ix) *S contains S'*;

(x) *S is weakly equivalent to S'; and*

(xi) *for all reasonable relations ρ_0 on the lmr schemes, $S\rho_0S'$. Then, the set $\{(S, S')|S$ and S' are lmr schemes and $\sim(S\rho S')\}$ is NP-hard.*

The next two propositions will be used to derive upper bounds on the computational complexity of halting, divergence, strong equivalence, weak equivalence, and containment for the lmr and svp schemes. The first proposition is new. The second closely follows results in [4].

PROPOSITION 2.8. *Let R be an lmr scheme with n defining equations. Then, R diverges for some interpretation if and only if there exists a free interpretation I of R for which the computation of R under I takes at least $2n+1$ steps.*

Proof. The "only if" part is obvious. We show the "if" part. Suppose the computation of R under I takes at least $2n+1$ steps. Then some defining equation, say

$$F_jx := \textbf{If } P_jx \textbf{ then } \alpha_jx \textbf{ else } \beta_jx,$$

must be applied at least three times during it. Hence, the computation of R under I must contain at least two applications of this equation for which the predicate P_j^I takes the same value. Thus letting G_S be the context-free grammar associated with S, there exist strings b_1, b_2, c_1, and c_2 of basis function symbols such that

$$F_1 \overset{*}{\underset{G_S}{\Longrightarrow}} b_1F_jc_1,$$

$$F_j \overset{+}{\underset{G_S}{\Longrightarrow}} b_2F_jc_2,$$

$$P_j^I(c_1x) = P_j^I(c_2c_1x)$$

for the legal derivation for I.

If c_2 equals λ, then the computation of R under I diverges. Otherwise, let I_0 be the free interpretation of R defined by; for all predicate symbols P_i in R;

$$P_i^{I_0}(wx) = \begin{cases} P_i^I(\alpha c_1x), & \text{if } w = \alpha c_2^k c_1 \text{ and } \alpha \text{ is a suffix of } c_2; \\ P_i^I(wx), & \text{otherwise.} \end{cases}$$

Then, the computation of R under I_0 diverges.

PROPOSITION 2.9. *Let R and S be two lmr schemes, with set of basis function symbols $\mathscr{F}$ and set of defined function symbols F such that*

(i) *both of R and S have at most n defining equations*;

(ii) *the length of each string α_i and β_i in a defining equation of R or S is less than m*; *and*

(iii) *each string α_i and β_i in a defining equation of R or S is an element of $\mathscr{F}^* \cdot F \cdot (\mathscr{F} \cup \{\lambda\}) \cup \phi \cup \{\lambda\}$.*

Then, (1) *if there exists an interpretation I' under which R and S differ but for which both of* $\text{val}_{I'}(R)$ *and* $\text{val}_{I'}(S)$ *are defined, then there is a free interpretation I, under which R and S differ and for which both of* $\text{val}_I(R)$ *and* $\text{val}_I(S)$ *are defined, such that the minimum of the lengths of* $\text{val}_I(R)$ *and* $\text{val}_I(S)$ *is less than $3n^3m$. Similarly,* (2) *if there exists an interpretation I' for which* $\text{val}_{I'}(R)$ *is defined and* $\text{val}_{I'}(S)$ *is not, then there is a free interpretation I, for which* $\text{val}_I(R)$ *is defined and* $\text{val}_I(S)$ *is not, such that the length of* $\text{val}_I(R)$ *is also less than $3n^3m$.*

The proofs of (1) and (2) appear on pages 154–157 in [4].

THEOREM 2.10. *The following sets are NP-complete*:

(i) $S_1 = \{S | S$ *is an lmr scheme*; *and S does not halt*$\}$;

(ii) $S_2 = \{S | S$ *is an lmr scheme*; *and S does not diverge*$\}$;

(iii) $S_3 = \{(S, S') | S$ *and S' are lmr schemes*; *and S and S' are not strongly equivalent*$)$;

(iv) $S_4 = \{(S, S') | S$ *and S' are lmr schemes*: *and S and S' are not weakly equivalent*$\}$;

and

(v) $S_5 = \{(S, S') | S$ *and S' are lmr schemes*; *and S does not contain S'*$\}$.

Proof. By Corollary 2.7 each of these sets is *NP*-hard. We illustrate how Propositions 2.8 and 2.9 can be used to show that these sets are in *NP*. We only sketch the proofs for S_1 and S_4. The proofs for S_2, S_3, and S_5 are similar and are left to the reader.

(i) Let M be the nondeterministic Turing machine that operates as follows:

Step 1. M, given input S, checks if S is an lmr scheme. If not, M halts without accepting.

Step 2. M guesses a rightmost derivation Π of the context-free grammar G_S associated with S of the form $F_1 \underset{G_S}{\Longrightarrow} b_{i_1}F_{i_1}c_{i_1} \underset{G_S}{\Longrightarrow} \cdots \underset{G_S}{\Longrightarrow} b_{i_k}F_{i_k}c_{i_k}$, where, letting n_0 be the number of the defining equations of S, $k = 2n_0 + 1$; $F_1, F_{i_1}, \cdots, F_{i_k}$ are defined functions symbols of S; and $b_{i_1}, c_{i_1}, \cdots, b_{i_k}, c_{i_k}$ are strings of basis function symbols of S.

Step 3. M verifies that Π is legal for some free interpretation I of S. If so, M accepts S. Otherwise, M halts without accepting.

By Proposition 2.8, M accepts S_1. Moreover, M is polynomially time-bounded. This follows since

(1) the lengths of each of the sentential forms $F_1, b_{i_1}F_{i_1}c_{i_1}, \cdots, b_{i_k}F_{i_k}c_{i_k}$ in Π is less than $(2n_0 + 1) \cdot (m+1) + 1$, where m is an upper bound on the lengths of the strings α_i, β_i in the defining equations of S; and

(2) Π is legal for some interpretation I, if and only if, for each pair $(b_{i_j}F_{i_j}c_{i_j}, b_{i_{j'}} F_{i_{j'}}c_{i_{j'}})$ of sentential forms in Π, if

$$b_{i_j}F_{i_j}c_{i_j} \underset{G_S}{\Longrightarrow} b_{i_j}\delta c_{i_j},$$

$$b_{i_{j'}}F_{i_{j'}}c_{i_{j'}} \underset{G_S}{\Longrightarrow} b_{i_{j'}}\delta' c_{i_{j'}},$$

$$c_{i_j} = c_{i_{j'}},$$

then $\delta = \delta'$.

Clearly conditions (1) and (2) can be checked deterministically in time bounded by a polynomial in the size of S.

(iv) Let M be the nondeterministic Turing machine that operates as follows:

Step 1. M, given input (S, S') checks if S and S' are lmr schemes. If not, M halts without accepting.

Step 2. M converts S and S' into strongly equivalent lmr schemes S_1 and S'_1, respectively, that satisfy the conditions of Proposition 2.9.

Step 3. M guesses a rightmost derivation Π of G_{S_1}

$$F_1 \underset{G_{S_1}}{\Longrightarrow} \cdots \underset{G_{S_1}}{\Longrightarrow} b_{i_k}F_{i_k}c_{i_k} \underset{G_{S_1}}{\Longrightarrow} b_{i_{k+1}}c_{i_{k+1}}$$

and a rightmost derivation Π' of $G_{S'_1}$

$$F'_1 \underset{G_{S'_1}}{\Longrightarrow} \cdots \underset{G_{S'_1}}{\Longrightarrow} b'_{i_l}F'_{i_l}c'_{i_l} \underset{G_{S'_1}}{\Longrightarrow} b'_{i_{l+1}}c'_{i_{l+1}},$$

where $k+1$ and $l+1$ are less than $3n^3m^3$, and $b_{i_{k+1}}c_{i_{k+1}} \neq b'_{i_{l+1}}c'_{i_{l+1}}$.

[Here, n equals the maximum of the number of defining equations in S and S'; and m equals the maximum of the lengths of the strings α_i and β_i in any of the defining equations of S_1 and S'_1.]

Step 4. M verifies that both of Π and Π' are legal for some interpretation. If so, M accepts (S, S'). Otherwise, M halts without accepting.

By Proposition 2.9 M accepts S_4. Moreover, M is polynomially time-bounded.

This follows by reasoning analogous to that in the proof of (i) and is left to the reader. Q.E.D.

COROLLARY 2.11. *The following sets are NP-complete*:

(i) $\{S|S$ *is an svp scheme*; *and S does not halt*$\}$;

(ii) $\{S|S$ *is an svp scheme*; *and S does not diverge*$\}$;

(iii) $\{(S, S')|S$ *and S' are svp schemes*; *and they are not strongly equivalent*$\}$;

(iv) $\{(S, S')|S$ *and S' are svp schemes*; *and they are not weakly equivalent*$\}$;

(v) $\{(S, S')|S$ *and S' are svp schemes*; *and S does not contain S'*$\}$; *and*

(vi) $\{(S, S')|S$ *and S' are loop-free svp schemes*; *and S and S' are not strongly equivalent*$\}$.

3. A deterministic polynomial time decidable equivalence problem.

3.1. Strongly free schemes. In § 2 we saw that the strong equivalence problem for the svp schemes is NP-complete and thus is likely to be computationally intractable. Here, we inquire if any interesting subclasses of the svp schemes have provably deterministic polynomial time decidable strong equivalence problems. We note that the proof above that strong equivalence for the svp schemes is NP-hard involves sieves of predicates of the type appearing in Figure 3.1 where (a) some predicates, such as P_1, P_2, and P_3 in Figure 3.1, test the same value twice; and (b) the sieve is a directed acyclic graph but not a tree. Schemes with predicates satisfying (a) have the property that not all paths are executable and thus are unlikely to correspond to well-written computer programs. This suggests that we consider svp schemes which have no such predicates.

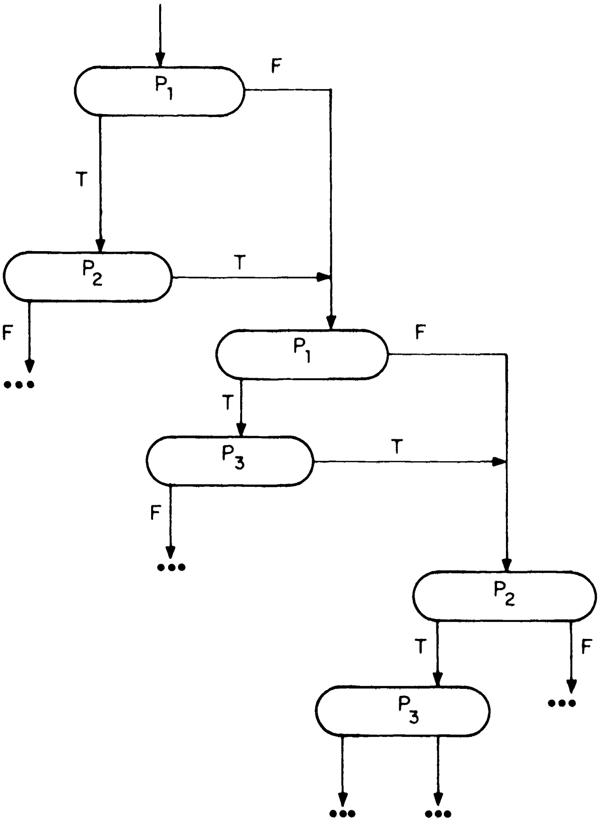


FIG. 3.1

Using the terminology of [4], [10], svp schemes with no predicates satisfying (a) or, equivalently, svp schemes in which all paths are executable are said to be *free*. Thus, we are led to the question—

Q1: "Do free svp schemes have a deterministic polynomial time decidable strong equivalence problem?"

Only a partial answer to question Q1 is presented here. We show that the class of svp schemes in which no two predicates test the same value in a Herbrand interpretation, called the strongly free schemes, has a deterministic polynomial time decidable strong equivalence problem.

In a strongly free svp scheme there is a function application between any two predicates. These schemes behave like deterministic finite automata; and our technique for showing that their strong equivalence problem is decidable deterministically in polynomial time is to consider them as deterministic finite automata (as described below). There is one nontrivial difficulty, however. A strongly free scheme S may have redundant predicates, i.e. predicates whose left and right branches are equivalent. To obtain a deterministic polynomial time strong equivalence test, we must find a deterministic polynomial time redundancy test. This is accomplished by modifying the usual state minimization algorithm for deterministic finite automata.

Before presenting the results of this section, we need some notation. Recall that the value of a scheme S under interpretation H is denoted by val (S, H). We denote the value of a scheme S under interpretation H starting with statement L_0 by val (S, H, L_0). With every svp scheme S, we associate the three languages $L(S)$, $L^c(S)$, and $L^{c\#}(S)$ defined as follows.

Definition 3.2. The *value language* of an svp scheme S, denoted by $L(S)$, is the set {val $(S, H)|H$ is a Herbrand interpretation for which S halts}.

Value languages were extensively used in [4].

Definition 3.3. Let S be an svp scheme. Let H be a Herbrand interpretation. The *computation string* of S under H, denoted by Comp (S, H) is the (possibly infinite) string

$$\cdots \alpha_{m+1}P^{\pm}_{i_m} \cdots P^{\pm}_{i_2}\alpha_2 P^{\pm}_{i_1}\alpha_1$$

such that each α_i is a (possibly empty) string of function symbols of S, $P^{\pm}_{i_j}$ is either $P^{+}_{i_j}$ or $P^{-}_{i_j}$ where P_{i_j} is a predicate symbol of S, and

$$P^{\pm}_{i_j} \text{ is } P^{+}_{i_j}, \text{ if and only if, } (P_{i_j})^H(\alpha_j \cdots \alpha_1) = \text{True}.$$

The *computation language* of S, denoted by $L^c(S)$, is the set {Comp $(S, H)|H$ is a Herbrand interpretation}. The *terminating computation language* of S, denoted by $L^{c\#}(S)$, is set {Comp $(S, H)|H$ is a Herbrand interpretation for which S halts}.

The proof of the following lemma about terminating computation languages is left to the reader.

Lemma 3.4. *For svp schemes S_1 and S_2, if $L^{c\#}(S_1) = L^{c\#}(S_2)$, then*

(i) *S_1 and S_2 halt for the same Herbrand interpretations*; *and*

(ii) *for all Herbrand interpretations for which both S_1 and S_2 halt,* Comp $(S_1, H) =$ Comp (S_2, H).

Recall that an svp scheme S is said to be free if *no* predicate is tested twice with the same argument values under any Herbrand interpretation. This implies that there must be a function application between any two separate occurrences of the same predicate test. We define a similar but stronger notion of freedom.

Definition 3.5. An svp scheme S is said to be *strongly free* if and only if no two predicates test the same value in any Herbrand interpretation.

For strongly free svp schemes, there must be a function application between any two predicate tests.

DEFINITION 3.6. The occurrence of a predicate P in statement L_1 in a scheme S, say

$$L_1. \textbf{ If } P(x) \textbf{ then } L_l \textbf{ else } L$$

is said to be *superfluous*, if and only if, val $(S, H, L_l) =$ val (S, H, L_r) for all Herbrand interpretations H. More generally, two statements L_1 and L_2 in schemes S_1 and S_2, respectively, are said to be *equivalent*, if and only if, val $(S_1, H, L_1) =$ val (S_2, H, L_1) for all Herbrand interpretations H. Finally, a scheme S is said to be *reduced*, if and only if, it contains no superfluous predicate occurrences.

Our first theorem shows how reduced strongly free svp schemes can be characterized by their terminating computation languages. It will be used to show how reduced strongly free svp schemes behave like deterministic finite automata.

THEOREM 3.7. *If S_1 and S_2 are reduced strongly free svp schemes, then $S_1 \equiv S_2$ if and only if $L^{c\#}(S_1) = L^{c\#}(S_2)$.*

Proof. (1) ($\Rightarrow$) Let $S_1 \equiv S_2$. We show that $L^{c\#}(S_1) = L^{c\#}(S_2)$ by proof by contradiction. Suppose $L^{c\#}(S_1) \neq L^{c\#}(S_2)$. Let $x = x_n \cdots x_1$ be a string in one of $L^{c\#}(S_1)$ and $L^{c\#}(S_2)$ but not both, say $x \in L^{c\#}(S_1) - L^{c\#}(S_2)$. Let x^1 be the subsequence of x obtained by deleting all predicate tests. Then, there exists a string $y = y_m \cdots y_1$ in $L^{c\#}(S_2)$ such that

(a) letting y^1 be the subsequence of y obtained by deleting all predicate tests, we have $y^1 = x^1$; and

(b) no other string in $L^{c\#}(S_2)$ satisfies (a) and agrees with x on a longer final segment.

Such a string y exists since $x^1 =$ val (S_1, H) for some Herbrand interpretation for which S_1 halts and $S_1 \equiv S_2$ by assumption.

Let k $(1 \leqq k \leqq \min(n, m))$ be the least positive integer such that $x_k \cdots x_1 \neq y_k \cdots y_1$. Let α be the string that results from $x_{k-1} \cdots x_1$ by deleting all predicate tests (α can be the empty string). Since $x^1 = y^1$ and S_1 and S_2 are strong free svp schemes, both of x_k and y_k must be predicate tests. Suppose the test in x_k is P_i and the test in y_k is P_j. By assumption $P_i \neq P_j$. Let the corresponding statements in S_1 and S_2 be

$$L_{i_0} \cdot \textbf{If } P_i(x) \textbf{ then } L_1 \textbf{ else } L_2 \quad \text{and}$$

$$L_{j_0} \cdot \textbf{If } P_j(x) \textbf{ then } L_1^1 \textbf{ else } L_2^1,$$

respectively. Then L_1 cannot be equivalent to both of L_1^1 and L_2^1, otherwise the occurrence of P_j in statement L_{j_0} is superfluous. So suppose that L_1 and L_1^1 are not equivalent. Then there is a Herbrand interpretation H_0 such that

$$\text{val}\,(S_1, H_0, L_1) \neq \text{val}\,(S_2, H_0, L_1^1).$$

Since S_1 and S_2 are free, we can also choose a Herbrand interpretation H_1 such that

$$\text{Comp}\,(S_1, H_1) = \cdots x_k x_{k-1} \cdots x_1,$$

$$\text{Comp}\,(S_2, H_1) = \cdots y_k y_{k-1} \cdots y_1.$$

Let H_2 be any Herbrand interpretation satisfying the following:

(A) For all proper suffixes α' of α and for all predicate symbols $P_{l'}$

$$(P_l)^{H_2}(\alpha' x) = (P_l)^{H_1}(\alpha' x);$$

(B) $(P_i)^{H_2}(\alpha x) = \text{True}$;

(C) $(P_j)^{H_2}(\alpha x) = \text{True}$; and

(D) For all strings $\alpha' = w'\alpha$ such that α is a proper suffix of α' and for all predicate symbols P_l,

$$(P_l)^{H_2}(\alpha' x) = (P_l)^{H_0}(w' x).$$

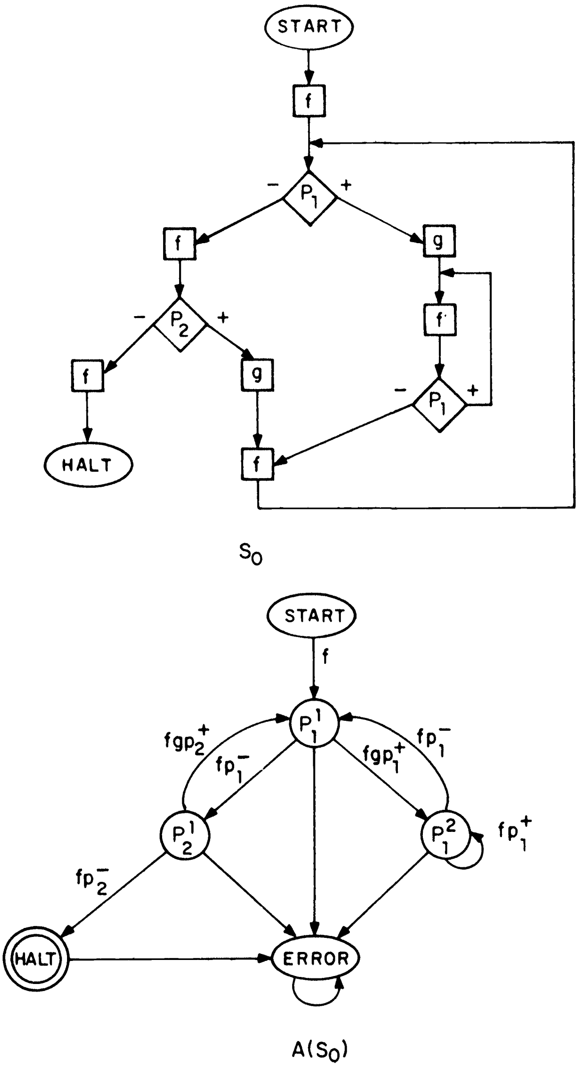
Clearly such Herbrand interpretations exists. For each such Herbrand interpretation H_2,

$$\text{val}\,(S_1, H_2) \neq \text{val}\,(S_2, H_2),$$

contradicting the assumption that S_1 and S_2 are strongly equivalent. (2) ($\Leftarrow$). This follows immediately from Lemma 3.4.

A reduced strongly free svp scheme S can be viewed as a deterministic finite automaton $A(S)$ accepting $L^{c\#}(S)$. To see how this works, consider the reduced strongly free svp scheme S_0 and its associated deterministic finite automaton $A(S_0)$ shown in Fig. 3.8. The alphabet of $A(S_0)$ is

$$\Sigma = \{f, fp_1^-, fgp_1^+, fp_1^+, fp_2^-, fgp_2^+\};$$



S_0

$A(S_0)$

FIG. 3.8

and the state set is

$$K = \{\text{start}, P_1^1, P_2^1, P_1^2, \text{halt}, \text{error}\},$$

where P_i^j is the jth occurrence of predicate P_i (in some arbitrary ordering of occurrences.) Finally in the state diagram of $A(S_0)$, we intend that all unlabeled edges be implicitly labeled by those elements of Σ not occurring as labels on outgoing edges.

Clearly the automaton $A(S_0)$ accepts the language $L^{c\#}(S_0)$. Thus rephrasing Theorem 3.7, for reduced strongly free svp schemes S_1 and S_2, $S_1 \equiv S_2$, if and only if, the associated deterministic finite automata $A(S_1)$ and $A(S_2)$ are equivalent. Noting that, for a strongly free svp scheme S_1, $A(S_1)$ is constructable from S_1 deterministically in polynomial time and that the equivalence problem for deterministic finite automata is decidable deterministically in polynomial time [6], we have the following.

THEOREM 3.9. *The strong equivalence problem for reduced strongly free svp schemes is decidable deterministically in polynomial time.*

3.2. Reducing strongly free Ianov schemes in deterministic polynomial time. In order to extend the equivalence algorithm to arbitrary strongly free Ianov schemes, we give a method of reducing such schemes. We can not simply regard these schemes S as finite automata $A(S)$ and then reduce $A(S)$. The difficulty is illustrated by a simple example which the reader can provide.

In order to decide whether a predicate test is superfluous we need to apply an algorithm similar to the usual finite automaton reduction technique. We search for nonredundancy. When we find it, we attached the predicate value P_i^+ or P_i^- to the edges leading from the state. Then we repeat the algorithm.

Informally the algorithm is the usual Moore type reduction algorithm on $A(S)$ except that if a predicate appears to be superfluous at stage n, that is, both branches lead to states which are equivalent at stage n, then it is treated as superfluous (the predicate label is not used in the equivalence algorithm). Whenever a suspected superfluous predicate turns out to be necessary, then we restore the predicate label and recompute the equivalence relation. This algorithm succeeds because if it is possible to reduce $A(S)$ and assume at every stage that a state is redundant, then it is really redundant (we prove this in Theorem 3.12).

Before we can describe the reduction algorithm we need a number of conventions. First, given scheme S and its associated automaton $A(S)$ we associate with each state the predicate P_i of S corresponding to it. Labels from each state have the form yP_i^+, yP_i^- for $x, y \in (f_i)^*$. To *remove a predicate from a label*, say from xP_i^+ or yP_i^-, means to replace these labels by x or y respectively.

In the reduction algorithm we will consider various sets of labels for the edges of the state diagram. At stage n of the algorithm we will use an alphabet denoted $\Sigma^n := \{a_1^n, \cdots, a_{p_n}^n\}$. For any state in the automaton $A(S)$ associated with a strongly free scheme S, at most two of these labels will apply (will lead to other than an error state). Call these letters 0_s (the predicate is false) and 1_s (the predicate is true). After predicates are removed from labels at a state, there may be only one label remaining. This gives rise to a nondeterministic transition function δ.

As in the Moore type minimization algorithm for finite automata (see [5, 6, 7]), we will group states into *blocks*. The blocks at stage n of the algorithm will be denoted B_i^n.

The algorithm starts with two blocks, $B_1^0 := \{\text{halt state}\}$, $B_2^0 := \{\text{all nonhalt states}\}$, and proceeds to split blocks into smaller blocks until no further splitting is possible. It is possible to split a block B_i^n as long as condition $**$ given below holds:

$**$ $\exists a \in \Sigma^n \exists s_1, s_2 \in B_i^n$ such that
$\delta(s_1, a) \in B_j^n$ and $\delta(s_2, a) \notin B_j^n$; for δ the transition function of $A(S)$.

That is, there are two states in a block which we can recognize as distinct (inequivalent) by one of δ's transition on the label a.

The informal algorithm is this.

REDUCTION ALGORITHM 3.10. Start with $\Sigma, A(S)$. Form Σ^0 as the set of labels with predicates removed and $A^0(S)$ as the automaton with predicates removed from labels (but written on the states). Let B_1^0 contain the half state and B_2^0 all non-halt states. Let N be the *stage* number, initially $N = 0$. Let δ_0 be the (nondeterministic) transition function arising from the δ of $A(S)$.

```
BEGIN REDUCTION ALGORITHM
    initialize (set $N = 0$, set up $B_1^0$, $B_2^0$).
    while ** do
        begin (1) compute the output behavior of each state under $\Sigma^N$ (at stage $N$)
                  using each possible transition of $\delta_N$.
              (2) locate the non-redundant states at stage $N$, i.e. $\delta_N(s, a) \in B_i^N$ and
                  $\delta_N(s, b) \in B_j^N$, $i \neq j$ (possibly $a = b$).
              (3) form a‘new set of labels, $\Sigma^{N+1}$, by restoring the predicates to the
                  labels on the outgoing edges of non-redundant states located in step
                  (2). The new automaton diagram is denoted $A^{N+1}(S)$.
              (4) recompute the output behavior using $\Sigma^{N+1}$, $\delta_{N+1}$.
              (5) split blocks $B_i^N$ to form blocks $B^{N+1}$ by grouping only those states
                  of $B_i^N$ which have the same output behaviour as computed in (4).
        end
    Redundant states are those whose outgoing edges do not have predicates restored
    to their labels.
END REDUCTION ALGORITHM.
```

Given the reduced automaton, say $\hat{A}(S)$, we can construct from it a scheme $\hat{S}$ having no redundant predicates. We remove each redundant state, say

L: **if** P_i **then** L_l **else** L_r

and connect all incoming edges to L_l (that is, replace any **goto** L by **goto** L_l).

Combining this algorithm with the reduction algorithm, we have an algorithm for transforming strongly free Ianov schemes S to reduced strongly free Ianov schemes $\hat{S}$ (we prove this below). The application of Algorithm 3.10 is illustrated in Figs. 3.11a, 3.11b, and 3.11c.

Analysis of runtime. It is easy to see that the Reduction Algorithm is in the worst case bounded above by $O(|\Sigma| \cdot |K|^2)$. Consider the time for each step, the bounds are

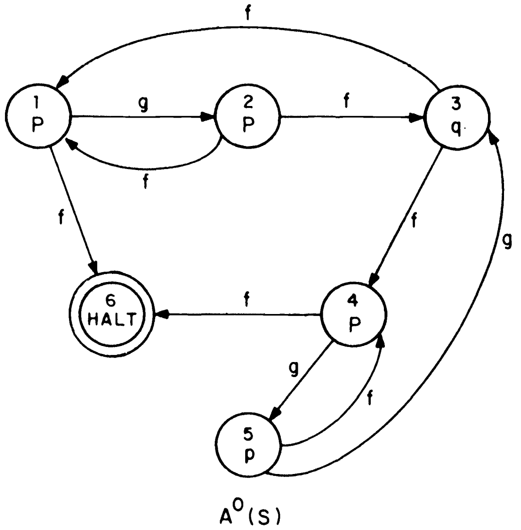
$$(1) \leqq |\Sigma| \cdot |K| \quad (2) \leqq |K| \quad (3) \leqq |\Sigma| \quad (4) \leqq |\Sigma| \cdot |K| \quad (5) \leqq |K|.$$

So the worst case occurs when at most one state is split off of a block on each iteration. Thus the worst case is

$$O(|K| \cdot [2 \cdot |\Sigma| \cdot |K| + 2 \cdot |K| + |\Sigma|]).$$

If we use a more efficient algorithm, such as Hopcroft [6] (also see Gries [5]), then the time is $O(|\Sigma| \cdot |K| \cdot \log(|K|))$.

In any case this is a polynomial time bounded algorithm in either $|K|$, $|\Sigma|$ or in $|S|$. We now summarize our knowledge in a theorem.



B_1^0	B_2^0
{6}	{1, 2, 3, 4, 5}

The B_i^1 blocks are for i = 1, 2, 3, 4, 5:

FIG. 3.11A

B_1^1	B_2^1	B_3^1	B_4^1	B_5^1
{6}	{1}	{2, 3}	{4}	{5}

FIG. 3.11B

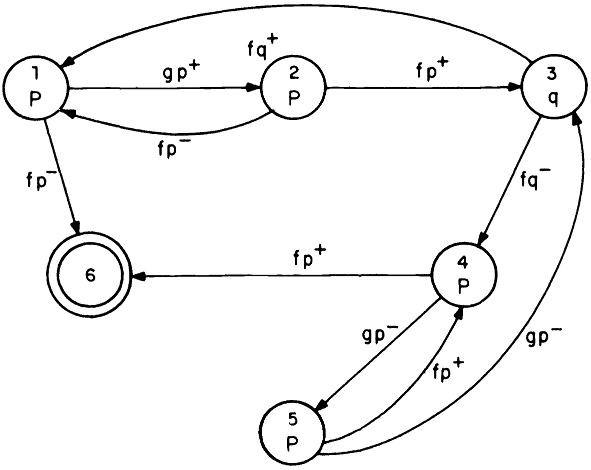


FIG. 3.11C

3.2.1. Correctness of the algorithms.

THEOREM 3.12. *There is an algorithm whose runtime is no more than a polynomial in $|S|$ which produces the reduced scheme $\hat{S}$ given S. That is,*

(i) *$S \equiv \hat{S}$ and*

(ii) *S contains no redundant predicates.*

Proof. The time analysis given above shows that the algorithm is polynomial in $|S|$. We need only show (i) and (ii). We first consider (i).

(1) Clearly if a predicate P_i remains in S then it is not redundant because the algorithm produces an interpretation under which the true and false branches from P_i are distinct. So we need only show that if a predicate occurrence is removed, say at state s as

$$s\colon \textbf{if } P_i \textbf{ then } L_l \textbf{ else } L_f,$$

then that occurrence is really redundant. To prove this, suppose some predicate occurrences were erroneously removed, say P_{i_1} at state $s_{i_1}, \cdots, P_{i_n}$ at states s_{i_n}. Then order these by the length of the interpretation under which the true and false branches are distinct. Suppose P_i is one with the least length interpretation. Then that interpretation cannot involve another predicate erroneously removed in an essential way. That is, either the two computations, the true one which is $x_n x_{n-1} \cdots x_1$ or the false one, $y_m y_{m1} \cdots y_1$ either (a) do not contain any P_{i_j} (erroneously removed predicates) or (b) if such a P_{i_j} does occur, then the true branch from it to the halt state (x_n or y_m) is the same as the false branch, because otherwise this P_{i_j} would have a shorter interpretation showing it to be nonredundant than P_i does, contradicting the definition of P_i. Thus in either case (a) or (b), the computations $y_m \cdots y_1$ and $x_n \cdots x_1$ appear already in some $A^k(S)$. That is, neither computation requires the presence of an erroneously classified predicate. Therefore, P_i would be discovered to be nonredundant at some state k of the reduction algorithm.

(2) Finally, to show $S \equiv \hat{S}$ we notice that S and $\hat{S}$ are nearly isomorphic. For every state s of S there is a corresponding state $\hat{s}$ of $\hat{S}$ unless s is redundant. But if s is redundant, then we know that the edges in S which by-pass s do not change equivalence. The reader can prove this by carefully considering these "near isomorphisms" under any Herbrand interpretation H.

We now state a fact about finite automata.

Fact 3.13. *There is an $O(|\Sigma| \cdot n \log(n))$ time algorithm to decide the equivalence of finite automata S_1, S_2 over Σ where $n = \max(|K_1|, |K_2|)$, K_1, K_2 the state sets of S_1, S_2.*

Using this we have the theorem we need.

THEOREM 3.14. *There is a polynomial time bounded algorithm to decide the equivalence of strongly free Ianov schemes.*

3.2.2. Extension to predicate clusters. We now want to mention an extension of the reduction and equivalence algorithms from strongly free Ianov schemes to strongly free Ianov schemes with tree-like predicate clusters substituted for predicates. The idea is to replace any tree-like cluster of predicates by a single multi-exit predicate.

Let S be a Ianov scheme, then a *cluster of predicates* in S is a loop free subscheme of S containing no function applications and such that no edge can be extended without including a function application. A *tree-like cluster* is such that the cluster is a tree whose nodes are predicates.

Notice that in a free scheme no predicate can occur more than once on a path from root to leaf in a cluster, but predicates may indeed occur more than once.

We represent these clusters by multi-exit predicates and can make this assignment of multi-exit predicates to clusters uniform if we choose a specified ordering of predicates. For example, suppose we have P, Q, R, T. We then label all outputs in the order P, Q, R, T.

To decide equivalence of free Ianov schemes S1, S2 we convert the predicate clusters to multi-exit predicates and then convert the result to a finite automaton, $A(S)$, with labels on predicates given in a standard order. Even in the case of free Ianov schemes, the generation of multi-exit predicates may require exponential time.

If all the predicate clusters in a Ianov scheme are tree-like, then the multi-exit predicate has the same number of exits as there are leaves in the tree, thus it can be generated in polynomial time (in the number of edges) given the cluster.

We can use essentially the same type of reduction algorithm as before, but we must be careful to say exactly when a predicate in a cluster is redundant on the basis of information gathered about the multi-exit predicate in $A(S)$.

During the reduction algorithm, the edges leaving a multi-exit predicate can be grouped together into edge-groups, $E_i^N(s)$; that is a stage N there may be $i = 1, \cdots, m$ edge-groups associated with state s. We say that a predicate occurrence Q in a cluster C *is redundant with respect to the edge-groups* E_i^N if for all sequences of predicate tests z_1 such that $z_1Q^+y \in E_i^N$ there is a sequence z_2 compatible with z_1 (no predicate P appears as P^+ in z_1 and P^- in z_2 or vice versa) such that $z_2Q^-y \in E_i^N$. That is, Q does not affect the decisions made by predicates tested after Q. For example, let the edge-groups be labeled A, B, C and consider the tree-like predicate cluster and the equivalent multi-exit predicate in Figs. 3.15a and 3.15b. The sequences in the edge-groups are

A	B	C
P^+Q^+	P^+Q^-	P^-Q^+
		P^-Q^-

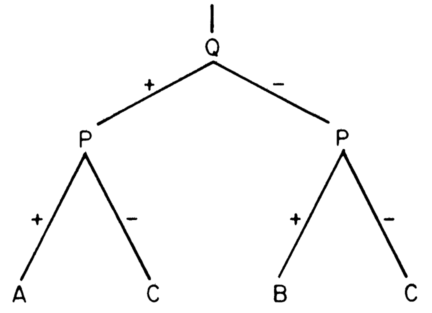


FIG. 3.15A

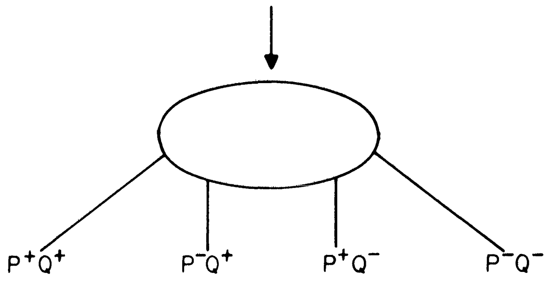


FIG. 3.15B

Thus, the predicate Q is redundant with respect to edge-group C. (Therefore in the reduction algorithm the labels P^-Q^+ and P^-Q^- are replaced by P^-.)

This example suggests how inefficient a predicate cluster might be. But we do not need to consider methods of finding the minimum cluster equivalent to a given cluster in order to obtain a polynomial equivalence algorithm. We only need a method of eliminating the redundant predicates from the labels on outgoing edges of multi-exit predicates.

This example is too simple to illustrate the difficulties in testing for redundant predicate occurrences. It is not sufficient to see whether xQ^+y and xQ^-y both appear in an edge group. For example, consider the tree-like predicate cluster in Figure 3.16. Then in edge-group B we have P^-Q^+y, $S^-R^+Q^-y$, R^-Q^-y. So Q is redundant for B because both S^-R^+ and R^- are compatible with P^-.

In order to mimic the reduction algorithm for strongly free schemes, we need a procedure to check for redundancy in predicate clusters given as assignment of edge-groups (this assignment comes from the main algorithm).

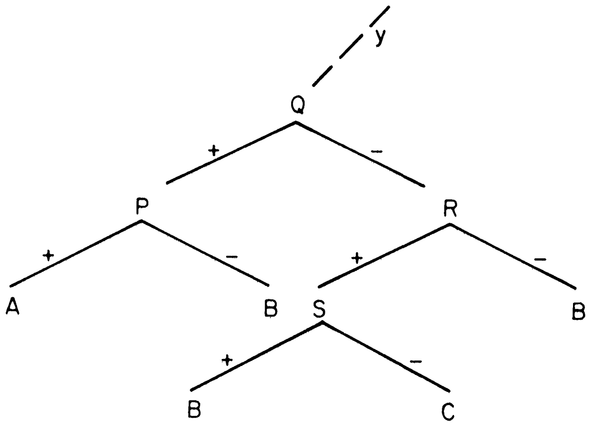


FIG. 3.16

3.2.3. Multi-exit nonredundancy procedure. Given predicate cluster C and edge-groups $E_1, \cdots, E_m$, to test whether a predicate occurrence Q in a label on an edge in E_i^N is redundant, do the following:

```
begin
        (1) locate $Q$ in the cluster (let $y$ be the path to $Q$).
        (2) list all prefixes of the form $z$ where $zQ^+y$ is in $E_i^N$.
        (3) for each $z$ in (2) check whether there is a prefix $w$
            where (a) $wQ^-y$ is in $E_i^N$
                  (b) $w$ and $z$ are compatible.
        (4) if there is a $w$ for each $z$, then $Q$ is redundant, otherwise it is not and the
            predicate is output.
end
```

The reader can now check the validity of the following claims.

PROPOSITION 3.17. *A predicate occurrence Q in tree-like cluster C is nonredundant with respect to edge group E_i if the multi-exit redundancy procedure generates Q given C and E_i.*

It is also easy to check that this procedure runs in polynomial time in the number of predicates in the cluster.

PROPOSITION 3.18. *If tree-like predicate cluster C has n predicates, then the multi-exit redundancy procedure runs in at most n^2 steps.*

4. Simple programs. We conclude by showing that the equivalence problems for several very elementary programming languages are also *NP*-complete. First, we consider the Loop 1 languages in [12], [14].

DEFINITION 4.1. A *loop program* is a finite sequence of instructions of the five types: (1) Do x, (2) End, (3) $x \leftarrow 0$, (4) $x \leftarrow y$, and (5) $x \leftarrow x+1$.

A subset of the variables used in a loop program is designated as the *input variables* of the program. One variable is designated as the *output variable* of the program. Loop programs compute functions of their input variables. We say that two loop programs are *equivalent* if they compute the same function.

DEFINITION 4.2. For all $i = 0, 1, 2 \cdots, L_i$ is the class of all loop programs in which the maximum level of nesting of Do statements equals i. The set *Inequiv* (L_i) is the set of all pairs of inequivalent L_l programs.

PROPOSITION 4.3. Inequiv (L_1) *is NP-complete.*

Proof. To show that Inequiv (L_1) is *NP*-hard, it suffices to show that $\mathcal{T}_2$, the set of satisfiable C$_3$-Boolean forms, is p-reducible to it. We show how, for each C$_3$-Boolean form f, to efficiently construct an L_1 program Π_f such that, for all assignments of initial values to its input variables, the value of its output variable upon termination equals 0, if and only if, f is not satisfiable.

Let $f = c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i = c_{i1} \vee c_{i2} \vee c_{i3}$ and each c_{ij} is a literal. Let the propositional variables of f be $x_1, \cdots, x_n$. Then, the L_1 program Π_f is constructed as follows. The input variables of Π_f are $x_1, \cdots, x_n$; and the output variable of Π_f is z. The program Π_f has the form—$(A_1, A_2, \cdots, A_n, B_1, \cdots B_k, D_1, \cdots, D_k, z \leftarrow 0, z \leftarrow z+1,$ Do C'_1, $z \leftarrow 0$, End, $\cdots$ Do, C_k $z \leftarrow 0$, End), where A_j, B_i, and D_m are program blocks defined as follows:

(1) A_j computes the value of $\bar{x}_j$, the negation of x_j

```
$A_j$ is $\bar{x}_j \leftarrow 0$
      $\bar{x}_j \leftarrow \bar{x}_j + 1$
      Do $x_j$
      $\bar{x}_j \leftarrow 0$
      End.
```

(2) B_i computes the value of the clause c_i, for any given values of $x_1, \cdots, x_n, \bar{x}_1, \cdots, \bar{x}_n$. We illustrate the construction of B_i by an example. Suppose c_i is $x_1 \vee \bar{x}_2 \vee \bar{x}_3$, then

```
$B_i$ is $c_1 \leftarrow 0$
      Do $x_1$
      $c_1 \leftarrow 0$
      $c_1 \leftarrow c_1 + 1$
      End
      Do $\bar{x}_2$
      $c_1 \leftarrow 0$
      $c_1 \leftarrow c_1 + 1$
      End
      Do $x_3$
      $c_1 \leftarrow 0$
      $c_1 \leftarrow c_1 + 1$
      End.
```

(3) D_m computes the value of $\bar{c}_m$, the negation of c_m.

```
$D_m$ is $\bar{c}_m \leftarrow 0$
      $\bar{c}_m \leftarrow \bar{c}_m + 1$
      Do $c_m$
      $\bar{c}_m \leftarrow 0$
      End.
```

We leave it to the reader to verify that the program Π_f outputs a 1 for some assignment of values to its input variables, if and only if, f is satisfiable. Otherwise, Π_f always outputs 0.

Finally, the fact that Inequiv (L_1) is in NP follows immediately from Theorems 4 and 7 in [14]. Q.E.D.

We note that the equivalence problem for L_0 programs can be solved deterministically in linear time, and that the equivalence problem for L_2 programs is undecidable [12].

DEFINITION 4.4. Let x and y be nonnegative integers. Then,

$$x \dot{-} y = \begin{cases} x-y, & \text{if } x \geqq y, \\ 0, & \text{otherwise.} \end{cases}$$

DEFINITION 4.5. $\mathcal{P}_1$ is the class of all programs consisting of a finite sequence of instructions of the form (1) $x_i \leftarrow x_j + x_k$, and (2) $x_j \leftarrow 1 \dot{-} x_j$, where x_j and x_k may be nonnegative integer constants. $\mathcal{P}_2$ is the class of all programs consisting of a finite sequence of instructions of the forms $x_i \leftarrow x_j \dot{-} x_k$, where x_j and x_k may be non-negative integer constants.

$\mathcal{P}_1$ and $\mathcal{P}_2$ programs compute functions in the obvious manner. We say that two $\mathcal{P}_1$ or $\mathcal{P}_2$ programs are *equivalent* if they compute the same function. *Inequiv* ($\mathcal{P}$) (*Inequiv* ($\mathcal{P}_2$)) is the set of all pairs of inequivalent $\mathcal{P}_1(\mathcal{P}_2)$ programs.

PROPOSITION 4.6. Inequiv ($\mathcal{P}_1$) *and* Inequiv ($\mathcal{P}_2$) *are NP-complete.*

Proof. (1) To show that Inequiv ($\mathcal{P}_1$) is NP-hard, it suffices to show that $\mathcal{T}_2$ is p-reducible to it. This is accomplished by simulating the construction in the proof of Proposition 4.3.

Let $f = c_1 \wedge c_2 \wedge \cdots \wedge c_k$, where $c_i = c_{i1} \vee c_{i2} \vee c_{i3}$ and each c_{ij} is a literal. Let the propositional variables of f be $x_1, \cdots, x_n$. The $\mathcal{P}_1$ program Π_f has n input variables $x_1, \cdots, x_n$, output variable $\mathcal{P}$, and has the form—($A_1, \cdots, A_n, B_1, \cdots, B_k, D_1, \cdots, D_k, P \leftarrow 1 \dot{-} \bar{c}_1, \cdots, P \leftarrow 1 \dot{-} \bar{c}_k$). Here, A_1, B_j and D_m are—

(a) A_i is $\bar{x}_i \leftarrow 1 \dot{-} x_i$,

(b) letting $c_j = x_1 \vee \bar{x}_2 \vee x_3$, B_j is $c_j \leftarrow x_1 + \bar{x}_2$, $c_j \leftarrow c_j + x_3$, and

(c) D_m is $\bar{c}_m \leftarrow 1 \dot{-} c_m$

To show that Inequiv ($\mathcal{P}_1$) is in NP, it suffices to note that there is a deterministic polynomially time-bounded Turing machine M such that M, given a $\mathcal{P}_1$ program Π as input, outputs an equivalent L_1 program Π'. Thus, since Inequiv (L_1) is in NP, so is Inequiv ($\mathcal{P}_1$).

M operates as follows. It replaces all instruction of the form $x_i \leftarrow 1 \dot{-} x_j$ in Π by the program fragment

```
$x_i \leftarrow 0$
$x_i \leftarrow x_i + 1$
Loop $x_j$
$x_i \leftarrow 0$
End
```

It replaces all instructions of the form $x_i \leftarrow x_j + x_k$ in Π by the L_1 program fragment $x_i \leftarrow x_j$

```
Loop $x_k$
$x_i \leftarrow x_i + 1$
End.
```

(2) The proof that Inequiv ($\mathcal{P}_2$) is NP-complete is similar and will not be presented here. Details can be found in [2].

REFERENCES

[1] E. ASHCROFT, Z. MANNA AND A. PNEULI, *Decidable properties of monadic functional schemes*, J. Assoc. Comput. Mach., 20 (1973), pp. 489–499.

[2] R. L. CONSTABLE, H. B. HUNT III, AND S. SAHNI, *On the computational complexity of scheme equivalence*, TR-74-201, Department of Computer Science, Cornell University, Ithaca, NY, 1974.

[3] S. A. COOK, *The complexity of theorem proving procedures*, Proceedings Third Annual ACM Symposium on Theory of Computing, May 1971, pp. 151–158.

[4] S. J. GARLAND AND D. C. LUCKHAM, *Program schemes, recursion schemes, and formal languages*, J. Comput. System Sci. 7 (1973), pp. 119–160.

[5] D. GRIES, *Describing an algorithm by Hopcroft*, Acta Informat. 2 (1973), pp. 97–109.

[6] J. E. HOPCROFT, *An nlogn algorithm for minimizing states in a finite automaton*, Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, 1971, pp. 189–196.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[8] I. IANOV, *On logical algorithm schemata*, Cybernetics Problems I, 1958.

[9] D. M. KAPLAN, *Regular expressions and the equivalence of programs*, JCSS, 4 (1969), pp. 361–386.

[10] D. C. LUCKHAM, D. M. R. PARK, AND M. S. PATERSON, *On formalized computer programs*, Ibid., 4 (1969), pp. 220–249.

[11] Z. MANNA, *Program schemas*, Currents in the Theory of Computing, A. V. Aho, ed., Prentice-Hall, Englewood Cliffs, NJ, 1973, pp. 90–142.

[12] A. R. MEYER AND D. RITCHIE, *Computational complexity and program structure*, IBM research paper, May 15, 1967.

[13] M. PATERSON, *Equivalence problems in a model of computation*, MIT A.I. Laboratory Technical Memo. No. 1, Massachusetts Institute of Technology, Cambridge, Ma, 1970.

[14] D. TSICHRITZIS, *The equivalence problem of simple programs*, J. Assoc. Comput. Mach., 17 (1970), pp. 729–738.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

SAVING SPACE IN FAST STRING-MATCHING*

ZVI GALIL† AND JOEL SEIFERAS‡

Abstract. The string-matching problem is to find all instances (as contiguous substrings) of a "pattern" character string x in a longer "text" string y. The naive algorithm, trying the pattern from scratch starting at each successive text position, requires only a fixed number of auxiliary storage locations but time proportional to $|x|\cdot|y|$ (worst case). On the other hand, the fast algorithm of Knuth, Morris, and Pratt requires only time proportional to $|y|$ but extra space proportional to $|x|$ (every case). Algorithms described in this paper reduce the extra *space* used by the Knuth–Morris–Pratt algorithm down to $O(\log |x|)$, and the *time* for the naive algorithm down to $O(|x|^{\varepsilon}|y|)$ for any fixed $\varepsilon > 0$. Also described are implementations on two-way multihead finite automata and multitape Turing machines.

Key words. string-matching, pattern-matching, text-searching, multihead finite automaton, Turing machine, time-space trade-off, counter simulation, string periodicity, failure function

Introduction. The inspiration for this paper was an attempt to implement the fast string-matching algorithm of Knuth, Morris, and Pratt [13] as a FORTRAN subroutine. Although a FORTRAN subroutine can use a variable-length array if it receives the array and its length as arguments, every local storage location must be allocated when the subroutine is compiled; i.e., there is no provision for "dynamic storage allocation" during execution. On the other hand, the Knuth–Morris–Pratt algorithm uses a number of local storage locations which grows with the size of the input to the algorithm. This rules out any straightforward implementation without limiting the generality (at least in principle) of the algorithm, and it motivates our search for fast string-matching algorithms which use less space.

The string-matching problem is to find all instances of a "pattern" character string x as a subword (contiguous substring) in a "text" string y. The classical "naive" algorithm (trying the pattern from scratch starting at each successive text position) requires time proportional to $|x|\cdot|y|$ in the worst case, while the "fast" algorithm of Knuth, Morris, and Pratt requires time proportional to only $|x|+|y|$ in the worst case. (We use $|w|$ to denote the length of the character string w.) On the other hand, the naive algorithm requires only a fixed number of additional memory locations, while the fast algorithm requires about $|x|$ additional memory locations in every case. (G. Barth's space-saving variant [2] of the Knuth–Morris–Pratt algorithm sometimes uses fewer additional memory locations, but only when no full instance of the pattern occurs in the text.) It is the latter fact that makes a general straightforward implementation of the fast algorithm impossible without dynamic storage allocation. (A "straightforward implementation" would neither change the contents of the $|x|+|y|$ memory locations initially containing x and y nor store more than $O(\log (|x|+|y|))$ bits (enough for an arbitrary reference into x or y) in a single memory location.) Our results improve each algorithm in its weak suit.

* Received by the editors August 30, 1978, and in revised form June 14, 1979.

† Department of Mathematical Sciences, Tel-Aviv University, Tel-Aviv, Israel. Part of this research was done while the author was visiting the Computer Sciences Department at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. The work of this author was supported in part by the Israel Commission for Basic Research.

‡ Department of Computer Science, University of Rochester, Rochester, New York 14627. Part of this research was done while the author was at The Pennsylvania State University, University Park, Pennsylvania, and part was done while he was visiting the M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts. The work of this author was supported in part by the National Science Foundation under Grants MCS 77-06613 and MCS 77-19754.

(1) We show how to reduce the additional space utilization by the fast algorithm down to $O(\log |x|)$ memory locations. Although theoretically this does still require some dynamic storage allocation in the worst case, a mere seventy additional memory locations suffice to accommodate every pattern up to a billion characters long and many (in fact, most) patterns longer than that. As in the Knuth–Morris–Pratt algorithm, the text need not be "backed up" or stored at all (e.g., it could be read sequentially from a card reader). Alternatively, if the text *is* buffered, the algorithm can read the text and detect the pattern instances in real time. (Galil observed that the Knuth–Morris–Pratt algorithm has this property [9].) On a multitape Turing machine with three input heads, this linear-time algorithm can be implemented in space $O((\log |x|)^2)$.

(2) We show how to reduce the running time of the naive algorithm all the way down to $O(|x|^\varepsilon(|x|+|y|))$ for any fixed $\varepsilon>0$. Thus we get an *almost* linear-time algorithm which can be implemented without any dynamic storage allocation at all. Like the slow naive algorithm, this faster algorithm can be implemented as a FORTRAN subroutine which receives the text only sequentially, say from a card reader. Also like the naive algorithm, in fact, this faster one can be implemented on a two-way multihead finite automaton with only a single, one-way head available for reading the text. (The number of heads used to read the pattern is proportional to $1/\varepsilon$.) On a *single*-worktape Turing machine with three input heads, this almost linear-time algorithm can be implemented in space $O(\log |x|)$.

The algorithms referred to in paragraphs (1) and (2) above are the extremes in a time-space-trade-off hierarchy of algorithms. The hierarchy is obtained from the Knuth–Morris–Pratt algorithm by leaving varying subsets of data unrecorded, essentially to be recomputed on demand. Although this is a familiar general idea (e.g., [3], [12]) the time penalty can be kept unusually small for the right choice of unrecorded data in the Knuth–Morris–Pratt algorithm.

The techniques and results described in this paper can be used to save space in various related algorithms as well. For example, theorems similar to ours below hold for palindrome recognition [10] and for string-matching with "forced mismatches" [17].

Preliminaries. For $1\leqq i\leqq |z|$, let $z(i)$ denote the *i*th character of the character string z. For $0\leqq i\leqq j\leqq |z|$, let $[i, j]_z$ denote that string v for which there are strings u and w such that $z=uvw$, $i=|u|$, and $j=|uv|$; i.e., $[i,j]_z=z(i+1)\cdots z(j)$.

The nonnull string u is a *period* of v if v is a prefix of u^k for some k. Equivalently, the nonnull string u is a period of v if and only if v is a prefix of uv [13].

PERIODICITY LEMMA [15], [13]. *If z has periods of lengths p and q, and $|z|\geqq p+q$, then z has a period of length $\gcd(p, q)$, the greatest common divisor of p and q.*

The basic algorithms. Consider a pattern x and a text y. For $1\leqq q\leqq |x|$, define

$$\mathrm{KMP}(q)=\min\{p>0 \mid [p,q]_x \text{ is a prefix of } [0,q]_x\}.$$

Note that KMP is nondecreasing and that $\mathrm{KMP}(q)\leqq q$ is the length of the shortest period of $[0,q]_x$. (Some authors [13], [6], [1] define only the related function $q-\mathrm{KMP}(q)$ explicitly. Aho, Hopcroft, and Ullman [1] call this the "failure function" for the pattern x.)

In our algorithms we will use p for the next prospective starting position of the pattern in the text. The algorithms check out these prospective positions in increasing order, so p can only increase. We use q for the length of the known match between the pattern and the text starting at position p; so $x(i)=y(p+i)$ for $1\leqq i\leqq q$ (i.e., $[0,q]_x=$

$[p, p+q]_y$). Briefly, then,

p = position of pattern in text,
q = length of known match starting at that position.

In these terms, the basic Knuth–Morris–Pratt algorithm is

```
$(p, q)$ := $(0, 0)$;
loop:
  while $x(q+1) = y(p+q+1)$ do $q := q+1$;
  if $q = 0 \rightarrow (p, q) := (p+1, 0)$
    □ $q > 0 \rightarrow (p, q) := (p + \mathrm{KMP}(q), q - \mathrm{KMP}(q))$
  fi;
  go to loop
```

(We use Dijkstra's **if–fi** alternative construct [5]. The symbol □ is used to separate the enclosed set of "guarded commands." Each guarded command consists of a "guard" (a boolean expression), an arrow, and a "guarded list" (a sequence of statements separated by semicolons). Each time the construct is reached in execution, a guarded command whose guard is true is selected, and the corresponding guarded list is executed.) Of course the algorithm stops when $p+q+1 > |y|$. To detect matches, just watch for $q = |x|$. So that the algorithm can continue after a match, let $x(|x|+1)$ be the unmatchable character \$. The total running time is $O(|y|+|x|)$ because the quantity $2p+q$ keeps increasing toward its bound of $2|y|+(|x|+1)$.

The naive algorithm differs in that the function KMP is not used when a mismatch occurs; instead, (p, q) is always set to $(p+1, 0)$. For each integer $k \geqq 1$, we can design a hybrid algorithm as follows:

```
$(p, q)$ := $(0, 0)$;
loop:
  while $x(q+1) = y(p+q+1)$ do $q := q+1$;
  if $q = 0 \rightarrow (p, q) := (p+1, 0)$
    □ ($q > 0$ and $\mathrm{KMP}(q) \leqq q/k$) $\rightarrow$
      $(p, q) := (p + \mathrm{KMP}(q), q - \mathrm{KMP}(q))$
    □ ($q > 0$ and $\mathrm{KMP}(q) > q/k$) $\rightarrow$
      $(p, q) := (p + \lceil q/k \rceil, 0)$
  fi;
  go to loop
```

(Each connective **and** above is actually the "conditional and"; its expression is considered false if the left conjunct is false, even if the right conjunct is undefined.) To see that this algorithm overlooks no prospective text position p for the pattern, just note that it never increments p by more than KMP(q). Note that we get the pure Knuth–Morris–Pratt algorithm for $k = 1$ and the pure naive algorithm for $k = |x|+1$.

The running time for the hybrid algorithm is $O(k|y|+|x|)$ because the quantity $(k+1)p+q$ keeps increasing. To compensate for the worse time bound for $k > 1$, the algorithm can get by with limited knowledge of the function KMP; the exact value is needed only when $\mathrm{KMP}(q) \leqq q/k$. We show below how this allows us to get by with only $O(\log_k |x|) = O((\log |x|)/(\log k))$ additional memory locations, provided $k \geqq 3$. (By a more careful analysis of essentially the same algorithm, we can extend this result down to $k \geqq 2$ and relax the requirement that k be an integer. See Appendix A for details. On the other hand, in a first reading of this and the next two sections, the reader can safely restrict attention to the specific case $k = 3$.)

Remark. In incrementing p, our algorithms take into account the match of length q, but they do not take into account the mismatch $x(q+1) \neq y(p+q+1)$. This is useful if we sometimes want variations of the algorithms to "force a mismatch" (increment p for some reason other than $x(q+1) \neq y(p+q+1)$). To find all a string's prefixes of the form ww, for example, we could match the string against itself, "forcing a mismatch" when such a prefix is found ($q = p$). For more examples and a formalization of string-matching with forced mismatches, see [17].

For string-matching without forced mismatches, on the other hand, Knuth, Morris and Pratt point out that taking the mismatch $x(q+1) \neq y(p+q+1)$ into account can save time. In our hybrid algorithm, taking the mismatch into account can also save space, by further limiting the necessary knowledge of KMP; more specifically, the exact value of KMP(q) becomes unnecessary unless $\mathrm{KMP}(q+1) \neq \mathrm{KMP}(q)$, provided $k \geqq 2$. To see this, consider the task of incrementing p when q is positive and $\mathrm{KMP}(q+1) = \mathrm{KMP}(q) \leqq q/k$. Since $x(q+1) = x(q+1-\mathrm{KMP}(q+1)) = x(q+1-2\cdot\mathrm{KMP}(q+1)) = \cdots$, incrementing p by any multiple of $\mathrm{KMP}(q) = \mathrm{KMP}(q+1)$ will lead to the same mismatch; therefore, p can be incremented by the length of some other larger period of $[0, q]_x$. It follows by the periodicity lemma that p can be incremented by at least $\lceil q - \mathrm{KMP}(q) \rceil \geqq \lceil q - q/k \rceil \geqq \lceil q/k \rceil$, provided $k \geqq 2$. This leads to the following revision of our basic hybrid algorithm:

```
(p, q) := (0, 0);
loop:
  while $x(q+1) = y(p+q+1)$ do $q := q+1$;
  if $q = 0 \rightarrow (p, q) := (p+1, 0)$
    □ ($q > 0$ and $\mathrm{KMP}(q+1) \neq \mathrm{KMP}(q)$ and $\mathrm{KMP}(q) \leqq q/k$) →
       $(p, q) := (p + \mathrm{KMP}(q), q - \mathrm{KMP}(q))$
    □ else → $(p, q) := (p + \lceil q/k \rceil, 0)$
  fi;
  go to loop
```

Below we refine the algorithm which does *not* take into account the mismatch $x(q+1) \neq y(p+q+1)$. See Appendix B for the corresponding refinement of the algorithm which does take the mismatch into account

The k-truncation and its properties.

DEFINITION. The *k-truncation* of the function KMP is the partial function KMP_k defined by

$$\mathrm{KMP}_k(q) = \begin{cases} \mathrm{KMP}(q) & \text{if } \mathrm{KMP}(q) \leqq q/k, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

(The k-truncation of KMP is just that part of KMP required for our basic hybrid algorithm, described above.)

Since KMP is nondecreasing, the k-truncation of KMP is nondecreasing on its domain of definition. Let $\mathrm{VAL}(1) < \mathrm{VAL}(2) < \cdots < \mathrm{VAL}(l)$ be the sequence of distinct values it assumes. Note that if $\mathrm{KMP}_k(q_1) = \mathrm{KMP}_k(q_2) = \mathrm{VAL}(r)$, then $\mathrm{KMP}_k(q) = \mathrm{VAL}(r)$ for every q between q_1 and q_2. For each r ($1 \leqq r \leqq l$), take

$$\mathrm{GATE}(2r-1) = \min\{q \mid \mathrm{KMP}_k(q) = \mathrm{VAL}(r)\},$$

$$\mathrm{GATE}(2r) = 1 + \max\{q \mid \mathrm{KMP}_k(q) = \mathrm{VAL}(r)\}.$$

Example.

pattern:	a	a	a	b	a	a	a	b	a	a	a	b	a	a	a	a	a	b	a	a	a	b	
position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
KMP:	1	1	1	1	4	4	4	4	4	4	4	4	4	4	4	4	13	14	14	14	14	14	14
KMP_3:	–	–	–	1	–	–	–	–	–	–	–	–	4	4	4	4	–	–	–	–	–	–	–

$l=2$, $\text{VAL}(1)=1$, $\text{VAL}(2)=4$

$\text{GATE}(1)=3$, $\text{GATE}(2)=4$, $\text{GATE}(3)=12$, $\text{GATE}(4)=16$.

Lemma 1 below summarizes the major consequence for KMP of the periodicity lemma. Lemma 2 and Corollaries 1–3 describe properties of KMP_k motivated by and to be cited in the following section.

LEMMA 1. *If* $\text{KMP}(q_2)>\text{KMP}(q_1)$, *then* $\text{KMP}(q_2)>q_1-\text{KMP}(q_1)$.

Proof. Suppose $\text{KMP}(q_2)>\text{KMP}(q_1)$ (hence $q_2>q_1$) but

$$\text{KMP}(q_1)+\text{KMP}(q_2)\leqq q_1<q_2.$$

Then $[0,q_1]_x$ has periods of both lengths $\text{KMP}(q_1)$ and $\text{KMP}(q_2)$, hence also a period of length $\gcd(\text{KMP}(q_1),\text{KMP}(q_2))$, by the periodicity lemma. Since $\text{KMP}(q_2)\leqq q_1$, the period $[0,\text{KMP}(q_2)]_x$ of $[0,q_2]_x$, and hence $[0,q_2]_x$ itself, also has a period of length $\gcd(\text{KMP}(q_1),\text{KMP}(q_2))$. Since $\text{KMP}(q_2)$ is the length of the *shortest* period of $[0,q_2]_x$,

$$\text{KMP}(q_2)\leqq\gcd(\text{KMP}(q_1),\text{KMP}(q_2))\leqq\text{KMP}(q_1)<\text{KMP}(q_2),$$

a contradiction. □

LEMMA 2. *If* $k\geqq 2$, *then* $\text{GATE}(2r-1)=k\cdot\text{VAL}(r)$ *for every* r $(1\leqq r\leqq l)$. (*We use only the fact that* $\text{VAL}(r)=\lfloor\text{GATE}(2r-1)/k\rfloor$, *which will hold even when* k *is not an integer* (*see Appendix* A).)

Proof. We always have $\text{KMP}(k\cdot\text{VAL}(r))\leqq\text{VAL}(r)$. (There must be some q with $\text{VAL}(r)=\text{KMP}_k(q)\leqq q/k$. But then $[0,k\cdot\text{VAL}(r)]_x$ is a prefix of $[0,q]_x$ and inherits a period of length $\text{KMP}(q)=\text{VAL}(r)$.) Below, using the hypothesis $k\geqq 2$, we rule out strict inequality. It follows that

$$k\cdot\text{VAL}(r)=\min\{q\mid\text{KMP}(q)=\text{VAL}(r)\text{ and }\text{KMP}(q)\leqq q/k\}=\text{GATE}(2r-1).$$

It remains only to rule out $\text{VAL}(r)>\text{KMP}(k\cdot\text{VAL}(r))$. By Lemma 1, this inequality would imply

$$\begin{aligned}\text{VAL}(r)&>k\cdot\text{VAL}(r)-\text{KMP}(k\cdot\text{VAL}(r))\\&\geqq k\cdot\text{VAL}(r)-\text{VAL}(r).\end{aligned}$$

For $k\geqq 2$, this would yield the contradiction $\text{VAL}(r)>\text{VAL}(r)$. □

LEMMA 3. *If* $\text{KMP}(q_2)>\text{KMP}(q_1)$ *and* $\text{KMP}(q_1)\leqq q_1/k$, *then* $\text{KMP}(q_2)>\lceil(k-1)q_1/k\rceil$.

Proof. Assume the hypothesis. By Lemma 1,

$$\begin{aligned}\text{KMP}(q_2)&\geqq q_1-\text{KMP}(q_1)+1\\&\geqq q_1-q_1/k+1\\&=(k-1)q_1/k+1\\&>\lceil(k-1)q_1/k\rceil.\quad\square\end{aligned}$$

COROLLARY 1. *If* $k > 2$, *then*

$$\mathrm{VAL}(r+1)/\mathrm{VAL}(r) > k-1 > 1$$

for every r $(1 \leqq r \leqq l-1)$.

COROLLARY 2. *If* $k \geqq 3$, *then*

$$\mathrm{GATE}(2r) < \mathrm{GATE}(2r+1)$$

for every r $(1 \leqq r \leqq l-1)$.

Proof. Take

$$q_1 = \mathrm{GATE}(2r) - 1,$$
$$q_2 = \mathrm{GATE}(2r+1).$$

By Lemma 3,

$$\begin{aligned}(k-1)q_1/k &< \mathrm{KMP}(q_2)\\ &\leqq q_2/k,\end{aligned}$$

so

$$\begin{aligned} q_2 &> (k-1)q_1\\ &= q_1 + (k-2)q_1\\ &\geqq q_1 + (k-2)k\\ &\geqq q_1 + 1 \quad \text{if } k \geqq 3\\ &\quad ((k-2)k \geqq 1, \text{provided } k \geqq 1+\sqrt{2} \approx 2.4). \quad \square \end{aligned}$$

COROLLARY 3. *If* $k \geqq 3$ *and* $\mathrm{KMP}(q) = \mathrm{KMP}_k(q) = \mathrm{VAL}(r)$, *then*

$$\mathrm{GATE}(2r-2) \leqq q - \mathrm{KMP}(q) < q < \mathrm{GATE}(2r).$$

Proof. Assume the hypothesis. Only the leftmost inequality is not immediate. By Lemma 3,

$$q \geqq k \cdot \mathrm{KMP}(q) > (k-1)(\mathrm{GATE}(2r-2)-1),$$

so

$$\begin{aligned}\mathrm{GATE}(2r-2) - 1 &< q/(k-1)\\ &\leqq q - q/k \quad (\text{since } k \geqq 3)\\ &\leqq q - \mathrm{KMP}_k(q). \quad \square\end{aligned}$$

Saving space in linear time. Rather than store a full table of the k-truncation of KMP, our hybrid algorithm can simply store the VAL and GATE sequences in $3l$ memory locations. In fact, only the GATE sequence ($2l$ memory locations) has to be stored explicitly if $k \geqq 2$; because then $\mathrm{VAL}(r) = \lfloor \mathrm{GATE}(2r-1)/k \rfloor$ for every r $(1 \leqq r \leqq l)$, by Lemma 2 above. For $k > 2$, it follows from Corollary 1 above that $l < \log_{k-1} |x| = O((\log |x|)/(\log k))$; so we assume in this section that k does exceed 2. (The inequality $l < \log_{k-1} |x|$ is strict because $\mathrm{VAL}(l) = \lfloor \mathrm{GATE}(2l-1)/k \rfloor \leqq \lfloor |x|/k \rfloor < |x|/(k-1)$.)

To get by with the proposed compact representation of the k-truncation of KMP, as q changes the algorithm will have to keep track of for which r $(0 \leqq r \leqq 2l)$ $\mathrm{GATE}(r) \leqq q < \mathrm{GATE}(r+1)$. (We adopt the conventions that $\mathrm{GATE}(0) = 0$ and $\mathrm{GATE}(2l+1) =$

$|x|+2$.) The possible changes are $q := 0$, $q := q+1$, and (when KMP(q)$\leqq q/k$)$q := q -$ KMP(q). By Corollaries 2 and 3 above, the second and third of these changes can require at most one increment $r := r+1$ and at most one decrement $r := r-1$, respectively. Therefore, keeping track requires only constant time for each change.

Shown below is a refinement of our hybrid algorithm along the lines suggested above. We use r for $\max\{r' \mid \text{GATE}(r') \leqq q\}$.

```
(p, q, r) := (0, 0, 0);
loop:
  while x(q+1) = y(p+q+1) do
    begin
      q := q+1;
      while q ≧ GATE(r+1) do r := r+1; [at most once]
      if q = |x| then declare match
    end;
  if q = 0 → (p, q, r) := (p+1, 0, 0)
    □ (q > 0 and r is odd) →
      (p, q) := (p + ⌊GATE(r)/k⌋, q − ⌊GATE(r)/k⌋);
      while q < GATE(r) do r := r−1 [at most once]
    □ (q > 0 and r is even) →
      (p, q, r) := (p + ⌈q/k⌉, 0, 0)
  fi;
  go to loop
```

Note that we can avoid some redundant tests above by changing the **while**-loops

while $q \geqq \text{GATE}(r+1)$ **do** $r := r+1$,
while $q < \text{GATE}(r)$ **do** $r := r-1$

to the respective **if**-statements

if $q \geqq GATE(r+1)$ **then** $r := r+1$,
if $q < \text{GATE}(r)$ **then** $r := r-1$.

Below, however, we neglect these optimizations, because they do not apply to the generalizations described in Appendix A.

Since p cannot decrease, our algorithm can indicate for certain *the first time* the scanned text position $p+q$ reaches i whether a pattern instance ends at text position i; i.e., the algorithm is *on-line*. Moreover, although $p+q$ sometimes decreases, we can modify the algorithm so that the text itself never has to be rescanned. To do this, we let a matching portion of the *pattern* temporarily play the role of the rescanned text. To see how this is possible, consider any point at which a new text symbol $y(p+q+1)$ is first read. Since p cannot decrease, the only portion of the text which might be rescanned before the next new text symbol is read is $[p, p+q+1]_y$. The single symbol $y(p+q+1)$ can be stored in a single additional memory location ("next" below). The rest, $[p, p+q]_y$ matches the pattern prefix $[0, q]_x$; so it will be accessible in the pattern if the current value ("p_0" below) of p is saved for index adjustment. In this way, we get an algorithm which reads the text on-line without "backing up" and which does not store more than a single character of it at a time. The algorithm, restructured to make the *read*-loop explicit, is shown below. We use i for $\max\{i' \mid y(i') \text{ has been read}\}$.

```
$(p, q, r) := (0, 0, 0)$;
$i := 0$;
loop:
  $i := i+1$; $[i = p+q+1]$
  **read** next; $[\text{next} = y(p+q+1)]$
  $p_0 := p$;
  **until** $p+q = i$ **do**
    **if** (($p+q+1 = i$ **and** $x(q+1) = \text{next}$) **or**
          ($p+q+1 < i$ **and** $x(q+1) = x(p+q+1-p_0)$))
      **then**
        **begin**
          $q := q+1$;
          **while** $q \geqq \text{GATE}(r+1)$ **do** $r := r+1$
        **end**
      **else**
        **if** $q = 0 \rightarrow (p, q, r) := (p+1, 0, 0)$
          □ ($q > 0$ **and** $r$ is odd) $\rightarrow$
            $(p, q) := (p + \lfloor \text{GATE}(r)/k \rfloor, q - \lfloor \text{GATE}(r)/k \rfloor)$;
            **while** $q < \text{GATE}(r)$ **do** $r := r-1$
          □ ($q > 0$ **and** $r$ is even) $\rightarrow$
            $(p, q, r) := (p + \lceil q/k \rceil, 0, 0)$
        **fi**;
  **if** $q = |x|$ **then** declare match;
  **go to** loop
```

It is interesting to note that the on-line algorithm above satisfies Galil's predictability condition [9]. This condition requires that the time spent between **read**'s be at most proportional to a constant plus the further postponement of the soonest conceivable next completed match. Of course the further postponent is just one less than the increase in p. The time spent is proportional to the increase in the quantity $(k+1)p+q$. Since $p+q$ increases by only 1, however, this means that the time spent is at most proportional to a constant plus the increase in kp (and hence in p). It follows [9] that the algorithm can be made to run in real time (bounded delay between **read**'s) if special memory locations are available for text buffering. Moreover, the real-time version would still use only $O(\log_k |x|)$ memory locations in addition to those reserved for x and y.

It remains only to show how initially to set up the GATE sequence. As in the original Knuth–Morris–Pratt algorithm [13], the preprocessing can be done by matching the pattern against itself, starting with $(p, q) = (1, 0)$. Of course no complete instance of the pattern will be found, but for each i we will have $\text{KMP}(i) = p$ the first time $p+q = i$ holds. To see this, consider any i $(1 \leqq i \leqq |x|)$. The first time $p+q = i$ holds, no symbol beyond $x(i)$ has been examined, so $\text{KMP}(i)$ is still a prospective position for the pattern; i.e., $p \leqq \text{KMP}(i)$. Since the algorithm guarantees that $[p, i]_x$ is a prefix of $[0, i]_x$ at this point, we cannot have $p < \text{KMP}(i)$; so $p = \text{KMP}(i)$, as claimed. Note that $\text{KMP}(i)$ cannot possibly be needed by the algorithm until later, when $q = i$ holds.

When $\text{KMP}(i) = p$ is discovered, our algorithm should consider adding i to its GATE sequence. Suppose the last element in the sequence so far is $\text{GATE}(R)$. Then i should be added to the sequence as $\text{GATE}(R+1)$ if and (by Corollary 2) only if either R is even and $\text{KMP}(i) \leqq i/k$ or R is odd and $\text{KMP}(i) > i/k$.

The preprocessing algorithm is shown explicitly below. As above, we use i for $\max\{i' | x(i') \text{ has been read}\} = \min\{i' | p+q = i' \text{ has not yet held}\}$. We use R for

$\max\{r' \mid \text{GATE}(r') \leqq i-1\}$.

```
GATE(0) := 0;
$(p, q, r) := (1, 0, 0)$;
$i$ := 1; **read** $x(i)$; $R$ := 0;
loop:
  $i := i+1$; **read** $x(i)$; $[i = p+q+1]$
  **if** $x(i) = \$$ **then**
    **begin**
      **if** $R$ is even
        **then** $[R = 2l]$ GATE$(R+1) := i+1$
        **else** $[R = 2l-1]$ (GATE$(R+1)$, GATE$(R+2)$) := $(i, i+1)$;
      **return**
    **end**;
  **until** $p+q = i$ **do**
    **if** $x(q+1) = x(p+q+1)$
      **then**
        **begin**
          $q := q+1$;
          **while** $(r+1 \leqq R$ **and** $q \geqq \text{GATE}(r+1))$ **do** $r := r+1$
        **end**
      **else**
        **if** $q = 0 \rightarrow (p, q, r) := (p+1, 0, 0)$
          □ $(q > 0$ **and** $r$ is odd$) \rightarrow$
            $(p, q) := (p + \lfloor \text{GATE}(r)/k \rfloor, q - \lfloor \text{GATE}(r)/k \rfloor)$;
            **while** $q < \text{GATE}(r)$ **do** $r := r-1$
          □ $(q > 0$ **and** $r$ is even$) \rightarrow$
            $(p, q, r) := (p + \lceil q/k \rceil, 0, 0)$
        **fi**;
  **if** (($p \leqq i/k$ **and** $R$ is even) **or**
    ($p > i/k$ **and** $R$ is odd)) **then**
    **begin**
      $R := R+1$;
      GATE$(R) := i$
    **end**;
  **go to** loop
```

Including preprocessing, our hybrid algorithm for fixed $k > 2$ performs string-matching in linear time $(O(k|x|+|x|+k|y|+|x|) = O(k(|x|+|y|)) = O(|x|+|y|))$ and only logarithmic additional space (about $10 + 2 \cdot \log_{k-1}|x| = O(\log|x|)$ memory locations, counting those used for p, q, r, etc.). This is an exponential reduction in the additional space used by Knuth, Morris, and Pratt for linear-time string-matching. In Theorem 1 we summarize these results.

THEOREM 1. *For each $k > 2$, about*[1] $10 + 2 \cdot \log_{k-1}|x|$ *local memory locations are enough for an algorithm to recognize the end of every instance of a pattern string* x (*passed*

[1] The exact number depends on how we count. For example the range of the variables r and R is sufficiently small that we might store them together in the same location. In addition, there are questions of whether locations are needed for constants such as 0 and for temporary results such as q/k and i/k. All this is further confused by the fact that the finite program itself would typically be stored together with the local variables. More precise automata-theoretic statements will be possible for the implementations on multihead finite automata and Turing machines which follow.

as an argument in a randomly accessible read-only array of length $|x|$) *as a subword in a longer on-line text string* y *in time proportional to* $k|y|$. *Each text character is read only once, after the algorithm has decided whether the pattern is a suffix of the preceding text prefix. Only the text character most recently read has to be stored, but the delay between successive* **read**'s *can be bounded by a constant proportional to* k *if text buffering is allowed* (*i.e., if the text is stored in a read-only array which can be read again*).

Although taking k larger above does improve space efficiency (at the expense of time efficiency), there is no *practical* reason to consider any k larger than, say, 3. With only about 30 local variables, for example, the $k=3$ algorithm can accommodate any pattern up to $2^{10}=1024$ characters long; with 70 local variables, it can handle any pattern up to a billion characters long. Even longer patterns can be handled within the same space if l for those patterns happens to be small enough.

(Only an exceedingly rare pattern does not have l *very* small. To see this, suppose $l \geqq l_0 \geqq 1$. By Corollary 1 above, $\mathrm{VAL}(l_0) > m$ for $m=(k-1)^{l_0-1}$. Therefore, any pattern with $l \geqq l_0$ has a prefix $[0, k\cdot \mathrm{VAL}(l_0)]_x$ with a period of length

$$\begin{aligned}\mathrm{KMP}(k\cdot\mathrm{VAL}(l_0)) &= \mathrm{KMP}(\mathrm{GATE}(2l_0-1)) \quad \text{(by Lemma 2)}\\ &= \mathrm{VAL}(l_0)\\ &> m.\end{aligned}$$

For each $i>m$, however, only 2^i of the 2^{ki} binary strings of length ki have a period of length i. Of the 2^n binary patterns of length $n \geqq k(m+1)$, therefore, the number with $l \geqq l_0$ is at most

$$\begin{aligned}\sum_{i \geqq m+1} 2^{n-ki}\cdot 2^i = 2^n \cdot \sum_{i\geqq m+1} 2^{i(1-k)} &= 2^n(2^{k-1}-1)^{-1}/2^{m(k-1)}\\ &= 2^n(2^{k-1}-1)^{-1}/2^{(k-1)^{l_0}};\end{aligned}$$

so the *frequency* of such patterns is at most $(2^{k-1}-1)^{-1}/2^{(k-1)^{l_0}}$, regardless of n. If $k=3$, for example, the frequency of such patterns is at most $(1/3)/2^{2^{l_0}}$, and the *average* value of l is no more than

$$(1/3)\sum_{l_0 \geqq 1} l_0/2^{2^{l_0}} < 1/2.$$

(A particular consequence of the calculation above is that the frequency of binary patterns with l *nonzero* (i.e. KMP_k nonempty) is no more than one in every $2^{k-1}(2^{k-1}-1)$. In other words, if we simply replace the assignment $p := p+1$ by $p := p + \lceil q/k \rceil$ in the naive string-matching algorithm, we get a linear-time algorithm which correctly performs string-matching (in any text) for all but one out of every $2^{k-1}(2^{k-1}-1)$ binary patterns of any given length.)

Saving more space. In principle, of course, the linear-time algorithms of Theorem 1 still do require dynamic storage allocation to perform completely general string-matching. We do not know whether completely general string-matching of this sort is possible in linear time without dynamic storage allocation, but it is possible to further reduce and even eliminate the dynamic storage allocation without retreating all the way to the slow ($O(|x|\cdot|y|)$) naive algorithm. To see how, note that the parameter k in our hybrid algorithm can be a *computed function* of the pattern length. If the function is easily computable, its calculation will not affect our time and space analyses. As we increase the function from a constant up to identically the pattern length, we will get a heirarchy of algorithms of increasing time complexity ($O(k(|x|+|y|))$) but decreasing space complexity ($O(\log_k |x|) = O((\log |x|)/(\log k))$). More significant than this time-space trade-off, however, is the effect of choosing $k \approx |x|^\varepsilon$ for a fixed small rational $\varepsilon > 0$.

THEOREM 2. *For each small rational $\varepsilon > 0$, about $10+2/\varepsilon$ local memory locations are enough for an algorithm to recognize the end of every instance of a pattern string x (passed as an argument in a randomly accessible read-only array of length $|x|$) as a subword in a longer on-line text string y in time proportional to $|x|^\varepsilon |y|$. Each text character is read only once, after the algorithm has decided whether the pattern is a suffix of the preceding text prefix. Only the text character most recently read has to be stored.*

Proof. Assume $\varepsilon \leqq 1$ and $|x| \geqq 2$. The algorithm computes

$$k := 1+2^{\lceil \varepsilon \cdot \lceil \log_2 |x| \rceil \rceil} \geqq 3$$

and then calls the algorithm of Theorem 1 for this value of k. (This is possible because our descriptions actually did use k only in a variable's role.) Since $\varepsilon \leqq 1$,

$$|x|^\varepsilon \leqq k-1 < 2^{1+\varepsilon}|x|^\varepsilon \leqq 4|x|^\varepsilon.$$

Therefore,

$$\begin{aligned} 10+2 \cdot \log_{k-1} |x| &= 10+2(\log |x|)/(\log (k-1)) \\ &\leqq 10+2(\log |x|)/(\log |x|^\varepsilon) \\ &= 10+2/\varepsilon \end{aligned}$$

and

$$k|y| \leqq (4|x|^\varepsilon+1)|y| = O(|x|^\varepsilon |y|). \quad \square$$

Theorem 2 finally makes general, fast string-matching possible without any dynamic storage allocation at all (e.g., in FORTRAN). About 30 local variables ($\varepsilon = 1/10$) suffice for completely general string-matching in time $O(|x|^{1/10}|y|)$, and about 70 local variables ($\varepsilon = 1/30$) reduce the time to $O(|x|^{1/30}|y|)$. Not only are these times very close to linear, but for patterns up to 1024 and a billion characters long, respectively, k evaluates to just 3. For all practical purposes, therefore, these algorithms are identical to the linear-time $k=3$ algorithm of Theorem 1, prefaced by the calculation of k as a theoretician's hedge.

Implementations on multihead finite automata. The simplicty of the naive string-matching algorithm is that it can be implemented on a multihead finite automaton. Only two two-way heads have to be maintained: one at position $p+q+1$ of the text, and one at position $q+1$ of the pattern. Alternatively, if "backing up" the text is to be avoided, a single one-way text head and three two-way pattern heads suffice. The algorithms of Theorem 2 have a similar simple structure, but the number of heads grows in proportion to $1/\varepsilon$.

THEOREM 3. *For each small rational $\varepsilon > 0$, a two-way $\lfloor 10+2/\varepsilon \rfloor$-head finite automaton can recognize the end of every instance of a pattern string x as a subword in a longer on-line text string y (accessible to only one of the heads) in time proportional to $|x|^\varepsilon |y|$. The text head shifts right only, and it does so only after the automaton has decided whether the pattern is a suffix of the text prefix scanned so far. (The multihead finite automata for this theorem can sense which heads are coincident.)*

Proof. This result is apparently stronger than Theorem 2. (Theorem 2 could be derived as an easy corollary of Theorem 3, but probably not vice versa.) For the proof, we adapt the algorithms designed for the earlier result.

Recall that there are three steps in the algorithms for Theorem 2: calculate k, preprocess the pattern, and search the text. To calculate k we implement the following algorithm:

```
$k := |x|$;
$m := 0$;
while $k > 1$ do
  begin
    $k := \lceil k/2 \rceil$;
    $m := m + \varepsilon$
  end;
$m := \lceil m \rceil$;
while $m > 0$ do
  begin
    $k := 2k$;
    $m := m - 1$
  end;
$k := k + 1$
```

We can maintain k and $\lfloor m \rfloor$ as head positions in the pattern and commit $m - \lfloor m \rfloor$ to finite-state memory. One auxiliary head enables the halving or doubling of k in time proportional to k. For $\varepsilon \leqq 1$, it follows that the entire calculation requires only time proportional to $|x|$.

The essential variables and quantities of the earlier preprocessing and searching algorithms will be maintained as head positions in the adaptations. First consider the preprocessing algorithm. Heads will be maintained at the following pattern positions:

k (calculated earlier),
i,
$q+1$,
$p+q+1$,
p,
$\lfloor i/k \rfloor$,
GATE(1), GATE(2), $\cdots$, GATE($2l$)
(at most $2l \leqq 2 \cdot \log_{k-1} |x| \leqq 2/\varepsilon$ positions).

Until assignment, the heads for GATE(1), GATE(2), $\cdots$, GATE($2l$) will advance with the head at position i. The positions GATE(0) and GATE($2l+1$) -1 are marked by the endmarkers around x, and the values of r and R are maintained in finite-state memory. Note that none of the maintained head positions need ever exceed $|x|+1$.

So that each head position can be updated in time proportional to a constant plus the largest mandated change, three auxiliary heads will be used. The first will count modulo k to enable $\lfloor i/k \rfloor$ to be updated in a single step when i increases. The second will be used to count modulo k for the calculation of $\lfloor \text{GATE}(r)/k \rfloor$ or $\lceil q/k \rceil$ from a copy of GATE(r) or from q, respectively, when either of these increments is needed. The third auxiliary head will be used for the necessary copy of GATE(r) when the increment $\lfloor \text{GATE}(r)/k \rfloor$ is needed. Thus the preprocessing algorithm accounts for all $\lfloor 9 + 2/\varepsilon \rfloor$ heads on the pattern.

For the searching algorithm, three fewer pattern heads will be needed. Of course the one text head will be maintained at position i of the text. Pattern heads will be maintained at the following positions:

k (calculated earlier),
GATE(1), GATE(2), $\cdots$, GATE($2l$) (calculated earlier),
$i-p$,
$q+1$,
$p+q+1-p_0$.

So that each head position can be updated in time proportional to a constant plus the largest mandated change, two auxiliary heads will be available to help in calculating $\lfloor \text{GATE}(r)/k \rfloor$ or $\lceil q/k \rceil$ when either of these increments is required. Note again that none of the maintained head positions need ever exceed $|x|+1$.

To adapt the time analysis, we should consider each test and each assignment in the preprocessing and searching algorithms. Each test can be performed in constant time by comparing scanned symbols or by checking for head coincidence or proximity; for example, $p+q=i$ holds in the searching algorithm if and only if $q+1$ exceeds $i-p$ by exactly 1. By design, the time for each assignment is proportional to the largest mandated change in head position, and only the following assignments in the two algorithms can mandate more than a constant change in the position of any head:

$$(p, q) := (p + \lfloor \text{GATE}(r)/k \rfloor, q - \lfloor \text{GATE}(r)/k \rfloor),$$
$$(p, q) := (p + \lceil q/k \rceil, 0),$$
$$p_0 := p.$$

Recall that the earlier time analysis was based on the proportional increase with time of the quantity $(k+1)p+q$. In each of the first two assignments, the quantity still does increase in proportion to the largest mandated change in any head position and hence in proportion to the largest mandated change in any head position and hence in proportion to time. The last assignment mandates a change only in $p+q+1-p_0$, a decrease of $p-p_0$. Since p never decreases, the text length is a bound on the sum of all such changes. Therefore, the *total* time spent on the last assignment is only $O(|y|)$. □

Remarks. (i) Only five of the heads above ever have to shift left. Three of these are used in the calculation of k. Since a one-way head can maintain the position k after it is found, all five two-way heads are available again for the preprocessing algorithm; they are used at positions $q+1$ and $p+q+1$ and as the three auxiliary heads. Finally, the two-way heads are used in the searching algorithm at positions $q+1$, $p+q+1-p_0$, and $i-p$ and as the two auxiliary heads. Since the searching algorithm requires three fewer heads, it does not matter that the one-way heads used at positions i, p, and $\lfloor i/k \rfloor$ in the preprocessing algorithm are no longer available. We conjecture that *some* left shifting is unavoidable; in fact, we conjecture that *no* one-way multihead finite automaton, no matter how slow, can perform string-matching. There are techniques available for proving apparently similar inadequacies of one-way multihead finite automata [16], [18].

(ii) Suppose the algorithm stores $\lfloor \text{GATE}(r)/k \rfloor$ instead of GATE(r) for each odd r, as in the appendix. When it is needed, then, the increment $\lfloor \text{GATE}(r)/k \rfloor$ can be looked up rather than calculated; and this saves one of the auxiliary (two-way) pattern heads in the multihead finite automaton implementation.

(iii) Equally fast string-matching can be performed by multihead finite automata which *cannot* sense which heads are coincident. For each pair of heads whose coincidence is relevant to the algorithms of Theorem 3, we need only maintain an additional head at a position equal to their separation; then coincidence will be indicated when the additional head scans the left endmarker. Not every head has to be compared with every other, and it turns out that the number of heads is still $O(1/\varepsilon)$.

Implementations on Turing machines. M. Fischer and M. Paterson [6] showed how to implement the Knuth–Morris–Pratt algorithm in linear time on a multitape Turing machine, so it is natural to ask how efficiently our algorithms can be implemented on Turing machines. Since the number of bits stored in each memory location by our algorithms might be proportional to $\log|x|$, we will probably have to settle for space proportional to at least $(\log|x|)^2$ for a linear-time algorithm, and space proportional to at least $\log|x|$ for a nearly linear-time algorithm. Even so, input access remains a problem. (With only one input head, in fact, the space-time product to recognize even $\{x\$x|x\in\{0, 1\}^*\}$ cannot be $o(|x|^2)$ [4].) If we allow our multitape Turing machines to have a one-way text input head and *two* two-way pattern input heads, however, we *can* get the desired implementations.

THEOREM 4. *For each $\varepsilon>0$, a single-worktape Turing machine can recognize the end of every instance of a pattern string x (accessible to two two-way input heads) as a subword in a longer on-line text string y (accessible to just one one-way input head) in time proportional to $|x|^{\varepsilon}|y|$ and space proportional to $\log|x|$. The text input head shifts right only, and it does so only after the Turing machine has decided whether the pattern is a suffix of the text prefix read so far.*

Proof. To get these implementations, we adapt the multihead finite automata of Remark (iii) following the proof of Theorem 3. After the easy calculation of k, only two of the pattern heads, at positions $q+1$ and $p+q+1$ during preprocessing and at positions $q+1$ and $p+q+1-p_0$ during searching, ever have to read any symbol except the left endmarker; all the rest are serving only as counters which can be incremented by 1, decremented by 1, and tested for 0. By Lemma 4 below, these counters can be maintained by a single head on a single Turing machine worktape in time proportional to the number of increments and decrements and in space proportional to the logarithm of their largest contents $|x|+1$. □

LEMMA 4. *Consider counters which can be incremented by* 1, *decremented by* 1, *and tested for* 0.

(a) *A single-tape Turing machine can simulate any fixed number of such counters in linear time and in space proportional to the logarithm of their largest contents.*

(b) *A single-tape Turing machine can simulate* one *such counter in* real *time and in space proportional to the logarithm of its largest contents.*

Proof. (a) This simulation is described in detail by P. Fischer, A. Meyer, and A. Rosenberg [8, pp. 276–277], but they do not point out the space bound. For each counter, the simulation maintains separate binary counts of increments and decrements, taking care to cancel corresponding 1's in the two binary representations. The cancellation guarantees that the longer representation in a pair will be lengthened only when the shorter one is all 0's.

(b) The one-dimensional version of the "origin-crossing problem" is simply to simulate one such counter. The one-dimensional version of the real-time solution described in detail by M. Fischer and A. Rosenberg [7] is the simulation we want, but they do not point out the space bound. Like the one cited above, their simulation is based on the idea of separate binary counts of increments and decrements. Whenever the number of bits in the smaller count gets within 1 of the number of bits in the larger count, the simulation subtracts the smaller count from the larger, leaving the smaller count at 0. This guarantees that the larger count is always at least twice the smaller count, hence at most twice the difference. For details (especially how to get by in real time), consult [7]. □

Remark. In the simulation for part (b) above, the length of the nonblank portion of the work tape is always proportional to a constant plus the logarithm of the current contents of the simulated counter. It follows by the methods of Leong and Seiferas [14, § 4] (using two "deque stacks") that a "stack" of such counters can be maintained in real time and in space proportional to the maximum sum of the logarithms of the stacked counters' contents. (Only a counter containing 0 may be pushed or popped, and only the top counter on the stack may be incremented, decremented, or tested.)

For the analogous linear-time Turing machine implementation to be sketched below (Theorem 5), a more powerful counter simulation lemma will be useful.

LEMMA 5. *For each increasing sequence of integers* $0=c_0<c_1<c_2<\cdots<c_m=B$, *consider a B-bounded counter which can be incremented by* 1, *decremented by* 1, *tested for* 0, *tested for B, and also tested for membership in* $\{c_0, c_1, c_2, \cdots, c_m\}$. *Any such counter can be described by the marked concatenation of the binary representations of the successive differences* $c_1-c_0, c_2-c_1, \cdots, c_m-c_{m-1}$. *There is a fixed multitape Turing machine which, given any such description* (*on an auxiliary one-way input tape, say*), *can simulate the corresponding counter in real-time and in space proportional to the length of the description.*

Proof. Let c be the current counter contents, and let $c'\geqq c$ be the largest counter contents so far. (Initially, $c=c'=0$.) The main idea is to use a Lemma 4(b) counter to maintain the length of each nontrivial interval determined by c, c', and the c_i's up to c'. Since only the intervals adjacent to c can change in length, a pair of "counter stacks" can be used for the purpose (see the remark following the proof of Lemma 4), one for the lengths of the nontrivial intervals below c ·and the other for the lengths of the nontrivial intervals above c. Finite-state memory can maintain whether $c=c_0$, whether $c=c_m$, whether $c=c'$, whether $c=c_i$ for any i, and whether $c'=c_i$ for any i. Then the simulation is straightforward so long as c' does not have to be incremented.

The only remaining problem is to increment the simulated counter contents c when $c=c'$. To make this easy, the simulator begins to quickly load an auxiliary Lemma 4(b) counter with $c_{i+1}-c_i$ as soon as c' reaches c_i for $i<m$. That same counter can be decremented when necessary to increase c' and c. Although the contents of the auxiliary counter might occasionally go negative, it will be positive when the loading process is completed, *provided* the loading requires fewer than $c_{i+1}-c_i$ steps. After the loading process is completed, the next following 0 contents in the auxiliary counter will indicate that $c'=c_{i+1}$ and free up to the counter to load $c_{i+2}-c_{i+1}$ if $i+1<m$.

Finally, we note that the auxiliary counter *can* be loaded sufficiently quickly from the successive binary representations in the given counter description. The straightforward approach of successive subtractions takes time proportional to $c_{i+1}-c_i$ [8], and that time can be sped up to less than $c_{i+1}-c_i$ by first recopying the binary representation (of length only $O(\log(c_{i+1}-c_i))$) sufficiently compactly [11]. (For $c_{i+1}-c_i$ close to 0, a more straightforward use of finite-state memory suffices.) □

THEOREM 5. *A multitape Turing machine can recognize the end of every instance of a pattern string x* (*accessible to two two-way input heads*) *as a subword in a longer on-line text string y* (*accessible to just one one-way input head*) *in time proportional to* $|y|$ *and space proportional to* $(\log|x|)^2$. *The text input head shifts right only, and it does so only after the Turing machine has decided whether the pattern is a suffix of the text prefix read so far.*

Remark. An immediate corollary is the weaker Fischer–Paterson result that an ordinary multihead (and hence multitape [14]) Turing machine can perform string-matching on-line in linear time. Galil's techniques [9] can be used to convert either of the two Turing machine algorithms to a real-time one.

Proof. Recall how we proved Theorem 4 by adapting the algorithms of Theorem 2 via Theorem 3 and Remark (iii). For this result, we adapt the $k=3$ algorithm of Theorem 1 in an analogous manner.

To adapt the algorithm, we separately adapt the preprocessing and searching phases. The searching algorithm is simpler, so we start with it. Input heads are maintained at pattern positions $q+1$ and $p+q+1-p_0$ and of course at text position i. The variable next and the parity of r are maintained in finite-state memory. Lemma 4(b) counters are maintained for $i-(p+q)$ and r. Finally, two Lemma 5 counters are maintained, both with c_i-set $\{\mathrm{GATE}(0), \cdots, \mathrm{GATE}(2l+1)=|x|+2\}$. (The adaptation below of the preprocessing algorithm will prepare the descriptions of these counters.) The first of these is used to maintain q, and the second is used to count up to GATE(r) for the calculation of $\lfloor \mathrm{GATE}(r)/3 \rfloor$ when it is needed.

During the preprocessing phase, the Turing machine adaptation maintains two more Lemma 5 counters, both with c_i-set $\{\mathrm{GATE}(0), \cdots, \mathrm{GATE}(R)\}$; and it also maintains the full descriptions (growing with R) of these counters. (At the end of preprocessing, these descriptions can be completed ($R=2l+1$) and supplied as the necessary input to the searching algorithm.) As in the searching algorithm, these counters are used to maintain q and to count up to GATE(r) for the calculation of $\lfloor \mathrm{GATE}(r)/3 \rfloor$ when it is needed. Lemma 4(b) counters are maintained for the following values:

$$\begin{aligned}&i-(p+q),\\ &r,\\ &R-r,\\ &p-\lfloor i/3 \rfloor,\\ &i-\mathrm{GATE}(R).\end{aligned}$$

The assignment $\mathrm{GATE}(R) := i$ following $R := R+1$ is executed by emptying the last counter above and appending the binary representation of its contents to the back ends of the two counter descriptions. (This calls for multihead tape units, but they can be simulated without time loss by single-head tape units [14].) The parities of r and R are maintained in finite-state memory, and input heads are maintained at pattern positions $q+1$ and $p+q+1$.

No counter contents or c_i (GATE value) ever exceeds $|x|+2$, and the number of c_i's is $O(\log |x|)$. It follows by Lemmas 4 and 5 that the space used by the Turing machine implementation we have sketched is $O((\log |x|)^2)$.

As in the proof of Theorem 3, we adapt the original time analysis. Each test can be performed in constant time, and each assignment can be performed in time proportional to the largest mandated counter change. The only assignments which can mandate more than a constant change in any counter contents are $\mathrm{GATE}(R) := i$ and those already analyzed in the proof of Theorem 3. The largest mandated counter change for the first assignment is $\mathrm{GATE}(R)-\mathrm{GATE}(R-1)$. The pattern length plus 2 is a bound on the sum of all such changes, so the *total* time spent on this assignment is only $O(|x|)$. □

Appendix A: Analysis for more general k. If we relax the requirement that k can be an integer and insist only that k exceed 2, no serious difficulties arise. It suffices to reformulate Lemma 2 and Corollaries 1–3 as shown below. The only proof that changes significantly is that of Corollary 3.

LEMMA A.2. *If* $k \geqq 2$, *then* $\mathrm{GATE}(2r-1) = \lceil k \cdot \mathrm{VAL}(r) \rceil$ *for every* r $(1 \leqq r \leqq l)$. (*Since* $\mathrm{VAL}(r)$ *is an integer and* $k \geqq 1$, *it follows that* $\mathrm{VAL}(r) = \lfloor \mathrm{GATE}(2r-1)/k \rfloor$ *for every* r.)

COROLLARY A.1. *If* $k > 2$, *then*

$$\mathrm{VAL}(r+1) > \lceil (k-1) \cdot \mathrm{VAL}(r) \rceil$$

for every r $(1 \leqq r \leqq l-1)$.

COROLLARY A.2. *If* $k \geqq 1+\sqrt{2} \approx 2.4$, *then*

$$\mathrm{GATE}(2r) < \mathrm{GATE}(2r+1)$$

for every r $(1 \leqq r \leqq l-1)$.

COROLLARY A.3. *If* $k > 2$ *and* $\mathrm{KMP}(q) = \mathrm{KMP}_k(q) = \mathrm{VAL}(r)$, *then*

$$\mathrm{GATE}(2r+2-2\lceil \log_{k-1} k \rceil) \leqq q - \mathrm{KMP}(q) < q < \mathrm{GATE}(2r).$$

Proof. Assume the hypothesis. Only the leftmost inequality is not immediate. By Lemma 3,

$$q \geqq k \cdot \mathrm{KMP}(q) > (k-1)(\mathrm{GATE}(2r-2)-1).$$

Similarly,

$$\mathrm{GATE}(2r-2i)-1 > (k-1)(\mathrm{GATE}(2r-2(i+1))-1)$$

for $i = 1, 2, \cdots$. For every i, therefore,

$$q > (k-1)^i(\mathrm{GATE}(2r-2i)-1).$$

In particular,

$$\begin{aligned} \mathrm{GATE}(2r+2-2\lceil \log_{k-1} k \rceil)-1 &< q/(k-1)^{\lceil \log_{k-1} k \rceil - 1} \\ &\leqq q - q/k \\ &\leqq q - \mathrm{KMP}_k(q). \quad \square \end{aligned}$$

For $k \geqq (3+\sqrt{5})/2 \approx 2.6$, it still follows from Corollary A.3 that at most one decrement $r := r-1$ is necessary after the change $q := q - \mathrm{KMP}(q)$. For smaller k, a larger, but still bounded, number of decrements may be necessary.

For $k \geqq 1+\sqrt{2} \approx 2.4$, it still follows from Corollary A.2 that at most one increment $r := r+1$ is necessary after the change $q := q+1$. For smaller k, the number of increments is at most two, since $\mathrm{GATE}(r+1) > \mathrm{GATE}(r)$ holds for every *odd* r, even without Corollary 2 or A.2.

In addition, in the case that $k < 1+\sqrt{2}$, one small change is needed in the preprocessing algorithm. Since $\mathrm{GATE}(r+1) = \mathrm{GATE}(r)$ is possible (for r even), the algorithm might have to add i to its GATE sequence *twice* when it discovers $\mathrm{KMP}(i) = p$. If the last element in the sequence so far is $\mathrm{GATE}(R)$, then this is necessary precisely when R is odd and i too should be an odd member of the GATE sequence. By Lemma A.2, the latter is the case if and only if $i = \lceil k \cdot \mathrm{KMP}(i) \rceil$. Shown below is the appropriate revision of the last **if**-statement in the preprocessing algorithm.

```
if (($p \leqq i/k$ and $R$ is even) or
    ($p > i/k$ and $R$ is odd)) →
    begin
      $R := R+1$;
      GATE($R$) := $i$
    end
  □ ($i = \lceil kp \rceil$ and $R$ is odd) [impossible if $k \geqq 1+\sqrt{2} \approx 2.4$] →
    begin
      $R := R+2$;
      GATE($R-1$) := $i$;
      GATE($R$) := $i$
    end
fi;
```

It is more difficult to relax the requirement that k be strictly greater than 2. For a favorable analysis of our hybrid algorithm, we need a constant c_k *strictly greater than* 1, such that $\mathrm{VAL}(r+1)/\mathrm{VAL}(r) > c_k$ for every r $(1 \leqq r \leqq l-1)$. Only for k strictly greater than 2 does Corollary 1 or A.1 to Lemma 3 above provide such a constant $(c_k = k-1)$. The analogous corollary to the following more difficult alternative to Lemma 3 provides such a constant even for $k=2$ $(c_k = 2k/3)$.

LEMMA A.3′. *If* $\mathrm{KMP}(q_3) > \mathrm{KMP}(q_2) > \mathrm{KMP}(q_1)$ *and* $\mathrm{KMP}(q_2) \leqq q_2/2$, *then* $\mathrm{KMP}(q_3) > \lceil 2q_1/3 \rceil$.

Proof. Assume the hypothesis. By Lemma 1,

$$q_1 - \mathrm{KMP}(q_1) + 1 \leqq \mathrm{KMP}(q_2).$$

Using the hypothesis $\mathrm{KMP}(q_2) \leqq q_2/2$, we show below that

$$q_1 + \mathrm{KMP}(q_1) + 1 \leqq \mathrm{KMP}(q_2) + \mathrm{KMP}(q_3).$$

Adding these inequalities, we get

$$\begin{aligned} 2q_1 + 2 &\leqq 2 \cdot \mathrm{KMP}(q_2) + \mathrm{KMP}(q_3) \\ &\leqq 3 \cdot \mathrm{KMP}(q_3) - 2, \end{aligned}$$

hence

$$\begin{aligned} \mathrm{KMP}(q_3) &\geqq 2q_1/3 + 4/3 \\ &> \lceil 2q_1/3 \rceil. \end{aligned}$$

It remains only to prove the inequality

$$q_1 + \mathrm{KMP}(q_1) < \mathrm{KMP}(q_2) + \mathrm{KMP}(q_3).$$

Suppose, to the contrary, that

$$q_1 - \mathrm{KMP}(q_2) \geqq \mathrm{KMP}(q_3) - \mathrm{KMP}(q_1).$$

Then let

$$\begin{aligned} u &= [\mathrm{KMP}(q_1), \mathrm{KMP}(q_2)]_x, \\ v &= [\mathrm{KMP}(q_2), \mathrm{KMP}(q_3)]_x, \\ w &= [\mathrm{KMP}(q_2), q_1]_x, \\ w' &= [q_1, q_2]_x. \end{aligned}$$

The string $[0, q_1]_x$ has periods of both lengths $\mathrm{KMP}(q_1)$ and $\mathrm{KMP}(q_2)$, so

$$\begin{aligned} w &= [\mathrm{KMP}(q_2), q_1]_x \\ &= [0, q_1 - \mathrm{KMP}(q_2)]_x \\ &= [\mathrm{KMP}(q_1), q_1 - \mathrm{KMP}(q_2) + \mathrm{KMP}(q_1)]_x. \end{aligned}$$

The last string is a prefix of uw, so u is a period of w. Similarly, v is a period of ww', and hence of w, too. The hypothesized inequality implies $|w| \geqq |u| + |v|$, so u and v must have a period z of length $\gcd(|u|, |v|)$, by the periodicity lemma. But then

$$\begin{aligned} v \text{ period of } ww' &\Rightarrow z \text{ period of } ww' \\ &\Rightarrow u \text{ period of } ww' \\ &\Rightarrow u \text{ period of } uww' = [\mathrm{KMP}(q_1), q_2]_x. \end{aligned}$$

In addition, since $[0, q_2]_x$ has a period of length $\mathrm{KMP}(q_2)$,

$$u \text{ period of } ww' = [\mathrm{KMP}(q_2), q_2]_x \Rightarrow u \text{ period of } [0, q_2 - \mathrm{KMP}(q_2)]_x.$$

But $q_2 - \mathrm{KMP}(q_2) \geqq \mathrm{KMP}(q_2)$, since $\mathrm{KMP}(q_2) \leqq q_2/2$. Thus u is a period of both $[0, \mathrm{KMP}(q_2)]_x$ and $[\mathrm{KMP}(q_1), q_2]_x = [\mathrm{KMP}(q_2) - |u|, q_2]_x$, hence of the entire string $[0, q_2]_x$. Therefore,

$$\begin{aligned} \mathrm{KMP}(q_2) &\leqq |u| \\ &= \mathrm{KMP}(q_2) - \mathrm{KMP}(q_1) \\ &< \mathrm{KMP}(q_2), \end{aligned}$$

a contradiction. □

COROLLARY A.1′. *If* $k \geqq 2$, *then*

$$\mathrm{VAL}(r+2) > \lceil (2k/3)\, \mathrm{VAL}(r) \rceil$$

for every r $(1 \leqq r \leqq l - 2)$.

COROLLARY A.3′. *If* $k \geqq 2$ *and* $\mathrm{KMP}(q) = \mathrm{KMP}_k(q) = \mathrm{VAL}(r)$, then $\mathrm{GATE}(2r - 12) \leqq \mathrm{GATE}(2r - 4\lceil \log_{2k/3} (k/(k-1)) \rceil) \leqq q - \mathrm{KMP}(q) < q < \mathrm{GATE}(2r)$.

Proof. Use Lemma A.3′ rather than Lemma 3 in the proof of Corollary A.3. □

By Corollary A.1′, $l < 2 \cdot \log_{2k/3} |x| = O((\log |x|)/(\log k))$, even for $k = 2$. In addition, this analysis is superior to the earlier one for $2 < k < (4 + \sqrt{7})/3 \approx 2.2$.

Although Corollary A.3 in this appendix gives a bound on the number of decrements $r := r - 1$ necessary after the change $q := q - \mathrm{KMP}(q)$, that bound depends on k and does not apply for $k = 2$. Corollary A.3′ guarantees that eleven decrements suffice regardless of k, even for $k = 2$.

The generalizations above lead to a favorable analysis of our hybrid algorithm for every rational $k \geqq 2$. The number of local memory locations is about $10 + 2 \cdot \min\{\log_{k-1} |x|, 2 \cdot \log_{2k/3} |x|\}$, and the time is proportional to $k|y|$. (This analysis notwithstanding, minimizing k may *not* give the fastest algorithm. The **while**-loops which increment and decrement r (at most twice and eleven times, respectively) can iterate the most for very small values of k. Moreover, only for $k \geqq (3 + \sqrt{5})/2 \approx 2.6 > 1 + \sqrt{2} \approx 2.4$ can both **while**-loops be replaced by **if**-statements to save additional time. With these changes, the algorithm for k equal to $(3 + \sqrt{5})/2$ (or some slightly larger rational number) becomes a stronger candidate for our fastest hybrid algorithm.)

Finally, for completeness, let us note that the hybrid algorithm for $1<k<2$ does *not* save significant space. For k in this range, l can be proportional to the pattern length. In particular, l is $\Omega((2-k)|x|/(k-1))$ for patterns x of the form $\mathrm{a}^n\mathrm{b}\mathrm{a}^{\lfloor n/(k-1)\rfloor}$.

Appendix B: Hybrid algorithm taking mismatch into account. In the manner described earlier, this most complete version of the hybrid algorithm for fixed $k\geqq 2$ (Appendix A is a prerequisite here) takes into account the mismatch $x(q+1)\neq y(p+q+1)$ when it increments p. More specifically, it increments p by KMP(q) only when $q+1=\mathrm{GATE}(r)$ for some even r. For each *odd* r, this version stores $\lfloor \mathrm{GATE}(r)/k\rfloor=\mathrm{VAL}((r+1)/2)$ instead of GATE(r). As above, we list the simpler text-searching algorithm first:

```
$(p, q, r) := (0, 0, 0)$; [$r$ always even]
$i := 0$;
loop:
  $i := i+1$: [$i = p+q+1$]
  read next; [next $= y(p+q+1)$]
  $p_0 := p$;
  until $p+q = i$ do
    if (($p+q+1 = i$ and $x(q+1) =$ next) or
        ($p+q+1 < i$ and $x(q+1) = x(p+q+1-p_0)$))
      then
        begin
          $q := q+1$;
          if $q \geqq \mathrm{GATE}(r+2)$ then $r := r+2$
        end
      else
        if $q = 0 \rightarrow (p, q, r) := (p+1, 0, 0)$
          □ $q+1 = \mathrm{GATE}(r+2) \rightarrow$
            $(p, q) := (p+\mathrm{VAL}(r/2+1), q-\mathrm{VAL}(r/2+1))$;
            while $q < \mathrm{GATE}(r)$ [impossible if $k \geqq (3+\sqrt{5})/2$]
              do $r := r-2$ [at most five times]
          □ else $\rightarrow (p, q, r) := (p+\lceil q/k \rceil, 0, 0)$
        fi;
  if $q = |x|$ then declare match;
  go to loop
```

Finally, here is the algorithm for preprocessing the pattern:

```
$\mathrm{GATE}(0) := 0$;
$(p, q, r) := (1, 0, 0)$;
$i := 1$; read $x(i)$; $R := 0$;
loop:
  $i := i+1$; read $x(i)$; [$i = p+q+1$]
  if $x(i) = \$$ then
    begin
      if $R$ is even
        then [$R = 2l$] $\mathrm{GATE}(R+2) := i+1$
        else [$R = 2l-1$] $(\mathrm{GATE}(R+1), \mathrm{GATE}(R+3)) := (i, i+1)$;
      return
    end
```

```
until $p+q=i$ do
  if $x(q+1)=x(p+q+1)$
    then
      begin
        $q := q+1$;
        if ($r+2 \leqq R$ and $q \geqq \text{GATE}(r+2)$) then $r := r+2$
      end
    else
        if $q=0 \rightarrow (p, q, r) := (p+1, 0, 0)$
          □ ($r+2 \leqq R$ and $q+1=\text{GATE}(r+2)$) →
            $(p, q) := (p+\text{VAL}(r/2+1), q-\text{VAL}(r/2+1))$;
            while $q<\text{GATE}(r)$ [impossible if $k \geqq (3+\sqrt{5})/2$]
              do $r := r-2$ [at most five times]
          □ else → $(p, q, r) := (p+\lceil q/k \rceil, 0, 0)$
        fi;
if ($p \leqq i/k$ and $R$ is even) →
    begin
      $R := R+1$;
      $\text{VAL}((R+1)/2) := p$
    end
  □ ($p>i/k$ and $R$ is odd) →
    begin
      $R := R+1$;
      $\text{GATE}(R) := i$
    end
  □ ($i=\lceil kp \rceil$ and $R$ is odd) [impossible if $k \geqq 1+\sqrt{2}$] →
    begin
      $R := R+2$;
      $\text{GATE}(R-1) := i$;
      $\text{VAL}((R+1)/2) := p$
    end
fi;
go to loop
```

Acknowledgments. Ruth Zimmerman pointed out that the VAL sequence can be inferred from the GATE sequence if k is large, Leo Guibas raised the question of how the hybrid algorithm behaves for $k=2$, and Allan Borodin reminded us of Cobham's cited work.

A preliminary version of this paper was presented at the Eighteenth Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, November 1977.

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. BARTH, *One pass and extended string matching*, manuscript, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1977.

[3] A. K. CHANDRA, *Efficient compilation of linear recursive programs*, 14th Annual Symposium on Switching and Automata Theory, Iowa City, IA, 1973, pp. 16–25.

[4] A. COBHAM, *The recognition problem for the set of perfect squares*, IEEE Conference Record of 1966 Seventh Annual Symposium on Switching and Automata Theory, Berkeley, CA, 1966, pp. 78–87.

[5] E. W. DIJKSTRA, *Guarded commands, nondeterminacy and formal derivation of programs*, Comm. ACM, 18 (1975), pp. 453–457.

[6] M. J. FISCHER AND M. S. PATERSON, *String-matching and other products*, Complexity of Computation (SIAM–AMS Proceedings, vol. 7), R. M. Karp, ed., American Mathematical Society, Providence, RI, 1974, pp. 113–125.

[7] M. J. FISCHER AND A. L. ROSENBERG, *Real-time solutions of the origin-crossing problem*, Math. Systems Theory, 2 (1968), pp. 257–263.

[8] P. C. FISCHER, A. R. MEYER AND A. L. ROSENBERG, *Counter machines and counter languages*, Ibid., pp. 265–283.

[9] Z. GALIL, *String-matching in real time*, J. Assoc. Comput. Mach., to appear.

[10] ———, *Palindrome recognition in real time by a multitape Turing machine*, J. Comput. System Sci. 16 (1978), pp. 140–157.

[11] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc. 117 (1965), pp. 285–306.

[12] J. E. HOPCROFT, W. J. PAUL AND L. G. VALIANT, *On time versus space*, J. Assoc. Comput. Mach. 24 (1977), pp. 332–337.

[13] D. E. KNUTH, J. H. MORRIS, JR. AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

[14] B. L. LEONG AND J. I. SEIFERAS, *New real-time simulations of multihead tape units*, J. Assoc. Comput. Mach., to appear.

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

[16] C. G. NELSON, *One-way automata on bounded languages*, Rep. TR-14-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA, July 1976.

[17] J. I. SEIFERAS AND Z. GALIL, *Real-time recognition of substring repetition and reversal*, Math. Systems Theory, 11 (1977), pp. 111–146.

[18] A. C. YAO AND R. L. RIVEST, *$k+1$ heads are better than k*, J. Assoc. Comput. Mach. 25 (1978), pp. 337–340.

SIAM J. COMPUT.
Vol. 9, No. 2, May 1980

0097-5397/80/0902-0014 $01.00/0

CORRIGENDUM: A FAMILY OF ALGORITHMS FOR POWERING SPARSE POLYNOMIALS*

DAVID K. PROBST† AND VANGALUR S. ALAGAR†

On p. 629, the concluding sentence of Section 2 should read:

Our analysis of various algorithms leads us to conjecture that algorithm BINF is optimal for time and space within the entire family of sequential binomial-expansion algorithms for computing integer powers of sparse polynomials.

* This Journal, 8 (1979), pp. 626-644.

† Department of Computer Science, Concordia University, Montreal, Quebec, Canada H3G 1M8.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

THE MATHEMATICS OF RECORD HANDLING*

HARTMUT EHRIG† AND BARRY K. ROSEN‡

Abstract. We propose a mathematical foundation for reasoning about the correctness and computational complexity of record handling algorithms, using algebraic methods recently introduced in graph theory. A class of pattern matching and replacement rules for graphs is specified, such that applications of rules in the class can readily be programmed as rapid transformations of record structures. When transformations of record structures are formalized as applications of rules to appropriate graphs, recent Church–Rosser type theorems of algebraic graph theory become available for proving that families of transformations are well behaved. In particular, we show that any Church–Rosser family of transformations can be combined with housekeeping operations involving indirect pointers and garbage collection without losing the Church–Rosser property, provided certain mild conditions on the rules defining the family are satisfied. This leads to suggestions for the design of record handling facilities in high level languages, especially when housekeeping chores are to be performed asynchronously by service processes that run in parallel with the main process. These results and the general theorems that support them can be used to analyze the behavior of a large record structure that can be updated asynchronously by several parallel processes or users.

Key words. record structure, indirect addressing, garbage collection, asynchronous parallelism, Church–Rosser property, category theory, pushout, graph, production, derivation

1. Introduction. One major difference between record handling [17, § 2.1], [30, § 4.4] and numerical computing is the lack of a standard mathematical foundation. Without precise but language independent definitions of record structures and the basic operations on them, it is difficult to study language design issues for record handling. Proofs of correctness or complexity bounds for record handling programs need a mathematical foundation. Graph theory is the obvious place to look for appropriate material, since a record structure is naturally viewed as a directed graph wherein each record is a node and each pointer from one record to another is an arc. There is a voluminous literature on algorithms involving graphs (as in [11], [16], [24], [27], [28]), but the relevance of this literature to many record handling problems is doubtful. Running time linear in the size of an input graph is ordinarily and rightly considered fast in the literature. But processes requiring that an entire set of records be scanned are ordinarily and rightly considered slow in record handling. This is not to disparage the literature: we merely want to emphasize that different problems arise in record handling. In particular, record handling algorithms often deal with local properties and transformations on the graph, and with possibly unpleasant interactions between transformations when several users can update a data base asynchronously. An algebraic approach to graph theory has recently been introduced, partly to deal with these concerns. This paper's results are applicable to record handling, but the presentation of these results is also a convenient occasion to explain the algebraic approach in a new way. The earliest work [6], [21] had unusual prerequisites and some technical complexities that were later eliminated [4], [7]. Enough results have now accumulated to support an exposition that is intelligible and perhaps even plausible without large prerequisites. Those who wish to read the proofs carefully will sometimes need to consult [20], [21], but only as indicated by citations. A two pass reading is recommended, with proofs omitted on the first pass.

* Received by the editors September 27, 1978, and in revised form May 15, 1979.

† Fachbereich Informatik, Technische Universität Berlin, 1000 Berlin 10, Federal Republic of Germany. Part of the work of this author was done while visiting the IBM Thomas J. Watson Research Center, with support from IBM Germany and IBM World Trade Corporation.

‡ Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 10598.

Section 2 reviews some well-known concepts from record handling and relates transformations of record structures to "productions" that can be applied to graphs. There is a similarity in ultimate intentions between this work and [19] without any technical similarity. A specific record handling language receives an interpretive semantics and validated proof rules in [19]. This language has the usual basic facilities and is generally of lower level than our discussion. The effects of any "node assignment statement" [19, p. 109] are readily obtained by applying an appropriate production, but many of our productions can only be simulated by nontrivial programs in the language. This paper complements [19]. After a very high level algorithm has been certified with the aid of our methods, the correctness of a realization at the level of ALGOL *W* or Pascal could be demonstrated with the aid of [19]. Section 3 presents fundamental existence theorems that are applied here but are not limited to record handling.

The next two sections deal with housekeeping tasks in record structures: maintenance of indirect pointers in § 4 and a simple form of garbage collection in § 5. We establish conditions under which housekeeping operations do not interfere with whatever else is being done. For example, list processing should not be stymied because a little garbage has been collected by a second processor, operating asynchronously in parallel with the main processor. (The resulting suggestions for language design are collected in § 6.) Noninterference is formalized by the "Church–Rosser property" (defined here essentially as in [20, Def. 3.2]), involving an arbitrary set of objects and a single relation among the objects. Given a set $\mathscr{B}$ and a relation $\gg$ on $\mathscr{B}$, consider any $\mathscr{F} \subseteq \mathscr{B}$ such that $\mathscr{F}$ is *closed* under $\gg$: if $\mathbf{G}$ is in $\mathscr{F}$ and $\mathbf{G} \gg \mathbf{H}$ then $\mathbf{H}$ is in $\mathscr{F}$. Then the system $(\mathscr{F}, \gg)$ is *Church–Rosser* iff

$$(1.1)\qquad \begin{array}{l}(\forall \mathbf{G}, \mathbf{H}, \mathbf{H}^{\#} \text{ in } \mathscr{F})\\ \qquad [(\mathbf{G} \gg^{*} \mathbf{H} \;\&\; \mathbf{G} \gg^{*} \mathbf{H}^{\#}) \text{ implies } (\exists \mathbf{X} \text{ in } \mathscr{B})(\mathbf{H} \gg^{*} \mathbf{X} \;\&\; \mathbf{H}^{\#} \gg^{*} \mathbf{X})],\end{array}$$

where $\gg^{*}$ is the reflexive transitive closure of $\gg$. If $\mathbf{G}$ is in $\mathscr{F}$ and

$$(1.2)\qquad \mathbf{G} \gg^{*} \mathbf{H} \quad \text{and} \quad \neg(\exists \mathbf{X} \text{ in } \mathscr{B})(\mathbf{H} \gg \mathbf{X})$$

then $\mathbf{H}$ is a *normal form* for $\mathbf{G}$ in $(\mathscr{F}, \gg)$. The Church–Rosser property implies that normal forms are unique when they exist at all. Our (1.1) is a special case of the more general Church–Rosser property involving two relations as studied in [15, § I.3], [23], [22]. By distinguishing $\mathscr{F}$ from $\mathscr{B}$ we gain some notational convenience over [15], [20], [22], [23] in applications without changing the theory.

Staples [25] calls the pair $(\mathscr{B}, \gg)$ "subcommutative" if the following property holds, where $\mathscr{F}$ is the set of all $\mathbf{G}$ in $\mathscr{B}$ that have normal forms in $(\mathscr{B}, \gg)$ and $\gg^{=}$ is the reflexive closure of $\gg$:

$$(1.3)\qquad \begin{array}{l}(\forall \mathbf{G}, \mathbf{H}, \mathbf{H}^{\#} \text{ in } \mathscr{F})\\ \qquad [(\mathbf{G} \gg^{=} \mathbf{H} \;\&\; \mathbf{G} \gg^{=} \mathbf{H}^{\#}) \text{ implies } (\exists \mathbf{X} \text{ in } \mathscr{B})(\mathbf{H} \gg^{=} X \;\&\; \mathbf{H}^{\#} \gg^{=} \mathbf{X})].\end{array}$$

For any $\mathscr{F} \subseteq \mathscr{B}$ such that $\mathscr{F}$ is closed under $\gg$ and (1.3) holds, we will say that $(\mathscr{F}, \gg)$ is *strongly Church–Rosser*. Because (1.3) implies closure under $\gg$ for the choice of $\mathscr{F}$ in [25], our property of being strongly Church–Rosser is a little more general than subcommutativity. When interest centers on normal form computations and their lengths, the concepts are interchangeable and have the significance explained in [25]. Strongly Church–Rosser systems will be important when we study indirection in § 4 here, because there are frequently occurring conditions under which one system's being Church–Rosser follows from another (and simpler) system's being strongly Church–Rosser. Finally, note that the first two occurrences of $\gg^{=}$ in (1.3) could be replaced by $\gg$

without changing the class of strongly Church–Rosser systems. This is helpful in verifying the property. For using the property the given formulation is more convenient.

Notations are standard, except that we avoid unnecessary parentheses. The value of a map f at an argument x is fx. If A is a subset of the domain of f then fA is $\{fx \mid x \in A\}$. A bare minimum of categorical machinery is used, and most of the ideas are explained when introduced here. The few exceptions are widely known basic concepts that need to be set in a broad context such as is provided by the introductory chapters in books like [1] or [14]. The reader should have a slight familiarity with *objects* and *morphisms* in *categories* [1, pp. 29, 30], [14, p. 16] and with *isomorphisms* [1, p. 35], [14, p.35] and *commutative diagrams* [1, pp. 2, 4], [14, pp. 3, 17]. The usefulness of these basic concepts is like the usefulness of procedures in programming. Elaborate computations can be invoked by a simple procedure call, and elaborate calculations with large combinatorial objects can be summarized by simple diagrams. Some of the calculations can even be avoided, thanks to simpler calculations that establish the hypotheses of categorical lemmas. Adding procedures to a programming language makes the language a little harder to learn but a lot easier to use. Adding categorical ideas to a theory of record handling has a similar effect, though we do not claim as dramatic a ratio of benefit to cost. Neither do we claim that category theory is a panacea. We do claim to have proved some interesting theorems about record handling, such that the theorems are extremely difficult to prove without categorical ideas. (After the fact, one can laboriously translate our proofs into ones that formally avoid category theory. This is not helpful.)

2. Record structures and expression graphs. Complex information can be represented in a computing system's memory by distributing it over many "records", each of which has a relatively small number of "fields". A field may directly contain a little information, such as a short string of characters or an integer that can be represented with 15 bits, or it may contain a "pointer" to another record that must be consulted if more information is desired. (In some applications a pointer might loop back to the same record.) Records are often divided into "classes", so that all records in a class have the same number of fields, the same names for the respective fields, and the same kinds of direct or pointer data in their fields. (Different fields may contain different kinds of data.) These concepts are well known [17, § 2.1], [30, § 4.4] but appear in many places under many names. We will speak as above. A set of records such that each pointer is to a record in the set is a *record structure*. We formalize the intuitive concept of record structures with the mathematical concept of *colored graphs*. Such a graph is a pair (G, m_G), where G is a finite direct graph. A *node* in G corresponds to a record. There is an *arc* from node x to node y whenever record x has a field that points to record y. (If there are two such fields then there are two such arcs; we do *not* assume that arcs are pairs of nodes.) Nodes and arcs are both called *items*, and m_G maps items to packets of information called *colors*. The color of a node tells how many fields it has and what their names are and what data is stored directly in fields that do not contain pointers. The color of an arc from x to y indicates which field of x is responsible for the arc's being in G. Mathematically, we just have nonempty sets of *node colors* and *arc colors* with m_G mapping nodes to node colors and arcs to arc colors. We will be casual about the distinction between the intuitive concepts like "record" and the mathematical concepts like "node" whenever there is no danger of confusion.

We begin our study of transformations of record structures with an example. Suppose RD1 is a record and we wish to replace all pointers to RD1 in the record structure with pointers to another record RD2. In many situations it is expensive to find all pointers to RD1: only pointers *from* RD1 are stored as fields of RD1. It is also intuitively plausible that we only really need to change whatever pointers will actually be followed in the future. In this context it is natural to simply replace RD1's data without moving it. We change RD1 so that it has just one pointer field, pointing to RD2. The direct data at RD1 is changed to indicate that RD1 is now merely a dummy record, so that pointers to RD1 will be treated as indirect pointers to RD2. In the future, whenever we follow a pointer from RD3 to RD1, we will update the appropriate field of RD3 so as to point directly to RD2 if the pointer is followed again. One of the contexts where this natural use of *indirection* is especially appropriate is in the efficient evaluation of recursively defined functions, as is discussed in [18, Appendix A]. The discussion above and in [18, Appendix A] can be formalized easily. We assume there is a distinguished node color I such that any node colored I in a record structure has exactly one outarc, and this outarc carries the distinguished arc color *ind*. Formally, a colored graph is a *record structure* here iff it uses the special colors I, *ind* in this way. The record structures that actually arise in any one application will of course satisfy many additional constraints because of the meanings of other colors.

For an arbitrary arc color c, suppose our record structure (G, m_G) includes an arc z whose target (the node it points to) is colored I. This node $\mathbf{t}z$ has a unique outarc with a target w, and w is the record indirectly pointed to by pointers to $\mathbf{t}z$. Thus z should be replaced by a new arc ζ with the same color c and the same source (the node it points from) that z has in G. In the new record structure (H, m_H), the target of ζ is w. Otherwise the new structure is like the old one. For example, consider (G, m_G) as shown on the left in Fig. 2.1. Then (H, m_H) is as shown on the right in Fig. 2.1. The role of the colored graph (D, m_D) in the figure is explained below.

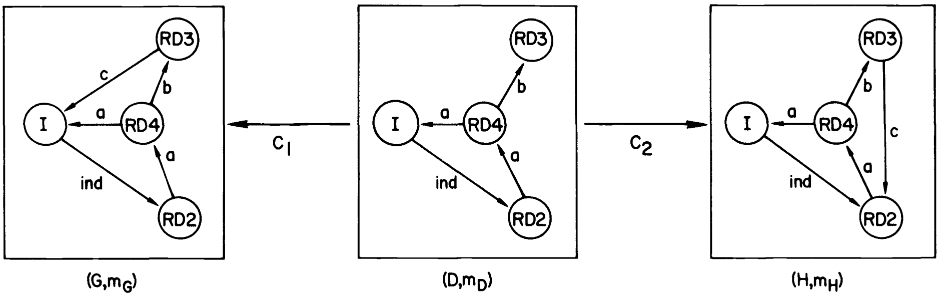


FIG. 2.1. *In (G, m_G) the field c of record RD3 points indirectly to RD2. In (H, m_H) the pointer data has been changed so as to point directly to RD2.*

To specify the transformation precisely without committing ourselves to any one programming language, we use a "production" in the sense of algebraic graph theory. We pass from (G, m_G) to (H, m_H) by applying a *production* $p = [(B_1, m_1) \leftarrow K \rightarrow (B_2, m_2)]$ consisting of a graph K, colored graphs (B_i, m_i), and maps $b_i\colon K \rightarrow B_i$. These maps are required to be *graph morphisms*: if x is an arc in K with source $\mathbf{s}_K x$ and target $\mathbf{t}_K x$, then the image arc $b_i x$ in B_i has sources and targets

$$(2.1) \qquad \mathbf{s}_i b_i x = b_i \mathbf{s}_K x \quad \text{and} \quad \mathbf{t}_i b_i x = b_i \mathbf{t}_K x.$$

Readers with an algebraic background will note that this coincides with the usual notion of a (homo)morphism of algebraic structures whose carriers are collections of sets. In our case, a graph is a finite algebra carried by a set of nodes and a set of arcs. The operations are the source and target maps $\mathbf{s}$ and $\mathbf{t}$ from arcs to nodes. The specific production $p_{\text{ind}}(c)$ that retargets an arc colored c whose target is colored I appears in

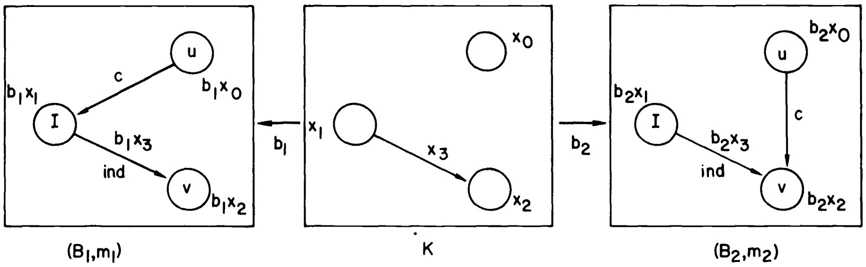


FIG. 2.2. *A production consists of an interface graph K, left and right colored graphs (B_i, m_i), and graph morphisms b_i: $K \to B_i$.*

Fig. 2.2. In addition to the node color I and the arc colors c, *ind* already introduced, we use node colors u, v in (B_i, m_i) as variables to represent whatever color may actually appear on the nodes in G. The first step in applying p to (G, m_G) is to "recolor" the variables u, v to appropriate node colors ru, rv. The resulting production $rp_{\text{ind}}(c)$ has colored graphs (B_i, rm_i) wherein any node x in B_i has the color rm_ix. (As a map from colors to colors, r leaves all colors but u, v fixed.) For the example (G, m_G) in Fig. 2.1, we use $ru = \text{RD3}$ and $rv = \text{RD2}$. Why do we not use a single production p_{ind}, with a variable arc color that can be recolored to each arc color c of interest? Intuitively, this would be preferable to having a different production $p_{\text{ind}}(c)$ for each c. Technically, however, it is very convenient to avoid having variably colored items in B_i that are not in b_iK. It is also very convenient to have the arcs colored c in $p_{\text{ind}}(c)$ be outside b_iK. We hope that latter convenience will be obviated by future developments, but for the present a multiplicity of productions provides a good deal of technical convenience at a moderate cost in intuitive naturalness.

Figure 2.3 summarizes the algebraic construction that applies $rp_{\text{ind}}(c)$ to (G, m_G) to the arc z with $m_Gz = c$. The horizontal arrows at the top of Fig. 2.3 come from the production $rp_{\text{ind}}(c)$, and we have added colorings $m_{K,i}$ of K such that b_1 and b_2 are *colored graph morphisms* now: they preserve colors as well as sources and targets. In this example $m_{K,1} = m_{K,2}$, but in general a production might change some colors: some x in K has $rm_1b_1x \neq rm_2b_2x$. The colored graph morphism g: $(B_1, rm_1) \to (G, m_G)$ picks out the "three" nodes and "two" arcs in G where the production is to be applied. (We use the quotation marks because there is no requirement that g or any other morphisms here be injective.) The image subgraph gB_1 in G may have connections with the rest of

$$\begin{tikzcd}
(B_1, m_1) \arrow[d, "g"] & (K, m_{K,1})(K, m_{K,2}) \arrow[l, "b_1"'] \arrow[r, "b_2"] \arrow[d, "d"] & (B_2, m_2) \arrow[d, "h"] \\
(G, m_G) & (D, m_{D,1})(D, m_{D,2}) \arrow[l, "c_1"] \arrow[r, "c_2"'] & (H, m_H)
\end{tikzcd}$$

FIG. 2.3. *A derivation consists of two pushouts in the category of colored graphs that share a graph morphism d: $K \to D$.*

G: an arc in $G-gB_1$ may have a source or target in gB_1. We require that such a source or target be in gb_1K, so that the connections can be displayed by graph morphisms $d: K \rightarrow D$ and $c_1: D \rightarrow G$ such that the composition c_1d is equal to the composition gb_1. These connections are retained when (B_2, rm_2) replaces (B_1, rm_1) to form (H, m_H): there is a graph morphism c_2 with $c_2d = hb_2$, where h displays how B_2 fits into H after the transformation. Thus Fig. 2.3 is a commutative diagram in the category GRAPHS$[C]$ of (finite directed) graphs together with colorings m_G: (items in $G) \rightarrow C$, where C is the set of colors used. (Actually, C is a pair of sets, one for nodes and one for arcs.) The morphisms in this category are of course the colored graph morphisms defined above.

For any recolored production rp, not just our example $rp_{\text{ind}}(c)$, Fig. 2.3 is called a *direct deviation* $(G, m_G) \Rightarrow (H, m_H)$ *via rp based on g* provided that two conditions hold. First, $m_{D,1}$ only differs from $m_{D,2}$ as required by color changes specified in p:

(2.2.1) $$m_{D,1}y = m_{D,2}y \quad \text{for all } y \text{ in } D-dK.$$

Second, the two squares in Fig. 2.3 completely describe (G, m_G) and (H, m_H) in the following precise sense:

(2.2.2) both squares are pushouts in GRAPHS$[C]$.

See Appendix B for an explanation of the intuitive significance of pushouts, followed by the algebraic definition. Much of this paper can be read with only the knowledge that pushouts are commutative squares.

For the example (G, m_G) in Fig. 2.1, there is a direct derivation $(G, m_G) \Rightarrow (H, m_H)$ via $rp_{\text{ind}}(c)$ based on $g: B_1 \rightarrow G$, where B_1 is from Fig. 2.2. The morphism g maps the arc colored c in (B_1, rm_1) to the arc colored c in (G, m_G) and it maps the arc colored *ind* in (B_1, rm_1) to the arc colored *ind* in (G, m_G). The horizontal arrows at the bottom of Fig. 2.3 are shown in detail in Fig. 2.1, with $m_{D,1} = m_{D,2}$ in this example. In general, derivations via $rp_{\text{ind}}(c)$ will have $m_{D,1} = m_{D,2}$ but derivations via many other productions will not. In this example the morphisms are all injective, but this is not required in general. We could easily have $g^{\#}b_1x_0 = g^{\#}b_1x_2$ in an application of $rp_{\text{ind}}(c)$ to some other graph $(G^{\#}, m_G^{\#})$. Of course a *derivation* is a sequence of direct derivations, and it is via the sequence of productions used. The qualifier "direct" is often omitted when confusion is unlikely. Of special interest are the *natural* derivations: those with c_1 injective. In our example b_1 is injective, and this implies (by an elementary property of pushouts) that any derivation via $rp_{\text{ind}}(c)$ is natural.

The finiteness of graphs will sometimes be helpful here, so we restrict attention to finite graphs at the start. Because of our interest in manipulating large record structures, however, we want to avoid productions such that the work required to apply rp to (G, m_G) depends on the size of G. In particular, suppose there is a node y in $B_1 - b_1K$. Then we cannot apply rp with $g: B_1 \rightarrow G$ unless the node gy in G has no inarcs or outarcs beyond those in gB_1. This is a property of pushouts in GRAPHS$[C]$ reflecting the intuition that such arcs would be left "dangling" when B_2 replaced B_1. To check that gy has no unwanted inarcs or outarcs without scanning G, we want to represent G in such a way that the indegree or outdegree of a node can be found in "one" step, or at worst in time independent of the size of G. In record handling the outdegree of a node is simply the number of nonnull pointer fields. The indegree is simply the value of the reference count if each record is maintained with such a count of how many pointers to the record are currently in the record structure. Programs for finding and maintaining indegree and outdegree information are easily written in this context. The general theory is not concerned with how this is done. (In the very common special case where b_1 is surjective on nodes, there is no worry about dangling arcs and hence no need for indegrees and outdegrees.) Given rp and $g: B_1 \to G$, it should be possible to determine whether a natural derivation via rp based on g can be constructed (and then construct one if the answer is yes) in such a way that the costs depend on the sizes of graphs in rp but not on the size of G. Mathematically, p is said to be *fast* iff

(2.3) $$b_1 \text{ and } b_2 \text{ are injective.}$$

Indirect productions $p_{\mathrm{ind}}(c)$ and all productions in [7, § 7] are fast. Before showing that fast productions can be applied rapidly, we prove a useful general lemma.

LEMMA 2.4. *Given an injective graph morphism $b_1: K \to B_1$ and a colored graph morphism $g: (B_1, m_1) \to (G, m_G)$, consider the pair (nodes, arcs) of sets*

(1) $$(N_0, A_0) = G - gB_1.$$

There is a pushout of the form

(2) $$\begin{tikzcd}(B_1, m_1) \arrow[d, "g"'] & & (K, m_K) \arrow[ll, "b_1"'] \arrow[d, "d"] \\ (G, m_G) & & (D, m_D) \arrow[ll, "c_1"]\end{tikzcd}$$

iff

(3) $$\mathrm{s}_G A_0 \cup \mathbf{t}_G A_0 \subseteq N_0 \cup gb_1K;$$

(4) $$(\forall y, y' \text{ in } B_1)[gy = gy' \textit{ implies } (y = y' \textit{ or } y, y' \textit{ in } b_1K)].$$

In that case (2) *is unique up to isomorphism.*

Proof. For b_1 injective the only pushouts (2) are those with c_1 injective: pushing out from an injection yields an injection on the opposite side of the square. That (2) exists iff (3) and (4) hold now follows from [21, Lemma 4.1]. Uniqueness follows from [21, Lemma 4.2]. □

An intuitive explanation of the "only if" part of Lemma 2.4 is helpful in understanding the significance of pushouts. In Lemma 2.4(2) we "glue" B_1 and D together to form G, as discussed in Appendix B. If $gy = gy'$ for $y \neq y'$ then y and y' are both involved in gluing. For items in B_1, to be involved in gluing *is* to be in b_1K, so it is to be expected that Lemma 2.4(4) holds when Lemma 2.4(2) is a pushout. For Lemma 2.4(3), consider any arc z in A_0. Because z is not in gB_1 it must be c_1y for some arc y of D. The target $\mathbf{t}_Dy$ has $c_1\mathbf{t}_Dy = \mathbf{t}_Gc_1y = \mathbf{t}_Gz$. We want this to be in $N_0 \cup gb_1K$. We may assume $\mathbf{t}_Gz$ is not in N_0 and hence is in gB_1 as well as c_1D. But the intersection of these images is gb_1K because G is as close to being a disjoint union as gluing will allow. The detailed calculations in the proofs in this paper often use this characterization of pushouts in categories like SETS or GRAPHS as well as the universal property (see Appendix B) that defines the pushout concept for any category.

THEOREM 2.5. *There is an algorithm that, given a colored graph (G, m_G), a recoloring $r: C \to C$, a fast production p, and a graph morphism $g: B_1 \to G$, determines whether there is a derivation $(G, m_G) \Rightarrow (H, m_H)$ via rp based on g and transforms (G, m_G) to the resulting (H, m_H) when the answer is affirmative. The derivation is unique* *up to isomorphism and the number of steps required by the algorithm is independent of the size of G provided that evaluations of relevant maps* $(g, m_G, \mathbf{s}_G, \mathbf{t}_G)$ *can be done in time independent of the size of G and provided that the indegree and outdegree of any node in G can be found in time independent of the size of G.*

Proof. It is easy to reduce the existence problem for derivations to the existence problem for "analyses" (the left square in Fig. 2.3). More precisely, an *analysis* of (G, m_G) for rp based on g is any pushout as in Lemma 2.4(2) such that

$$(1) \qquad (\forall x, x' \text{ in } K)(gb_1x = gb_1x' \text{ implies } rm_2b_2x = rm_2b_2x').$$

Given g and rp, we can test whether (1) holds in time independent of the size of G. By [21, Thm. 3.7], an analysis determines a unique derivation. When such a derivation is known to exist, injectivity of b_2 allows us to pass from (G, m_G) to (H, m_H) directly without explicitly constructing the derivation. (The time to copy parts of G that are merely carried along unchanged would of course depend on the size of G, so we must avoid the explicit construction.) Thus the whole problem reduces to the problem addressed in Lemma 2.4. Whether Lemma 2.4(3) holds can be checked by comparing indegrees and outdegrees in G with those expected from B_1 for nodes in $gB_1 - gb_1K$. (If b_1 is surjective on nodes then Lemma 2.4(3) is trivially true.) Whether Lemma 2.4(4) holds can also be checked in time independent of the size of G. □

Graph morphisms $g: B_1 \to G$ provide a flexible way to say *where* we are applying a production, but in most examples we only need the values of g on a few items of B_1 to determine the rest. In particular, let y be the arc colored c in Fig. 2.2. Because nodes colored I have unique outarcs in the colored graphs we consider, g is determined by gy. The pair $(p_{\text{ind}}(c), y)$ is a *rule*, as is any pair consisting of a production p and an item y in B_1. In a derivation based on g the rule is applied *at* gy. This wording has no theoretical significance, but it enhances brevity and clarity whenever the rule is such that g can be recovered from knowledge of gy alone.

We restrict the set R of recolorings $r: C \to C$ a little more than necessary rather than burden the statements of theorems with assumptions about R that are weak but difficult to remember. Specializing (5.5) and (5.7) from [7] for technical convenience here, we consider the *fixed* colors C_{fix} and the *variable* colors C_{var}:

$$(2.6.1) \qquad C_{\text{fix}} = \{c \text{ in } C \mid (\forall r \text{ in } R)(rc = c)\} \quad \text{and} \quad C_{\text{var}} = C - C_{\text{fix}}.$$

Now R is assumed to contain everything it might conceivably contain, with no correlation between the recolorings of different colors:

$$(2.6.2) \qquad R = \{r: C \to C \mid (\forall c \text{ in } C_{\text{fix}})(rc = c)\}.$$

With this choice of R, derivations via $Rp_{\text{ind}}(c)$ (which is to say, derivations via $rp_{\text{ind}}(c)$ for some r in R) have the intended effect: any indirect c-pointer is retargeted to the record that its old target points to, as soon as we apply $(Rp_{\text{ind}}(c), y)$ at the pointer in question.

3. Existence theorems for derivations. This section presents fundamental existence theorems for derivations among colored graphs. The proofs appear elsewhere, as indicated for each result. The pure mathematics applied in this paper is developing rapidly, and the results presented here are not the strongest known. This section contains just enough of the mathematics to support the applications in the following sections. We have tried to avoid formulations that might become significantly outdated by likely developments in the future, while resisting the temptation to rely on future work or to wait for the theory to mature without the benefit of experience in applications.

Given a derivation from (G_0, m_0) to (G_n, m_n) and a morphism $\gamma_0: (G_0, m_0) \to (\Gamma_0, \mu_0)$, we would like to construct a "corresponding" derivation from (Γ_0, μ_0) to a colored graph (Γ_n, μ_n), with an "embedding" $\gamma_n: (G_n, m_n) \to (\Gamma_n, \mu_n)$. Moreover, details of the intermediate steps between (G_0, m_0) and (G_n, m_n) should not affect (Γ_n, μ_n) and γ_n unless they also affect (G_n, m_n). For every i with $0 \leqq i \leqq n$, there should be an embedding $\gamma_i: (G_i, m_i) \to (\Gamma_i, \mu_i)$, with which we can describe Γ_i as the result of gluing G_i to an invariant graph Σ that specifies the context of G_0 as embedded in Γ_0 by γ_0. The invariance of Σ (it is the same for all i) and the details of the gluing construction will lead to the desired conclusion that intermediate steps can only affect (Γ_n, μ_n) insofar as they affect (G_n, m_n). To formulate this intuition precisely requires some ingenuity, but the effort buys the ability to draw conclusions about infinite sets of large graphs from calculations with a few small graphs. A typical application is in § 4.

The mathematical notion of gluing here is of course the pushout construction. The interface between G_i and Σ in the desired pushout to Γ_i will be denoted S and will be a subgraph of G_0 without arcs. The nodes of S will be called "boundary" nodes, and we will need assurances that these nodes persist throughout the given derivation. Consider a derivation via fast productions

(3.1.1) $$(G_0, m_0) \Rightarrow (G_1, m_1) \Rightarrow \cdots \Rightarrow (G_n, m_n) \quad \text{via } (p_1, \cdots, p_n),$$

a colored graph (Γ_0, μ_0), and a morphism $\gamma_0: (G_0, m_0) \to (\Gamma_0, \mu_0)$. The *residue* map $\mathbf{r}_i$ from certain nodes of G_0 to nodes of G_i is defined inductively by $\mathbf{r}_0 z = z$ and

(3.1.2) $$\mathbf{r}_j z = c_{2j} c_{1j}^{-1} \mathbf{r}_{j-1} z \quad \text{for } 1 \leqq j \leqq n,$$

where c_{1j} and c_{2j} are from the step via p_j in (3.1.1). Thus $\mathbf{r}_j z$ is only defined if $\mathbf{r}_{j-1} z$ is defined and is in $c_{1j} D_j$. Let the *persistent* nodes of G_0 be those z with $\mathbf{r}_n z$ defined. On the other hand, a node z in G_0 is a *boundary* node iff either

(3.1.3) $$\gamma_0 z \text{ has an inarc or outarc in } \Gamma_0 \text{ that is not in } \gamma_0 G_0$$

or there is a node z' in G_0 such that

(3.1.4) $$\gamma_0 z = \gamma_0 z' \quad \text{and} \quad z \neq z'.$$

With these notations the hypotheses of the embedding theorem can be stated briefly.

THEOREM 3.2. *Let there be a derivation via fast productions*

(1) $$(G_0, m_0) \Rightarrow (G_1, m_1) \Rightarrow \cdots \Rightarrow (G_n, m_n) \quad \textit{via } (p_1, \cdots, p_n).$$

Suppose there is a colored graph (Γ_0, μ_0) *and a morphism* $\gamma_0: (G_0, m_0) \to (\Gamma_0, \mu_0)$ *such that* (*where residues of nodes, persistent nodes, and boundary nodes are as in* (3.1))

(2) $$\gamma_0 \textit{ is injective on arcs};$$

(3) $$\textit{all boundary nodes are persistent};$$

(4) $$(\forall z, z' \textit{ nodes in } G_0)[\gamma_0 z = \gamma_0 z' \textit{ implies } (\forall j)(m_j \mathbf{r}_j z = m_j \mathbf{r}_j z')].$$

Then there are colored graphs (Γ_j, μ_j) *and a natural derivation*

(5) $$(\Gamma_0, \mu_0) \Rightarrow (\Gamma_1, \mu_1) \Rightarrow \cdots \Rightarrow (\Gamma_n, \mu_n) \quad \textit{via } (p_1, \cdots, p_n).$$ □

See [8, Thm. 3.5] for the proof, which also formalizes our intuitive remarks on Γ_i as the result of gluing G_i and an invariant context Σ together. Specifically, the proof has the following corollary.

COROLLARY 3.3. *As in Theorem* 3.2, *there are graphs* S *and* Σ *and morphisms* $f_0: S \to G_0$ *and* $\sigma: S \to \Sigma$, *depending only on* γ_0, *such that for every* i *with* $0 \leqq i \leqq n$ *there is* *a pushout*

(1)
$$\begin{tikzcd}
(G_i, m_i) \arrow[d, "\gamma_i"'] & (S, n_i) \arrow[l, "f_i"'] \arrow[d, "\sigma"] \\
(\Gamma_i, \mu_i) & (\Sigma, \nu_i) \arrow[l, "\phi_i"]
\end{tikzcd}$$

with colorings n_i and ν_i that depend only on γ_0, μ_0 and on $m_i\mathbf{r}_i$. There are no arcs in S and f_i is just $\mathbf{r}_i f_0$ on nodes. □

Theorem 3.2 and Corollary 3.3 are essentially special cases of the embedding theorem for derivations in a category STRUCT that has GRAPHS$[C]$ as a subcategory. The more general theorem and some other properties of STRUCT appear in [5]. Because the pushout in Corollary 3.3(1) is determined (up to isomorphism) by f_i and σ, we have a further corollary when two derivations that reach the same place are embedded in larger derivations.

COROLLARY 3.4. *As in Theorem* 3.2, *suppose there is also another derivation*

$$(G_0, m_0) = (G_0^{\#}, m_0^{\#}) \Rightarrow (G_1^{\#}, m_1^{\#}) \Rightarrow \cdots \Rightarrow (G_{n^{\#}}^{\#}, m_{n^{\#}}^{\#}) \quad \textit{via} \ (p_1^{\#}, \cdots, p_{n^{\#}}^{\#})$$

satisfying the same hypotheses. Suppose $\mathbf{r}_{n^{\#}}^{\#} = \mathbf{r}_n$ and $(G_{n^{\#}}^{\#}, m_{n^{\#}}^{\#}) = (G_n, m_n)$. Then a single colored graph (Γ_n, μ_n) has derivations

$$(\Gamma_0, \mu_0) \Rightarrow (\Gamma_n, \mu_\nu) \quad \textit{via} \ (p_1, \cdots, p_n) \quad \textit{and} \quad (\Gamma_0, \mu_0) \Rightarrow (\Gamma_n, \mu_n) \quad \textit{via} \ (p_1^{\#}, \cdots, p_{n^{\#}}^{\#}).$$
□

Suppose a colored graph $\mathbf{G} = (G, m_G)$ can be changed in two ways, as indicated by derivations $\mathbf{G} \Rightarrow \mathbf{H}$ via rp and $\mathbf{G} \Rightarrow \mathbf{H}^{\#}$ via $r^{\#}p^{\#}$. Instead of having to decide which change we prefer, perhaps we can make both changes. Perhaps p can still be applied to $\mathbf{H}^{\#}$ and $p^{\#}$ can still be applied to $\mathbf{H}$, with no need to choose between the order $(p, p^{\#})$ and the order $(p^{\#}, p)$ in applying the two productions in sequence to $\mathbf{G}$.

DEFINITION 3.5. Derivations $\mathbf{G} \Rightarrow \mathbf{H}$ via rp and $\mathbf{G} \Rightarrow \mathbf{H}^{\#}$ via $r^{\#}p^{\#}$ *commute* iff there are recolorings ρ, $\rho^{\#}$ and a colored graph $\mathbf{X}$ such that $\mathbf{H} \Rightarrow \mathbf{X}$ via $\rho^{\#}p^{\#}$ and $\boldsymbol{H}^{\#} \Rightarrow \boldsymbol{X}$ via ρp.

Weak but complicated sufficient conditions for commutativity are presented in [7, § 4]. Here we will consider stronger but simpler conditions that are still weak enough to hold in many situations. The *proper* productions defined by four conditions (5.6.1)–(5.6.4) in [7] will be helpful. The last of the conditions is made trivially true by (2.6) here, so we restate only the first three conditions here for ease of reference:

(3.6.1) $(\forall x, y \text{ in } B_1)(m_1 x = m_1 y \text{ in } C_{\text{var}} \text{ implies } x = y)$;

(3.6.2) $(\forall y \text{ in } B_2)(m_2 y \text{ in } C_{\text{var}} \text{ implies } y \text{ in } b_2 K)$;

(3.6.3) $(\forall x \text{ in } K)(m_1 b_1 x \text{ or } m_2 b_2 x \text{ in } C_{\text{var}} \text{ implies } m_1 b_1 x = m_2 b_2 x)$.

A *biproper* production satisfies these three conditions and their mirror images, with 1 and 2 subscripts reversed. Indirect productions $p_{\text{ind}}(c)$ are biproper, as are all the productions in [7, § 7]. Productions with only fixed colors are trivially biproper. Theorem 4.5 in the next section will deal with biproper productions because it will need the mirror image of (3.6.2) as well as the following general theorem about commutativity of derivations via proper productions.

THEOREM 3.7. [7, Thm. 5.9]. *Consider proper productions p, $p^{\#}$ and natural derivations* $\mathbf{G} \Rightarrow \mathbf{H}$ *via* rp *and* $\mathbf{G} \Rightarrow \mathbf{H}^{\#}$ *via* $r^{\#}p^{\#}$. *Suppose that*

(1) $$gB_1 \cap g^{\#}B_1^{\#} \subseteq gb_1K \cap g^{\#}b_1^{\#}K^{\#}.$$

Suppose that all x *in* K *and* $x^{\#}$ *in* $K^{\#}$ *such that*

(2) $$gb_1x = g^{\#}b_1^{\#}x^{\#} \quad and \quad m_2b_2x \in C_{\text{fix}} \quad and \quad m_2^{\#}b_2^{\#}x^{\#} \in C_{\text{fix}}$$

have

(3) $$m_1b_1x = m_2b_2x \quad and \quad m_1^{\#}b_1^{\#}x^{\#} = m_2^{\#}b_2^{\#}x^{\#}.$$

Then the derivations $\mathbf{G} \Rightarrow \mathbf{H}$ *and* $\mathbf{G} \Rightarrow \mathbf{H}^{\#}$ *commute.* □

Another property shared by indirect productions and all productions in [7, § 7] will be helpful in the next two sections. A production p is *rooted in* a node r in K iff

(3.8.1) every node in B_1 is reachable from b_1r;

(3.8.2) $m_1b_1x = m_2b_2x$ for all nodes x in K with $x \neq r$;

(3.8.3) $m_1b_1x = m_2b_2x$ for all arcs x in K with $s_Kx \neq r$.

Apart from the special node r and its outarcs, all items in K have the same colors m_1b_1 in B_1 that they have in B_2. Applying p may delete some items and add others, but it can only change the colors the root node and its outarcs. The production p is *rooted* iff it is rooted in some node of K.

Finally, we will use some elementary properties of pushouts in GRAPHS and many other categories. Lemma 3.9 below is valid for any category [14, Exercise 21E]. Lemma 3.10 is valid for any category that *has pushouts*: given two morphisms with a common domain (such as b and d in Fig. 3.1), we can always form a pushout incorporating these

$$\begin{tikzcd}
K \arrow[rrrr, "a"] \arrow[dddddd, "b"'] \arrow[ddrr, "d"'] & & & & E \arrow[dddddd, "\beta"] \\
& & (3) & & \\
& & D \arrow[uurr, "\delta"'] \arrow[dd, "c"] & & \\
& (2) & & (1) & \\
& & G \arrow[ddrr, "\phi"] & & \\
& & (4) & & \\
B \arrow[rrrr, "\alpha"'] \arrow[uurr, "g"] & & & & H
\end{tikzcd}$$

FIG. 3.1. *Diagram for Lemma* 3.9 *and Lemma* 3.10 *on combining pushouts.*

$$\begin{tikzcd}
K \arrow[rrr, "b"] \arrow[ddd, "d"'] & & & B \arrow[ddd, "h"] \\
& K' \arrow[ul] \arrow[r, "b'"] \arrow[d, "d'"'] & B' \arrow[ur] \arrow[d, "h'"] & \\
& D' \arrow[dl] \arrow[r, "c'"'] & H' \arrow[dr] & \\
D \arrow[rrr, "c"'] & & & H
\end{tikzcd}$$

FIG. 3.2. *Diagram for Lemma* 3.11 *on restricting pushouts.*

morphisms (such as the subdiagram (2) in Fig. 3.1). All categories considered in this paper have pushouts. Both lemmas use only the universal property of pushouts (Appendix B). In Lemma 3.11 the universal property interacts with special considerations for the category SETS whose objects are sets and whose morphisms are total functions.

LEMMA 3.9. *Let Fig.* 3.1 *be a commutative diagram in* GRAPHS *such that subdiagram* (2) *is a pushout. Then subdiagram* (1) *is a pushout iff the outer square is a pushout.* □

LEMMA 3.10. *Let the outer square in Fig.* 3.1 *be a pushout in* GRAPHS *and let Fig.* 3.1(3) *commute. Then there are G, c, g, ϕ such that the entire diagram commutes and subdiagrams* (1) *and* (2) *are pushouts.* □

LEMMA 3.11. *Let Fig.* 3.2 *be a commutative diagram in* GRAPHS[C] *such that the outer square is a pushout and all diagonal arrows are inclusions of subobjects. Suppose $B' = h^{-1}(H')$ and $D' = c^{-1}(H')$ and $K' = d^{-1}(D') = b^{-1}(B')$. Then the inner square is a pushout in* GRAPHS[C].

Proof. Because a commutative diagram in GRAPHS[C] is a pushout iff the corresponding diagrams in SETS for nodes and arcs separately are pushouts, it will suffice to prove this in SETS. Suppose β': $B' \to X$ and $D' \to X$ with $\beta' b' = \delta' d'$. If $X = \emptyset$ then $B' = D' = \emptyset$, so $H' = \emptyset$ and we have a trivial pushout. Therefore we may assume $X \neq \emptyset$. Choose $\xi \in X$. Let β: $B \to X$ with $\beta y = \beta' y$ if $y \in B'$ and $\beta y = \xi$ otherwise. Let δ: $D \to X$ similarly. Direct calculation (using the hypothesis on K') shows that $\beta b = \delta d$, so there is a unique η: $H \to X$ such that $\eta h = \beta$ and $\eta c = \delta$. Let η' be the restriction of η to H'. Then $\eta' h'$ is the restriction of β to B', which is β'. Similarly, $\eta' c'$ is the restriction of δ to D', which is δ'. Uniqueness of η' follows from uniqueness of η and the hypotheses on B' and D'. □

4. Indirection. For each fixed arc color c the production $p_{\text{ind}}(c)$ shown in Fig. 2.2 uses variable node colors u, v and fixed node color I. As in § 2, a *record structure* is a colored graph such that any node colored I has a unique outarc, and this arc is colored *ind*. Let y be the arc colored c in Fig. 2.2, so that applying the rule $(p_{\text{ind}}(c), y)$ at $z = gy$ in (G, m_G) has the effect of following the indirect c-pointer from $\mathbf{s}z$ and changing the pointer field at $\mathbf{s}z$ so as to point directly to the node in G corresponding to the node colored v in Fig. 2.2. Given any record structures $\mathbf{G} = (G, m_G)$ and $\mathbf{H} = (H, m_H)$, let $\mathbf{G} \gg_{\text{ind}} \mathbf{H}$ iff there is a fixed arc color c, a recoloring r in R, and an arc z in G such that

$$(4.1) \qquad \mathbf{G} \Rightarrow \mathbf{H} \quad \text{via } rp_{\text{ind}}(c) \text{ at } z.$$

Given any family $\mathscr{F}$ of structures for which indirection makes sense, we can try to establish the Church–Rosser property (1.1) for the system $(\mathscr{F}, \gg_{\text{ind}})$, no matter what other properties $\mathscr{F}$ may have. The family $\mathscr{F}$ should be closed under $\gg_{\text{ind}}$:

$$(4.2.1) \qquad (\mathbf{G} \text{ is in } \mathscr{F} \text{ and } \mathbf{G} \gg_{\text{ind}} \mathbf{H}) \quad \text{implies} \quad \mathbf{H} \text{ is in } \mathscr{F}.$$

There should be nothing analogous to the tight loop (LABEL: **goto** LABEL) in programming: for all $\mathbf{G}$ in $\mathscr{F}$ and all arcs x in G,

$$(4.2.2) \qquad m_G\mathbf{s}_G x = I \quad \text{implies} \quad \mathbf{t}_G x \neq \mathbf{s}_G x.$$

In particular, (4.2.2) holds if all graphs in $\mathscr{F}$ are acyclic. Families that satisfy (4.2) are said to *allow indirection.*

LEMMA 4.3. *If $\mathscr{F}$ allows indirection then every member of $\mathscr{F}$ has a unique normal form in* $(\mathscr{F}, \gg_{\text{ind}})$.

Proof. By induction on cycle lengths, (4.2) implies that no graph in $\mathscr{F}$ has a cycle whose nodes are colored I. For each $\mathbf{G}$ in $\mathscr{F}$ there are finitely many paths having only nodes colored I as targets of arcs in the path, so there is a finite nonnegative "weight"

$$(1) \qquad w\mathbf{G} = \sum_{\text{all such paths } \theta} (\text{length of } \theta)$$

such that, for all $\mathbf{G}$, $\mathbf{H}$ in $\mathscr{F}$,

$$(2) \qquad \mathbf{G} \gg_{\text{ind}} \mathbf{H} \quad \text{implies} \quad w\mathbf{G} > w\mathbf{H}.$$

This implies the existence of normal forms. As is well-known [15, Lemma 4] and easily demonstrated by induction on weights, (2) also implies that the Church–Rosser property will follow from the usually weaker property

$$(3) \qquad \begin{aligned} &(\forall \mathbf{G}, \mathbf{H}, \mathbf{H}^{\#} \text{ in } \mathscr{F}) \\ &[(\mathbf{G} \gg_{\text{ind}} \mathbf{H} \;\&\; \mathbf{G} \gg_{\text{ind}} \mathbf{H}^{\#}) \text{ implies } (\exists \mathbf{X})(\mathbf{H} \gg^{*}_{\text{ind}} \mathbf{X} \;\&\; \mathbf{H}^{\#} \gg^{*}_{\text{ind}} \mathbf{X})]. \end{aligned}$$

We prove (3) by assuming that $\mathbf{G}, \mathbf{H}, \mathbf{H}^{\#}$ are as above and deriving the existence of an appropriate $\mathbf{X}$. Consider the derivations (4.1) for $\mathbf{G} \gg_{\text{ind}} \mathbf{H}$ and (4.1$^{\#}$) for $\mathbf{G} \gg_{\text{ind}} \mathbf{H}^{\#}$. If $z = z^{\#}$ then $\mathbf{H} = \mathbf{H}^{\#}$ and we may let $\mathbf{X}$ be $\mathbf{H}$ also. (As usual, we say $\mathbf{H} = \mathbf{H}^{\#}$ in situations where all we really have is that $\mathbf{H}$ is isomorphic to $\mathbf{H}^{\#}$, provided that the distinction between equality and isomorphism is only a nuisance and conceals no real difficulty.) We may assume $z \neq z^{\#}$. If $gB_1 \cap g^{\#}B_1^{\#} \subseteq gb_1K \cap g^{\#}b_1^{\#}K$ then Theorem 3.7 provides the desired $\mathbf{X} = (X, m_X)$. Otherwise we may assume $z^{\#} = gb_1x_3$ where x_3 is the unique arc in K, so that gB_1 and $g^{\#}B_1^{\#}$ overlap as shown in Fig. 4.1 (top). Note that $c^{\#}$ here is *ind*. The nodes colored ru and $r^{\#}v^{\#}$ in the picture might actually be one node in G, but (4.2.2) requires that all other seemingly distinct nodes in the picture be truly distinct in the graph G. The relevant parts of $\mathbf{H}$ and $\mathbf{H}^{\#}$ are pictured in Fig. 4.1 (left) and Fig. 4.1 (right). As suggested by Fig. 4.1 (bottom), there is a colored graph $\mathbf{X}$ with $\mathbf{H} \gg_{\text{ind}} \gg_{\text{ind}} \mathbf{X}$ by applying $p_{\text{ind}}(ind)$ and $p_{\text{ind}}(c)$ and with $\mathbf{H}^{\#} \gg_{\text{ind}} \mathbf{X}$ by applying $p_{\text{ind}}(c)$. To *show* this we apply Corollary 3.4 to embed Fig. 4.1 into derivations

$\mathbf{G} \gg_{\text{ind}} \mathbf{H} \gg_{\text{ind}} \gg_{\text{ind}} \mathbf{X}$ and $\mathbf{G} \gg_{\text{ind}} \mathbf{H}^{\#} \gg_{\text{ind}} \mathbf{X}$. Hypothesis (4) in Theorem 3.2 follows from the fact that $m_{1i}b_{1i} = m_{2i}b_{2i}$ for all the productions p_i. □

As in the diagrammatic proofs in category theory or in [20, § 3], use of Fig. 4.1 enabled us to avoid a tedious procession of stipulations like "let z_8 be · · ·" and to display the intuitive naturalness of our proof. But no geometric intuition or tacit assumptions were smuggled into the reasoning, and the relevance of the intuitive picture to the mathematical problem was rigorously established by Theorem 3.2.

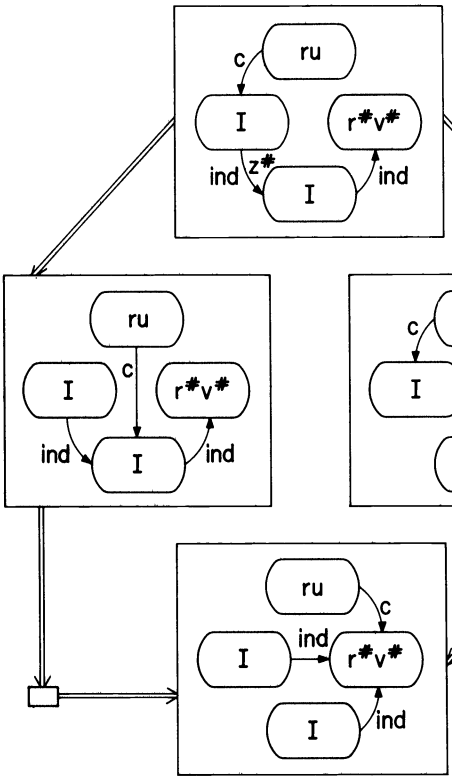


FIG. 4.1. *Difficult case in the proof of Lemma* 4.3.

The good behavior of $\gg_{\text{ind}}$ is not in itself very interesting. Given some other well-behaved relation $\gg_1$ on $\mathcal{F}$, perhaps induced by productions that add I nodes to record structures, we would like to show that the union $\gg$, with $\mathbf{G} \gg \mathbf{H}$ iff ($\mathbf{G} \gg_1 \mathbf{H}$ or $\mathbf{G} \gg_{\text{ind}} \mathbf{H}$), still behaves well. Some mild restrictions on $\gg_1$ will be needed. We are concerned with situations where $\gg_1$ is induced by a set $\mathcal{P}$ of fast productions. Productions that delete (or change the colors of) I nodes or *ind* arcs could destroy opportunities to apply $p_{\text{ind}}(c)$. On the other hand, applying indirect productions could destroy opportunities to apply productions in $\mathcal{P}$ such that (B_1, m_1) includes an I node with an inarc. We will therefore restrict $\mathcal{P}$ to contain only productions wherein no node in B_1 is colored I and no arc in B_1 is colored *ind*. Because variables in (B_1, m_1) may be recolored to I or *ind*, there is still some need for caution, and the full hypothesis of the theorem is somewhat elaborate. Before stating the theorem we state an easy general Church–Rosser lemma that will be used.

LEMMA 4.4. *Let $\mathscr{F} \subseteq \mathscr{B}$ be closed under a relation $\gg$ on $\mathscr{B}$. Suppose $\gg_4$ is a relation on $\mathscr{B}$ such that*

(1) $$(\gg_4) \subseteq (\gg^*);$$

(2) $$(\forall \mathbf{G}, \mathbf{H} \textit{ in } \mathscr{F})[\mathbf{G} \gg^* \mathbf{H} \textit{ implies } (\exists \mathbf{L})(\mathbf{G} \gg_4^* \mathbf{L} \,\&\, \mathbf{H} \gg^* \mathbf{L})];$$

(3) $$(\mathscr{F}, \gg_4) \textit{ is Church–Rosser.}$$

Then $(\mathscr{F}, \gg)$ is Church–Rosser. □

Lemma 4.4 has a well known special case [20, Lemma 3.4] with many uses. Intuitively, consider members of $\mathscr{F}$ to be possible states of a computer system. Some transitions $\mathbf{G} \gg \mathbf{H}$ introduce user *requests* while others *service* these requests. Let $\mathbf{G} \gg_4 \mathbf{H}$ if there is a sequence $\mathbf{G} = \mathbf{X}_0 \gg \mathbf{X}_1 \gg \cdots \gg \mathbf{X}_n = \mathbf{H}$ where the first transition introduces at most one request and the remaining $n-1$ transitions do everything necessary to service it along with any outstanding requests in $\mathbf{G}$. The system $(\mathscr{F}, \gg)$ is difficult to analyze because arrivals of new requests are interleaved with actions to service old requests. The system $(\mathscr{F}, \gg_4)$ is simpler because each request is fully serviced before another request arrives. It may well be Church–Rosser, as required by Lemma 4.4(3). Lemma 4.4(2) is one way to express the idea that $(\mathscr{F}, \gg)$ is free of *deadlock*: no matter how actions are interleaved in $\mathbf{G} \gg^* \mathbf{H}$, the system can continue (with $\mathbf{H} \gg^* \mathbf{L}$) so as to reach a state wherein it looks as if each request was fully serviced before the next one came in (with $\mathbf{G} \gg_4^* \mathbf{L}$). Lemma 4.4(1) is trivially true for $\gg_4$ as specified in this intuitive discussion. By adding it to the other conditions we obtain a precise lemma without needing precise concepts of "request" or "service" or "deadlock". The lemma can now be applied to many different situations.

THEOREM 4.5. *Let $\mathscr{F}$ be a family of acyclic record structures that allows indirection. Let $\gg_1$ be a relation on colored graphs such that, for some set $\mathscr{P}$ of rooted biproper fast productions wherein no node in B_1 is colored I and no arc in B_1 is colored ind or in C_{var},*

(1) $$\mathbf{G} \gg_1 \mathbf{H} \quad \textit{iff} \quad (\exists p \textit{ in } \mathscr{P})(\exists r \textit{ in } R)(\mathbf{G} \Rightarrow \mathbf{H} \textit{ via } rp);$$

(2) $$(\mathscr{F}, \gg_1) \textit{ is strongly Church–Rosser.}$$

Then $(\mathscr{F}, \gg)$ is Church–Rosser, where $\gg$ is the union of $\gg_1$ and $\gg_{\text{ind}}$. Moreover, $(\mathscr{F}, \gg_4)$ is strongly Church–Rosser, where

(3) $$\mathbf{G} \gg_2 \mathbf{H} \quad \textit{iff} \quad \mathbf{G} \gg_1^{=} \mathbf{H};$$

(4) $$\mathbf{G} \gg_3 \mathbf{H} \quad \textit{iff} \quad [\mathbf{G} \textit{ has normal form } \mathbf{H} \textit{ in } (\mathscr{F}, \gg_{\text{ind}})];$$

(5) $$\mathbf{G} \gg_4 \mathbf{H} \quad \textit{iff} \quad (\exists \mathbf{X})(\mathbf{G} \gg_2 \mathbf{X} \gg_3 \mathbf{H}).$$

Proof. We will apply Lemma 4.4. Intuitively, think of $\mathbf{G} \Rightarrow \mathbf{X}$ via rp as introducing a *single request*, that all indirect pointers created by applying rp be retargeted so as to point directly to interesting nodes rather than nodes colored I. Think of $\mathbf{X} \gg_{\text{ind}}^* \mathbf{H}$ as *servicing* requests, with complete servicing when $\mathbf{H}$ has no more indirect pointers. Assume for the moment that

(6) $$(\mathbf{G} \gg_2 \mathbf{H} \,\&\, \mathbf{G} \gg_3 \mathbf{H}^{\#}) \quad \text{implies} \quad (\exists \mathbf{W}, \mathbf{X})(\mathbf{H} \gg_3 \mathbf{X} \,\&\, \mathbf{H}^{\#} \gg_2 \mathbf{W} \gg_3 \mathbf{X}).$$

Figure 4.2 is an example of what (6) says when p corresponds to the LISP rule that $\mathbf{cdr}(\mathbf{cons}(u, v)) = v$. The figure is also a counterexample to the stronger assertion one might be tempted to use, with $\mathbf{H}^{\#} \gg_2 \mathbf{X}$ instead of $\mathbf{H}^{\#} \gg_2 \mathbf{W} \gg_3 \mathbf{X}$. The assumption (6) is diagrammed in the style of [20, § 3] on the left in Fig. 4.3. The diagram on the right in Fig. 4.3 may be verified as follows: (a) because normal forms exist and are unique in

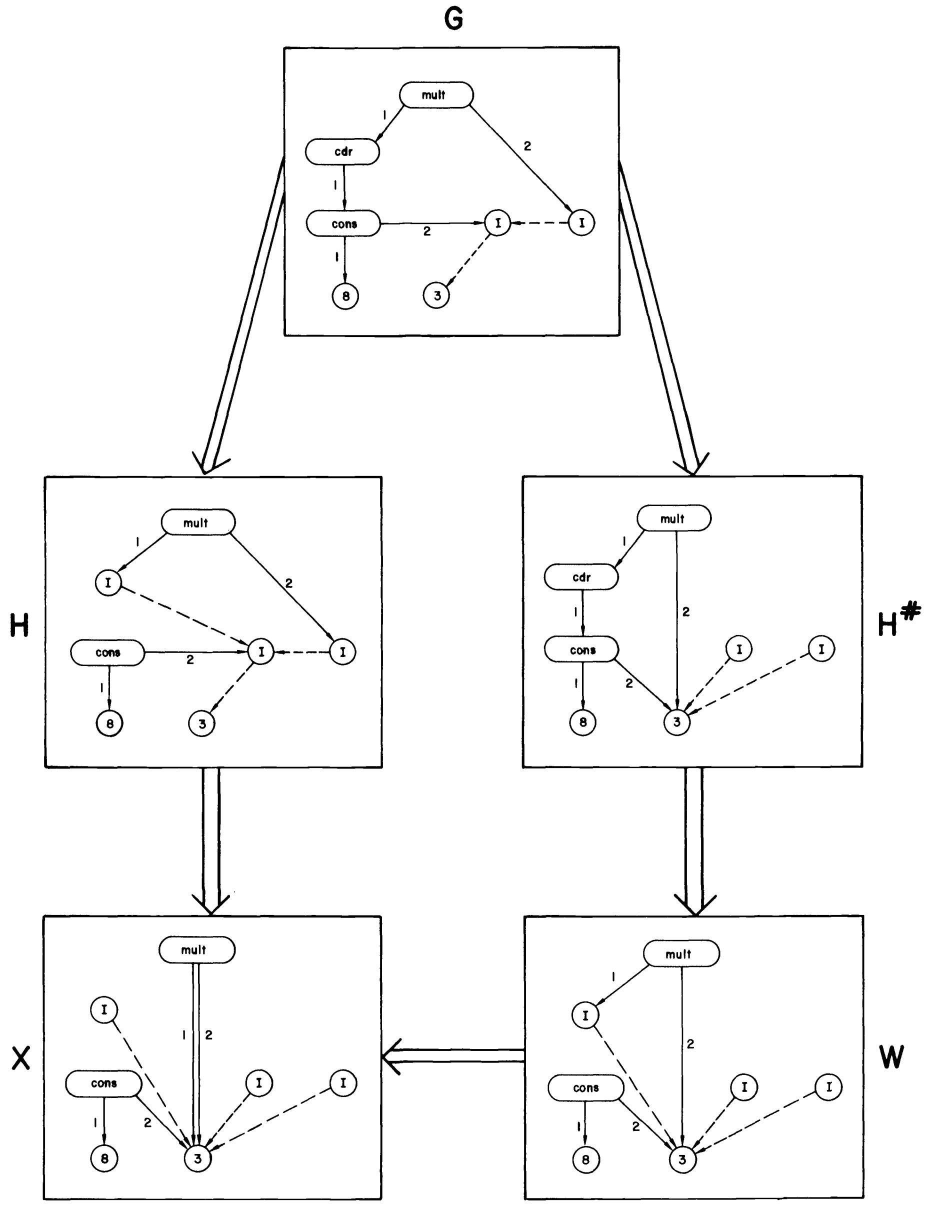


FIG. 4.2. *Example of the assumption* (6) *in proving Theorem* 4.5. *This is a pure* LISP *metaexpression that will evaluate to* 9.

$(\mathscr{F}, \gg_{\text{ind}})$, then (b) because of (6), and then (c) because normal forms are unique in $(\mathscr{F}, \gg_{\text{ind}})$. Let $\circ$ be composition of relations, so that $(\gg_4) = (\gg_2)\circ(\gg_3)$. The diagram just verified implies that

$$(\gg_{\text{ind}})\circ(\gg_4) \subseteq (\gg_3)\circ(\gg_4) \subseteq (\gg_4^*).$$

By $(\gg_1) \subseteq (\gg_2)$, a similar argument with two uses of (6) shows that

$$(\gg_1)\circ(\gg_4) \subseteq (\gg_2)\circ(\gg_4) \subseteq (\gg_3)\circ(\gg_4)\circ(\gg_4) \subseteq (\gg_4^*).$$

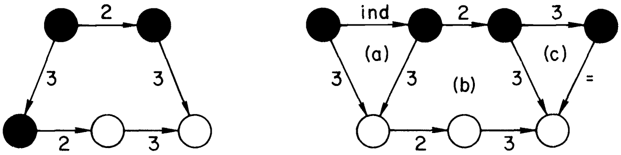


FIG 4.3. *The diagram on the left is equivalent to Theorem* 4.5(6). *In general, filled circles correspond to* $x_1, x_2, \cdots$ *and open circles correspond to* $y_1, y_2, \cdots$ *in a formula* $(\forall x_1, x_2, \cdots)[HYP$ *implies* $(\exists y_1, y_2, \cdots)$ $(CON)]$, *where HYP and CON are conjunctions of formulas* α ⌑ β *such that* α, β *are in* $\{x_1, y_1, x_2, y_2, \cdots\}$ *and* ⌑ *is in* $\{\gg_2, \gg_3, \gg_{\mathrm{ind}}, =, \cdots\}$. *The diagram on the right is used in proving the theorem.*

But $\gg$ is the union of $\gg_{\mathrm{ind}}$ and $\gg_1$, so

$$(7) \qquad (\gg)\circ(\gg_4) \subseteq (\gg_4^*).$$

We can now verify the three hypotheses of Lemma 4.4. Lemma 4.4(1) is trivial. Lemma 4.4(2) will be verified by induction on n in $\mathbf{G} \gg^n \mathbf{H}$. For $n = 0$ it suffices to let $\mathbf{L}$ be $\mathbf{G}$. To pass from n to $n+1$, suppose $\mathbf{G} \gg \mathbf{X} \gg^n \mathbf{H}$. By the induction hypothesis there is $\mathbf{Y}$ with $\mathbf{X} \gg_4^* \mathbf{Y}$ and $\mathbf{H} \gg^* \mathbf{Y}$. Let $\mathbf{L}$ be the normal form of $\mathbf{Y}$ in $(\mathscr{F}, \gg_{\mathrm{ind}})$, so that $\mathbf{Y} \gg_4 \mathbf{L}$. Lemma 4.4(1) implies $\mathbf{Y} \gg^* \mathbf{L}$, and so $\mathbf{H} \gg^* \mathbf{L}$. We must show that $\mathbf{G} \gg_4^* \mathbf{L}$. But $\mathbf{G} \gg \mathbf{X} \gg_4^* \mathbf{Y} \gg_4 \mathbf{L}$, so (7) implies $\mathbf{G} \gg_4^* \mathbf{L}$. For Lemma 4.4(3) we verify Fig. 4.4 (which implies that $(\mathscr{F}, \gg_4)$ is strongly Church–Rosser), with (a) because $(\mathscr{F}, \gg_1)$ is strongly Church–Rosser and then (b), (c) because of (6) and uniqueness of normal forms in $(\mathscr{F}, \gg_{\mathrm{ind}})$. By Lemma 4.4, $(\mathscr{F}, \gg)$ is Church–Rosser.

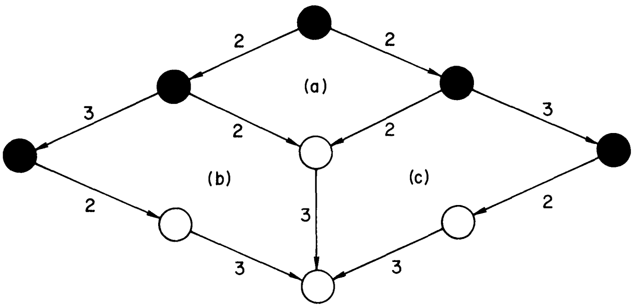


FIG. 4.4. *Verification of Lemma* 4.4(3) *in the proof of Theorem* 4.5.

All that remains is to prove (6). Suppose $\mathbf{G} \gg_2 \mathbf{H}$ and $\mathbf{G} \gg_3 \mathbf{H}^{\#}$. If $\mathbf{G} = \mathbf{H}$ then we may let $\mathbf{W}$ and $\mathbf{X}$ be $\mathbf{H}^{\#}$, so we may assume $\mathbf{G} \neq \mathbf{H}$. Therefore there is a direct derivation

$$(8) \qquad \mathbf{G} \Rightarrow \mathbf{H} \quad \text{via } rp \text{ based on } g$$

for some p in $\mathscr{P}$, r in R, and $g\colon B_1 \to G$. We want to construct a similar derivation

$$(9) \qquad \mathbf{H}^{\#} \Rightarrow \mathbf{W} \quad \text{via } r^{\#}p \text{ based on } g^{\#}$$

such that $\mathbf{W}$ and $\mathbf{H}$ have the same normal form $\mathbf{X}$ in $(\mathscr{F}, \gg_{\mathrm{ind}})$. With colorings suppressed for readability, Fig. 4.5 summarizes how (9) will be obtained. The pushouts Fig. 4.5(1) and Fig. 4.5(2) in GRAPHS are from (8). The other squares are also pushouts, but in a category TARGRAPHS that is like GRAPHS but has only the target operation mapping arcs to nodes. We may consider G, D, H to be objects in this category by simply forgetting about sources of arcs. Categorical ideas and results help us see the forest despite all the trees, while the hypotheses of categorical lemmas are obtained by

$$\begin{tikzcd}
B_1 \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "b_1"'] \arrow[r, "b_2"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & B_2 \arrow[d, "h"] \\
G \arrow[d, dashed, "\phi"'] \arrow[dr, phantom, "(3)"] & D \arrow[l, "c_1"'] \arrow[r, "c_2"] \arrow[d, dashed, "\delta"] \arrow[dr, phantom, "(4)"] & H \arrow[d, dashed, "\psi"] \\
H^{\#} & D^{\#} \arrow[l, "c_1^{\#}"] \arrow[r, "c_2^{\#}"'] & W
\end{tikzcd}$$

FIG. 4.5. *Solid arrows are maps that preserve sources and targets, as are the compositions $g^{\#} = \phi g$, $d^{\#} = \delta d$, and $h^{\#} = \psi h$. Dashed arrows are maps that only preserve targets.*

calculations that involve reasoning about paths in graphs and inductive proofs of properties of recursively defined functions. The details appear as Appendix A. □

Instead of assuming that $(\mathcal{F}, \gg_1)$ is *strongly* Church–Rosser, we could assume that $\gg_1$ is induced by derivations wherein g is close to being injective. Specifically, if $gy = gy'$ then either $y = y'$ or both are colored in C_{var}. The corrected version of [8, Thm. 4.2] that is needed here appears in [9] for the case of color preserving productions. A proof for this case appears in [10]. The general case can probably be handled by essentially the same methods, but a formal statement of the alternative version of Theorem 4.5 is omitted here, pending resolution of all details.

5. Garbage collection. In any record structure there are a few records directly accessible to the external world. Often there is just one external record, the "root". The other records can only be reached by following paths from external records. To model this distinction we may assume that the fixed node colors have been classified as *external* or *internal.* The distinction has no effect on the preceding sections, unlike the use of triples (G, m_G, e_G), with e_G being the root, in the earlier version [8] of this paper. The distinction is important now because we wish to delete internal nodes that are not reachable from external nodes, where nodes are classified by their colors. Only a simple form of such *garbage collection* will be treated here. Given an internal fixed node color c_{in} and a string $\lambda = (\lambda_1 \cdots \lambda_N)$ of fixed arc colors we consider a production $p_{\text{gar}}(c_{\text{in}}, \lambda)$ as shown in Fig. 5.1. The rule $(p_{\text{gar}}(c_{\text{in}}, \lambda), y)$, where y is the node colored c_{in} in Fig. 5.1, may be used to delete a node colored c_{in} with outarcs colored $\lambda_1, \cdots, \lambda_N$ and with no inarcs. Let $\mathbf{G} \gg_{\text{gar}} \mathbf{H}$ iff there are an internal fixed node color c_{in}, a string λ of fixed arc colors, a recoloring r in R, and a node z in G such that

$$\text{(5.1)} \qquad \mathbf{G} \Rightarrow \mathbf{H} \quad \text{via } rp_{\text{gar}}(c_{\text{in}}, \lambda) \text{ at } z.$$

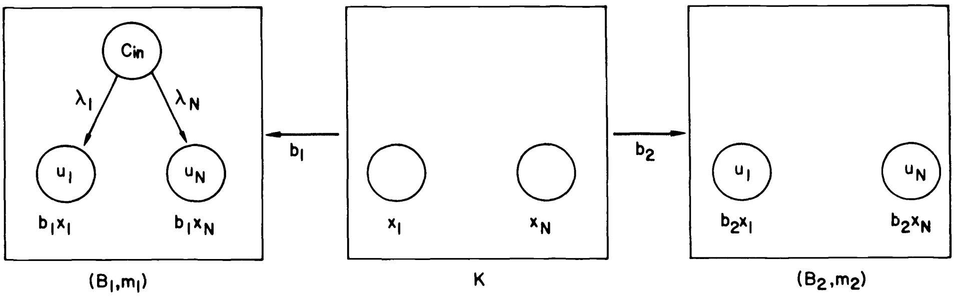


FIG. 5.1. *The node colored c_{in} in the production $p_{\text{gar}}(c_{\text{in}}, \lambda)$ is collected as garbage if it has no inarcs and has no other outarcs beside those colored $\lambda_1, \cdots, \lambda_N$. The drawing has $N = 2$.*

LEMMA 5.2. *If $\mathcal{F}$ is closed under garbage collection then every member of $\mathcal{F}$ has a unique normal form in $(\mathcal{F}, \gg_{\text{gar}})$.*

Proof. If $\mathbf{G} \gg_{\text{gar}} \mathbf{H}$ and $\mathbf{G} \gg_{\text{gar}} \mathbf{H}^{\#}$ and $\mathbf{H} \neq \mathbf{H}^{\#}$, then Theorem 3.7 provides $\mathbf{X}$ with $\mathbf{H} \gg_{\text{gar}} \mathbf{X}$ and $\mathbf{H}^{\#} \gg_{\text{gar}} \mathbf{X}$. This clearly implies the Church–Rosser property. Existence of normal forms follows from the fact that $\gg_{\text{gar}}$ decreases the size of a colored graph. □

It is easy to define a family $\mathcal{F}$ and relation $\gg_1$ on $\mathcal{F}$ such that $(\mathcal{F}, \gg_1)$ is Church–Rosser and $\mathcal{F}$ is closed under garbage collection, but the union system $(\mathcal{F}, \gg)$ is not Church–Rosser. Premature garbage collection may destroy opportunities to add arcs between nongarbage nodes. Considering the recent interest in collecting garbage in parallel with list processing [2], [3], [12], [13], [26], [29], we seek simple conditions such that the union will inherit the Church–Rosser property from $(\mathcal{F}, \gg_1)$. Let $\mathbf{s}_i$ and $\mathbf{t}_i$ be the source and target maps of B_i in a production p. Then p is *treelike* iff

$$\text{(5.3.1)} \qquad B_2 \text{ is acyclic and } m_2(B_2 - b_2K)_{\text{nodes}} \subseteq (\text{internal node colors})$$

and there is a node r in K such that

$$\text{(5.3.2)} \qquad p \text{ is rooted in } r;$$

$$\text{(5.3.3)} \qquad m_1b_1r \text{ is external iff } m_2b_2r \text{ is external}$$

and, for $i = 1, 2$ and for every arc z in B_i and node $x \neq r$ in K,

$$\text{(5.3.4)} \qquad b_ix = \mathbf{s}_iz \quad \text{implies} \quad (\exists y \text{ arc in } K)(b_iy = z).$$

All productions $p_{\text{ind}}(c')$ are treelike. So are all productions considered in [7, § 7].

LEMMA 5.4. *Let $(\mathcal{F}, \gg_1)$ and $(\mathcal{F}, \gg_2)$ be Church–Rosser systems such that, whenever $\mathbf{G} \gg_1 \mathbf{H}$ and $\mathbf{G} \gg_2 \mathbf{H}^{\#}$ in $\mathcal{F}$,*

$$\text{(1)} \qquad (\exists \mathbf{X})[\mathbf{H} \gg_2^* \mathbf{X} \textit{ and } (\mathbf{H}^{\#} \gg_1 \mathbf{X} \textit{ or } \mathbf{H}^{\#} = \mathbf{X})].$$

Then $(\mathcal{F}, \gg)$ is Church–Rosser, where $\gg$ is the union of $\gg_1$ and $\gg_2$.

Proof. Direct application of [20, Lemma 3.6] and then [20, Thm. 3.5]. □

THEOREM 5.5. *Let $\mathcal{F}$ be closed under $\gg_{\text{gar}}$ and let $\gg_1$ be a relation on colored graphs such that, for some set $\mathcal{P}$ of treelike proper fast productions wherein each b_1 is surjective on* *nodes,*

(1) $$\mathbf{G} \gg_1 \mathbf{H} \quad \textit{iff} \quad (\exists p \textit{ in } \mathcal{P})(\exists \rho \textit{ in } R)(\mathbf{G} \Rightarrow \mathbf{H} \textit{ via } \rho p)$$

(2) $$(\mathcal{F}, \gg_1) \textit{ is Church–Rosser.}$$

Then $(\mathcal{F}, \gg)$ *is Church–Rosser, where* $\gg$ *is the union of* $\gg_1$ *and* $\gg_{\text{gar}}$.

Proof. By (2), Lemma 5.2, and Lemma 5.4, it will suffice to show

(3) $$(\exists \mathbf{X})[\mathbf{H} \gg^*_{\text{gar}} \mathbf{X} \text{ and } (\mathbf{H}^\# \gg_1 \mathbf{X} \text{ or } \mathbf{H}^\# = \mathbf{X})]$$

under the assumption $\mathbf{G} \gg_1 \mathbf{H}$ and $\mathbf{G} \gg_{\text{gar}} \mathbf{H}^\#$. Let $\mathbf{G} \Rightarrow \mathbf{H}$ via ρp in (1) and let $\mathbf{G} \Rightarrow \mathbf{H}^\#$ via $\rho^\# p^\#$ in (5.1). There are two cases to consider. We begin with the easier one.

Case 1 [$z^\#$ is not in gB_1]. Then no arcs are in $gB_1 \cap g^\# B_1^\#$ and we have $gB_1 \cap g^\# B_1^\# \subseteq g^\# b_1^\# K^\#$. Consider any node $gy = g^\# y^\#$ in $gB_1 \cap g^\# B_1^\#$. We claim that gy is in gb_1K as well as $g^\# b_1^\# K^\#$. By $g^\# y^\# \neq z^\#$, some arc $x^\#$ in $B_1^\#$ has target $y^\#$. Therefore $g^\# x^\#$ is an arc of G with target $g^\# y^\#$ that is not in gB_1 because $gB_1 \cap g^\# B_1^\#$ lacks arcs. By Lemma 2.4(3) for the pushout to G in $\mathbf{G} \Rightarrow \mathbf{H}$, gy is in gb_1K. We have shown that

$$gB_1 \cap g^\# B_1^\# \subseteq gb_1K \cap g^\# b_1^\# K^\#,$$

which is the first hypothesis of Theorem 3.7. But $m_2^\# b_2^\# x^\#$ is in C_{var} for all $x^\#$ in $K^\#$, so the second hypothesis holds trivially. Theorem 3.7 yields $\mathbf{X}$ with $\mathbf{H} \gg_{\text{gar}} \mathbf{X}$ and $\mathbf{H}^\# \gg_1 \mathbf{X}$, so (3) holds.

Case 2 [$z^\#$ is in gB_1]. This is the difficult case because part of gB_1 is garbage that has already been collected when we try to apply p to $\mathbf{H}^\#$. Instead of applying p we will collect some garbage by a derivation $\mathbf{H} \gg^*_{\text{gar}} \mathbf{X}$, and then (3) will follow as soon as we show that $\mathbf{X}$ is (isomorphic to) $\mathbf{H}^\#$. The first step is to relate $z^\#$ to the node r from (5.3). Because $z^\#$ has no inarcs in G, (3.8.1) implies that $z^\# = gb_1r$ and $gy_1 \neq z^\#$ for all $y_1 \neq b_1r$ in B_1. It then follows that $dx \neq dr$ for all $x \neq r$ in K and hence that $hy_2 \neq hb_2r$ for all $y_2 \neq b_2r$ in B_2. For future reference we summarize these observations by saying that the derivation $\mathbf{G} \Rightarrow \mathbf{H}$ via ρp *treats r injectively*. Because $z^\#$ lacks inarcs in G, $gb_1r = z^\#$ implies that b_1r lacks inarcs in B_1 and dr lacks inarcs in D. This implies that r lacks inarcs in K, which implies by (5.3.4) that any inarc of b_2r has source in $B_2 - b_2K$. By (5.3.3) and the fact that $m_G z^\#$ is internal, we also know that $\rho m_1 b_1 r$ and $\rho m_2 b_2 r$ are internal. (We also use (3.6.3) if $m_1 b_1 r$ is in C_{var}.) For future reference we summarize these observations by saying that r is *almost garbage in* the production ρp, the "almost" being a concession to any inarcs of b_2r in B_2.

We will construct a sequence of derivations $\mathbf{G}[j] \Rightarrow \mathbf{H}[j]$ via $\rho p[j]$ for $j = 0, 1, \cdots, L$ with $L = |(B_2 - b_2K)_{\text{nodes}}| + 1$, wherein each graph in the jth derivation for $j > 0$ is a subgraph of the corresponding graph in the $(j-1)$th derivation and each morphism is a restriction of the corresponding morphism. For $j = 0$, the derivation $\mathbf{G}[j] \Rightarrow \mathbf{H}[j]$ via $\rho p[j]$ is the given $\mathbf{G} \Rightarrow \mathbf{H}$ via ρp. For $\mathbf{X} = \mathbf{H}[L]$ we will have $\mathbf{H} \gg^*_{\text{gar}} \mathbf{X}$ because $\mathbf{H}[j^{-1}] \gg_{\text{gar}} \mathbf{H}[j]$ for all $j > 0$. There will also be $j^\# \leqq L$ such that

(4) $$(\forall j < j^\#)(\mathbf{G}[j] = \mathbf{G}) \quad \text{and} \quad (\forall j \geqq j^\#)(\mathbf{G}[j] = \mathbf{H}^\#).$$

Finally, $\mathbf{X}$ will be isomorphic to $\mathbf{H}^\#$ because $\mathbf{H}[L]$ will be isomorphic to $\mathbf{G}[L]$.

Because B_2 is acyclic there is a listing of the nodes of B_2 such that every inarc of a node is an outarc of a previous node in the listing. Extracting the nodes in $N = (B_2 - b_2K)_{\text{nodes}} \cup \{b_2r\}$ from this listing, we get $(y_1, \cdots, y_L)$ such that, by (5.3.4) and the property of the original listing, every inarc of y_j is an outarc of y_i for $i < j$. Let $j^\#$ be such that $y_j = b_2r$ for $j = j^\#$. For each j with $1 \leqq j \leqq L$ the jth derivation is constructed from the $(j-1)$th derivation by collecting some garbage and restricting some morphisms. For all j we collect garbage in $B_2[j-1]$ and in $H[j-1]$. For $j=j^{\#}$ we also collect garbage in the other graphs. (Colors are just carried along.)

Consider the case $j \neq j^{\#}$ first. By induction on j and the property of the listing $(y_1, \cdots, y_L)$, we may assume that y_j is garbage in $B_2[j-1]$ even though it may have had inarcs in B_2. We remove y_j and its outarcs from $B_2[j-1]$ to form $B_2[j]$. We also remove hy_j and its outarcs from $H[j-1]$ to form $H[j]$, as is indeed possible by garbage collection because each inarc of hy_j in H is an outarc of hy_i for $i<j$. Because $B_2[j]=h^{-1}(H[j])$, we can restrict $h[j-1]$: $B_2[j-1] \rightarrow H[j-1]$ to define $h[j]$ from $B_2[j]$ to $H[j]$. Let $K[j]$ be $K[j-1]$ and let $D[j]$ be $D[j-1]$. Because $K[j]=b_2^{-1}(B_2[j])$, the (argument, value) pairs for $b_2[j-1]$ also define $b_2[j]$: $K[j] \rightarrow B_2[j]$. Similarly, $D[j]=c_2^{-1}(H[j])$ and we have $c_2[j]$: $D[j] \rightarrow H[j]$. Finally, let $d[j]=d[j-1]$. All hypotheses of Lemma 3.11 hold, so the right square in Fig. 2.3 is a pushout for j as well as for $j-1$. Leaving the left square unchanged, we have two pushouts that constitute the desired $\mathbf{G}[j] \Rightarrow \mathbf{H}[j]$ via $\rho p[j]$.

Now consider $j=j^{\#}$, so that $y_j=b_2r$ and is garbage in $B_2[j-1]$. Because r is almost garbage in ρp, we can form $B_i[j]$ for $i=1,2$ by collecting b_ir, $K[j]$ by collecting r, $D[j]$ by collecting dr, G[j] by collecting gb_1r and $H[j]$ by collecting hb_2r. Because $gb_1r=z^{\#}$, we do have (4). Because the original derivation treats r injectively, we do have $B_2[j]=h^{-1}(H[j])$ and $D[j]=c_2^{-1}(H[j])$ and $K[j]=b_2^{-1}(B_2[j])=d^{-1}(D[j])$. Restricting morphisms appropriately, we obtain the hypotheses of Lemma 3.11 and find that Fig. 2.3 is a pushout for j as well as for $j-1$. Similarly, the new left square is a pushout by Lemma 3.11. We have two pushouts that constitute the desired $\mathbf{G}[j] \Rightarrow \mathbf{H}[j]$ via $\rho p[j]$.

We must show that $\mathbf{H}[L]$ is isomorphic to $\mathbf{G}[L]$. By (3.8.2) and (3.8.3), any $j \geqq j^{\#}$ has $m_{K,1}[j]=m_{K,2}[j]$ and $m_{D,1}[j]=m_{D,2}[j]$, so it will suffice to show that $c_1[L]$ and $c_2[L]$ are isomorphisms. Because pushing out from an isomorphism yields an isomorphism, it will suffice to show that $b_1[L]$ and $b_2[L]$ are isomorphisms. We already have injectivity, so only surjectivity needs to be checked. Nodes and arcs will be treated separately. Because b_1 is surjective on nodes, induction on j shows that each $b_1[j]$ is surjective on nodes. For $j \geqq j^{\#}$, (5.3.4) then implies surjectivity of $b_1[j]$ on arcs. For $j=0$ there are $L-1$ nodes in $(B_2-b_2K)[j]$ and each step with $j \neq j^{\#}$ removes one of these nodes, so $b_2[L]$ is surjective on nodes. By (5.3.4) and $L \geqq j^{\#}$, it is also surjective on arcs. □

We conjecture that Theorem 5.5 is still true without the assumption that b_1 is surjective on nodes. Without this assumption, there are cases where $\mathbf{G} \gg_1 \mathbf{H}$ and $\mathbf{G} \gg_{\text{gar}} \mathbf{H}^{\#}$ but Theorem 5.5(3) is not true. The weaker statement

$$(\exists \mathbf{X})[\mathbf{H} \gg_{\text{gar}}^{*} \mathbf{X} \text{ and } (\mathbf{H}^{\#} \gg_1 \mathbf{X} \text{ or } \mathbf{H}^{\#} \gg_{\text{gar}}^{*} \mathbf{X})]$$

is true, but we have not been able to show that this suffices for proving the theorem.

COROLLARY 5.6. *Let $\mathcal{F}$ be a family of acyclic record structures that allows indirection and is closed under garbage collection. Let $\gg_1$ be a relation on colored graphs such that, for some set $\mathcal{P}$ of productions satisfying the hypotheses of Theorem* 4.5 *and Theorem* 5.5,

(1) $\mathbf{G} \gg_1 \mathbf{H}$ *iff* $(\exists p \text{ in } \mathcal{P})(\exists \rho \text{ in } R)(\mathbf{G} \Rightarrow \mathbf{H} \text{ via } \rho p)$;

(2) $(\mathcal{F}, \gg_1)$ *is strongly Church–Rosser.*

Then $(\mathcal{F}, \gg)$ is Church–Rosser, where $\gg$ is the union of $\gg_1$, $\gg_{\text{ind}}$, and $\gg_{\text{gar}}$.

Proof. By Theorem 4.5, $(\mathcal{F}, (\gg_1) \cup (\gg_{\text{ind}}))$ is Church–Rosser. The union of $\mathcal{P}$ and all the productions $p_{\text{ind}}(c')$ satisfies the hypotheses on $\mathcal{P}$ in Theorem 5.5, and the conclusion follows. □

6. Language design suggestions. Results like Theorem 4.5 and Theorem 5.5 can be interpreted as suggestions for the low level language programmer or for the high level language designer. Their significance is clearest in a multiprocessing context. One or more main processes under user control manipulate a record structure while service processes operate asynchronously in parallel with the main processes. The service processes follow paths of *ind* pointers or collect garbage. For the sake of definiteness, consider garbage collection and suppose that a list of backpointers is maintained for each record. Writing at low level, the user can inspect and manipulate backpointer information. Without some synchronization between the user and the garbage collector, a very unpleasant interaction can occur. The user looks at a backpointer and decides to follow it to a record RD. Then the garbage collector deletes RD. Then the user follows the invalid pointer. To prevent this interaction the user should lock out the garbage collector during some user computations, but locked out periods should be kept brief to obtain the benefits of parallelism. Theorem 5.5 suggests that the garbage collector be locked out during computations that correspond to applications of productions, but that garbage collection between applications of certain productions can do no harm. The user could apply a production, give the garbage collector free rein while he does a long private computation not involving the record structure, and then apply another production. If other user's activities during the private computation do not cause trouble [the Church–Rosser property for $(\mathscr{F}, \gg_1)$], then no combination of other user's activities and garbage collection during the private computation can cause trouble [the Church–Rosser property for $(\mathscr{F}, \gg)$]. This is a slight overstatement, in that "trouble" is an intuitive concept and does not perfectly coincide with any precise mathematical concept like lack of the Church–Rosser property. As in our backpointer example, many particular troubles do correspond to failures of the Church–Rosser property.

Of course it is difficult to correlate low level code with the definition of treelike proper fast productions. Low level programming is always difficult, especially with asynchronous parallelism. Rather than ask programmers to write at low level and keep Theorem 5.5 in mind, we would ask language designers to ensure that the record handling facilities of high level languages have the same net effect when high level programs are compiled. Garbage collection can take place "during" execution of a single high level operation, provided that the compiled code corresponds to a sequence of applications of treelike proper fast productions interspersed with private computations. The garbage collector is only locked out during each application of a production. Similar but more complex recommendations can be made for user communities with several record structures, for synchronization mechanisms that can lock the garbage collector out of a region rather than the whole structure, and so on. In practice it may be necessary to give programmers more flexibility, so that some high level programs will be such that the compiler does not guarantee good interaction with the garbage collector. In that case the language facilities that put the burden of assuring good interaction on the user should be very clearly visible whenever they occur in a program. The facilities for which good interaction is guaranteed should be rich enough that only expert programmers with stringent performance goals will ever feel a need to use the dangerous facilities. This is like the consensus that is beginning to emerge regarding control structures: **goto** should be provided, but a varied collection of more disciplined control structures should be provided to serve the needs of most programmers most of the time. The main limitation on the significance of Theorem 5.5 for language design is the simplicity of the garbage collection considered here. Unreachable cycles are not recognized as garbage, so we needed to assume that B_2 is acyclic in a treelike production. The basic theory has no such limitation, but its role in the more complex situation remains to be worked out.

Appendix A: Proof of Theorem 4.5(6). We begin by recalling what is hypothesized by Theorem 4.5. Let $\mathscr{F}$ be a family of acyclic record structures that allows indirection. Let $\gg_1$ be a relation on colored graphs such that, for some set $\mathscr{P}$ of rooted biproper fast productions wherein no node in B_1 is colored I and no arc in B_1 is colored *ind* or in C_{var},

$$\mathbf{G} \gg_1 \mathbf{H} \quad \text{iff} \quad (\exists p \text{ in } \mathscr{P})(\exists r \text{ in } R)(\mathbf{G} \Rightarrow \mathbf{H} \text{ via } rp); \tag{A.1}$$

$$(\mathscr{F}, \gg_1) \text{ is strongly Church–Rosser.} \tag{A.2}$$

Under these hypotheses the theorem claims that $(\mathscr{F}, \gg_4)$ is strongly Church–Rosser, where

$$\mathbf{G} \gg_2 \mathbf{H} \quad \text{iff} \quad \mathbf{G} \gg_1^{=} \mathbf{H}; \tag{A.3}$$

$$\mathbf{G} \gg_3 \mathbf{H} \quad \text{iff} \quad [\mathbf{G} \text{ has normal form } \mathbf{H} \text{ in } (\mathscr{F}, \gg_{\text{ind}})]; \tag{A.4}$$

$$\mathbf{G} \gg_4 \mathbf{H} \quad \text{iff} \quad (\exists \mathbf{X})(\mathbf{G} \gg_2 \mathbf{X} \gg_3 \mathbf{H}). \tag{A.5}$$

The proof in § 4 proceeds under the assumption that, for arbitrary $\mathbf{G}$, $\mathbf{H}$, $\mathbf{H}^{\#}$ in $\mathscr{F}$,

$$(\mathbf{G} \gg_2 \mathbf{H} \;\&\; \mathbf{G} \gg_3 \mathbf{H}^{\#}) \quad \text{implies} \quad (\exists \mathbf{W}, \mathbf{X})(\mathbf{H} \gg_3 \mathbf{X} \;\&\; \mathbf{H}^{\#} \gg_2 \mathbf{W} \gg_3 \mathbf{X}). \tag{A.6}$$

All that remains is to prove (A.6). Suppose $\mathbf{G} \gg_2 \mathbf{H}$ and $\mathbf{G} \gg_3 \mathbf{H}^{\#}$. If $\mathbf{G} = \mathbf{H}$ then we may let $\mathbf{W}$ and $\mathbf{X}$ be $\mathbf{H}^{\#}$, so we may assume $\mathbf{G} \neq \mathbf{H}$. Therefore there is a direct derivation

$$\mathbf{G} \Rightarrow \mathbf{H} \quad \text{via } rp \text{ based on } g \tag{A.7}$$

for some p in $\mathscr{P}$, r in R, and $g\colon B_1 \to G$. We want to construct a similar derivation

$$\mathbf{H}^{\#} \Rightarrow \mathbf{W} \quad \text{via } r^{\#}p \text{ based on } g^{\#} \tag{A.8}$$

such that $\mathbf{W}$ and $\mathbf{H}$ have the same normal form $\mathbf{X}$ in $(\mathscr{F}, \gg_{\text{ind}})$. A closer look at such normal forms will be helpful. From $\mathbf{G} \gg_3 \mathbf{H}^{\#}$ it follows that $\mathbf{H}^{\#}$ has the same (under the obvious bijection between pairs of sets) nodes and arcs as does $\mathbf{G}$. The colorings and the source maps are the same. Targets of arcs, however, are different:

$$\mathbf{t}^{\#}_{H}x = (\textbf{if } m_G\mathbf{t}_G x \neq I \textbf{ then } \mathbf{t}_G x \textbf{ else } \mathbf{t}^{\#}_{H}y), \tag{A.9}$$

where y is the outarc from $\mathbf{t}_G x$ in case $m_G\mathbf{t}_G x = I$. With colorings suppressed for readability, Fig. 4.5 summarizes how (A.8) will be obtained. The pushouts Fig. 4.5(1) and Fig. 4.5(2) in GRAPHS are from (A.7). The other squares are also pushouts, but in a category TARGRAPHS that is like GRAPHS but has only the target operation mapping arcs to nodes. We may consider G, D, H to be objects in this category by simply forgetting about sources of arcs.

To obtain Fig. 4.5(3) we begin by defining ϕ as a map from items of G to items of $H^{\#}$. For each node ζ in G let

$$\phi\zeta = (\textbf{if } m_G\zeta \neq I \textbf{ then } \zeta \textbf{ else } \mathbf{t}^{\#}_{H}y), \tag{A.10}$$

where y is the outarc of ζ in case $m_G\zeta = I$. Let ϕ be the identity map on arcs. Defining $g^{\#} = \phi g$ yields a map from items of B_1 to items of $H^{\#}$. We claim that

$$\phi \text{ preserves targets and } g^{\#} \text{ is a graph morphism.} \tag{A.11}$$

Given an arc x in G, let $\zeta = \mathbf{t}_G x$ and let y be the outarc from ζ in case $m_G\zeta = I$. Then (A.10) and (A.9) imply $\phi\mathbf{t}_G x = \mathbf{t}^{\#}_{H}x = \mathbf{t}^{\#}_{\mathrm{H}}\phi x$, so ϕ preserves targets. By $g^{\#} = \phi g$, this implies that $g^{\#}$ preserves targets. We must show that $g^{\#}$ preserves sources. Let z be an arc in B_1. Then $m_G g z = r m_1 z \neq ind$ and so $m_G \mathbf{s}_G g z \neq I$. Applying (A.10) to $\zeta = \mathbf{s}_G g z = g \mathbf{s}_1 z$, we find that

$$g^{\#}\mathbf{s}_1 z = \phi g \mathbf{s}_1 z = \phi\zeta = \zeta = \mathbf{s}_G g z = \mathbf{s}_G \phi g z = \mathbf{s}_G g^{\#} z,$$

so $g^{\#}$ preserves sources and the proof of (A.11) is complete.

The analog of Lemma 2.4 for TARGRAPHS is obtained by forgetting about sources and colors. To obtain Fig. 4.5(3) as a pushout in TARGRAPHS we must check that

$$\mathbf{t}_H^{\#} A_0 \subseteq N_0 \cup \phi c_1 D \quad \text{for} \quad (N_0, A_0) = H^{\#} - \phi G;$$

$$(\forall y, y' \text{ in } G)\,[\phi y = \phi y' \text{ implies } (y = y' \text{ or } y, y' \text{ in } c_1 D)].$$

Because ϕ is the identity on arcs, $A_0 = \varnothing$ and the inclusion is trivial. The second condition will be derived from

(A.12) $$\zeta \neq \phi\zeta \quad \text{implies} \quad m_G\zeta = I = m_H^{\#}\zeta \quad \text{implies} \quad \zeta \in c_1 D - \phi G.$$

To prove (A.12), suppose $\zeta \neq \phi\zeta$. Then ζ is a node and (A.10) implies $m_G\zeta = I$, while $m_H^{\#} = m_G$. The mirror image of (3.6.2) implies that no node in $B_1 - b_1 K$ colored in C_{var}, while no node in B_1 at all is colored I. Therefore no node y in $B_1 - b_1 K$ can have $r m_1 y = I$. Since $G = gB_1 \cup c_1 D$ as a set of items and $g b_1 = c_1 d$, $m_G \zeta = I$ implies that ζ is in $c_1 D$. By (A.9) and (A.10), it also implies that ζ is not in ϕG. Therefore (A.12) holds.

Our first use of (A.12) is in dealing with the situation where $\phi y = \phi y'$ but $y \neq y'$. We may then assume $y \neq \phi y$, which implies that y is a node in $c_1 D$. If $y' \neq \phi y'$ also there is nothing more to show, so we may assume that $y' = \phi y' = \phi y$. There is a nonnull path from y to y' in G using arcs colored *ind*. But no arc in B_1 is colored *ind* or in C_{var}, so these arcs are not in gB_1. By Lemma 2.4(3) for the pushout Fig. 4.5(1), the last arc in the path has target y' in $(G - gB_1) \cup g b_1 K$ and hence in $c_1 D$. Lemma 2.4 in TARGRAPHS yields Fig. 4.5(3).

Now Fig. 4.5(1) and Fig. 4.5(3) are both pushouts in TARGRAPHS, so Lemma 3.9 implies that the large square Fig. 4.5(1, 3) is a pushout from b_1 and $d^{\#} = \delta d$ to form $H^{\#}$. To obtain the first pushout needed for (A.8) we must bridge the gap between TARGRAPHS and GRAPHS[C]. It will be helpful to know that $c_1 D \subseteq c_1^{\#} D^{\#}$ (an inclusion that makes sense because G and $H^{\#}$ have the same set of items). Any y in D with $c_1 y = \phi c_1 y$ has $c_1 y = c_1^{\#} \delta y$. By (A.12), any y in D with $c_1 y \neq \phi c_1 y$ has $c_1 y$ in $H^{\#}$ but not in ϕG, so it must be in $c_1^{\#} D^{\#}$. Thus $c_1 D \subseteq c_1^{\#} D^{\#}$.

To define $\mathbf{s}_D^{\#}$: $(D^{\#})_{\text{arcs}} \to (D^{\#})_{\text{nodes}}$, consider any $y^{\#}$ in $(D^{\#})_{\text{arcs}}$. There is a unique arc y in D with $y^{\#} = \delta y$ because ϕ is the identity map on arcs. Now $\mathbf{s}_D^{\#} y^{\#}$ may be defined to be

$$\mathbf{s}_D^{\#} y^{\#} = (c_1^{\#})^{-1} c_1 \mathbf{s}_D y = (c_1^{\#})^{-1} \mathbf{s}_G c_1 y = (c_1^{\#})^{-1} \mathbf{s}_G c_1^{\#} y^{\#},$$

where $(c_1^{\#})^{-1} c_1 \mathbf{s}_D y$ is well-defined because $c_1 D \subseteq c_1^{\#} D^{\#}$. Because $\mathbf{s}_G = \mathbf{s}_{\mathrm{H}}^{\#}$, $c_1^{\#}$ becomes a graph morphism when $\mathbf{s}_D^{\#}$ is taken to the source operation on $D^{\#}$. By (A.11), $c_1^{\#} d^{\#} = g^{\#} b_1$ is a graph morphism. But $c_1^{\#}$ is an injective graph morphism, so $d^{\#}$ must be a graph morphism also. Fig. 4.5(1, 3) is now a pushout in GRAPHS.

For colors we need an appropriate recoloring $r^{\#}$. Let $r^{\#} c = rc$ except when c is in C_{var} and $rc = I$. In that case $c = m_1 z$ for a unique node z in B_1 by (3.6.1). Let $r^{\#} c = m_H^{\#} g^{\#} z$ then. Now $g^{\#}$: $(B_1, r^{\#} m_1) \to (H^{\#}, m_H^{\#})$ in GRAPHS[C]. When K is colored by $m_{K,1}^{\#} = r^{\#} m_1 b_1$ and $D^{\#}$ is colored by $m_{D,1}^{\#} = m_H^{\#} c_1^{\#}$ the maps b_1 and $c_1^{\#}$ preserve colors, and then $d^{\#}$ preserves colors because $c_1^{\#} d^{\#} = g^{\#} b_1$. Figure 4.5(1, 3) is now a pushout in GRAPHS[C].

We claim that any x, x' in K with $d^{\#}x = d^{\#}x'$ have $r^{\#}m_2b_2x = r^{\#}m_2b_2x'$. Suppose $d^{\#}x = d^{\#}x'$ and let $c = m_2b_2x$ and $c' = m_2b_2x'$. If x is an arc then $dx = dx'$ because ϕ is the identity on arcs, and then $rc = rc'$ because colors are assigned consistently in (A.7). But (3.6.3) and the lack of variable colored arcs in B_1 imply that c, c' are in C_{fix}, so $r^{\#}c = c = rc = rc' = c' = r^{\#}c'$. On the other hand, suppose x is a node. We may assume x is not the root node in (3.8.2). Therefore

$$r^{\#}c = r^{\#}m_1b_1x = m_H^{\#}c_1^{\#}d^{\#}x = m_H^{\#}c_1^{\#}d^{\#}x',$$

so $r^{\#}c = r^{\#}c'$ will follow as soon as we show that x' is not the root node. We suppose x' is the root and derive a contradiction. By (3.8.1), b_1x is reachable from $b_1x' \neq b_1x$ in B_1, so there is a nonnull path from $g^{\#}b_1x'$ to $g^{\#}b_1x$ in $H^{\#}$. But $d^{\#}x = d^{\#}x'$ implies that $g^{\#}b_1x = g^{\#}b_1x'$, contradicting the fact that $H^{\#}$ is acyclic. Our claim does hold, and so there is a coloring $m_{D,2}^{\#}$ of $D^{\#}$ that agrees with $m_{D,1}^{\#}$ on $D^{\#} - d^{\#}K$ and makes $d^{\#}$ preserve the coloring $m_{K,2}^{\#} = r^{\#}m_2b_2$ of K. Pushing out from b_2: $(K, m_{K,2}^{\#}) \to (B_2, r^{\#}m_2)$ and $d^{\#}$: $(K, m_{K,2}^{\#}) \to (D^{\#}, m_{D,2}^{\#})$ in GRAPHS[C] yield $\mathbf{W}$ in GRAPHS[C] to complete (A.8). By Lemma 3.10 in TARGRAPHS, the pushout to $\mathbf{W}$ can be factored as Fig. 4.5(2, 4) in TARGRAPHS.

We must show that $\mathbf{W}$ and $\mathbf{H}$ have the same normal form $\mathbf{X}$ in $(\mathcal{F}, \gg_{\text{ind}})$. Specifically, we will construct an isomorphism β: $\mathbf{H} \to \mathbf{W}$ in the category SOUGRAPHS[C] that is like GRAPHS[C] but has only the source operation mapping arcs to nodes. When β is used to identify items of $\mathbf{H}$ with items of $\mathbf{W}$, only targets of arcs can be different in the two colored graphs. We will also show that the differences in targets are not too severe. Specifically, each x in $(H)_{\text{arcs}} = (W)_{\text{arcs}}$ will be shown to have

(A.13) $(m_H\mathbf{t}_Hx = I \;\&\; \mathbf{t}_Wx \neq \mathbf{t}_Hx)$ implies $(\mathbf{t}_Wx = \mathbf{t}_Wz$ for z the outarc of $\mathbf{t}_Hx)$.

Another helpful property of β will be that $\beta z = \psi z$ unless z is a node in c_2D. Thus ψ will resemble the identity map, much as ϕ resembles the identity map in (A.12), when β is used to identify items of H with items of W. Assuming β and (A.13) for the time being, we can now prove that the normal form $\mathbf{X}$ of $\mathbf{H}$ is the same as the normal form $\mathbf{X}^{\#}$ of $\mathbf{W}$ in $(\mathcal{F}, \gg_{\text{ind}})$. Putting objects into SOUGRAPHS[C] by forgetting about targets of arcs, we already have $\mathbf{X} \equiv \mathbf{H} \equiv \mathbf{W} \equiv \mathbf{X}^{\#}$. Thus the only way $\mathbf{X}^{\#}$ and $\mathbf{X}$ might possibly differ is in targets of arcs. Consider targets in $\mathbf{X}$. The relation analogous to (A.9) but between $\mathbf{t}_X$ and $\mathbf{t}_H$ leads to a recursive procedure for computing $\mathbf{t}_X$ by inspecting nodes of $\mathbf{H}$ and following outarcs when the nodes are colored I. For each arc x in $\mathbf{H}$, the sequence $\mathbf{T}_Hx$ of nodes encountered is easily described. If $m_H\mathbf{t}_Hx \neq I$ then $\mathbf{T}_Hx$ is the sequence $(\mathbf{t}_Hx)$ of length 1. Otherwise $m_H\mathbf{t}_Hx = I$ and $\mathbf{t}_Hx$ has a unique outarc z in $\mathbf{H}$. Then $\mathbf{T}_Hx$ is the concatenation $(\mathbf{t}_Hx) \cdot \mathbf{T}_Hz$. Note that $\mathbf{t}_Xx$ is the last node in $\mathbf{T}_Hx$. There is a similar sequence $\mathbf{T}_Wx$ for finding targets in $\mathbf{X}^{\#}$, and $\mathbf{X} = \mathbf{X}^{\#}$ will follow as soon as $\mathbf{T}_Hx$ and $\mathbf{T}_Wx$ have been shown to end with the same node. We use induction on the length of $\mathbf{T}_Hx$. Suppose first that the length is 1, so that $m_H\mathbf{t}_Hx \neq I$. We claim that $\psi\mathbf{t}_Hx = \mathbf{t}_Hx$. If $\mathbf{t}_Hx$ is not in c_2D this is trivial. Otherwise $m_Gc_1c_1^{-1}\mathbf{t}_Hx \neq I$ (by (3.6.3) and the lack of I nodes in B_1), so $\psi\mathbf{t}_Hx = \mathbf{t}_Hx$ because (A.12) implies $\phi c_1c_2^{-1}\mathbf{t}_Hx = c_1c_2^{-1}\mathbf{t}_Hx$. But $\psi\mathbf{t}_Hx = \mathbf{t}_Hx$ implies that $\mathbf{t}_Wx = \mathbf{t}_Hx$ and so $\mathbf{T}_Wx = (\mathbf{t}_Hx) = \mathbf{T}_Hx$. To pass from length θ to length $\theta + 1$, consider $\mathbf{T}_Hx = (\mathbf{t}_Hx) \cdot \mathbf{T}_Hz$ with $m_H\mathbf{t}_Hx = I$ and z the outarc of $\mathbf{t}_Hx$. If $t_Wx = \mathbf{t}_Hz$ then z is also the outarc of $\mathbf{t}_Wx$ in W and $\mathbf{T}_Wx = (\mathbf{t}_Wx) \cdot \mathbf{T}_Wz$. If $\mathbf{t}_Wx \neq \mathbf{t}_Hx$ then (A.13) implies $\mathbf{T}_Wx = \mathbf{T}_Wz$. In both cases $\mathbf{T}_Wx$ has the same last node as $\mathbf{T}_Wz$, which has the same last node as $\mathbf{T}_Hz$ by the induction hypothesis. But $(\mathbf{t}_Hx) \cdot \mathbf{T}_Hz = \mathbf{T}_Hx$, so $\mathbf{T}_Wx$ has the same last node as $\mathbf{T}_Hx$. This completes the proof that $\mathbf{W}$ and $\mathbf{H}$ have the same normal form, assuming that β exists and (A.13) holds.

All that remains is to construct β and verify (A.13). Because $c_1D \subseteq c_1^\# D^\#$, there is an injective SOUGRAPHS morphism $\sigma: D \to W$ with $\sigma = c_2^\# (c_1^\#)^{-1} c_1$. There is also a SOUGRAPHS morphism $\tau: B_2 \to W$ that agrees with $\sigma\, db_2^{-1}$ on b_2K and with $h^\# = \psi h$ on $B_2 - b_2K$. (The only nontrivial point about τ is checking that an arc in $B_2 - b_2K$ with source in b_2K has the source operation preserved by τ. But this follows because ϕ is the identity on arcs.) By the universal property of Fig. 4.5(2), there is $\beta: H \to W$ in SOUGRAPHS with $\beta h = \tau$ and $\beta c_2 = \sigma$. Note that $\beta z = \psi z$ unless z is a node in c_2D. Because σ is injective while τ is injective on $B_2 - b_2K$ and never collapses items in $B_2 - b_2K$ with items in b_2K, β is injective. For showing that β is surjective it will help to know that $c_1^\# D^\# \subseteq c_1D$. For arcs the inclusion is trivial. Consider any node $y^\#$ in $D^\#$, and let ζ be $c_1^\# y^\#$. If $\zeta = \phi\zeta$ then ζ is in $\phi G \cap c_1^\# D^\#$, which implies that ζ is in c_1D because Fig. 4.5(3) is a pushout. If $\zeta \neq \phi\zeta$ then (A.12) implies that ζ is in c_1D. Therefore $c_1^\# D^\# \subseteq c_1D$. As a set of items $W = h^\#(B_2 - b_2K) \cup c_2^\# D^\#$, so β is surjective because

$$h^\#(B_2 - b_2K) \subseteq \tau B_2 \subseteq \beta H;$$

$$c_2^\# D^\# = c_2^\# (c_1^\#)^{-1} c_1^\# D^\# \subseteq c_2^\# (c_1^\#)^{-1}) c\acute{c}_1 D = \sigma D \subseteq \beta H.$$

Therefore β is an isomorphism in SOUGRAPHS. To pass from SOUGRAPHS to SOUGRAPHS[C] we must show that any z in H has $m_H z = m_W \beta z$.

Case A.1 [$z = hy$ for y in B_2 with $m_2 y$ in C_{fix}]. We claim $\psi z = \beta z$. This is trivial unless y is a node in b_2K. In that case let $b_2 x = y$. By (3.6.3), $m_1 b_1 x$ is in C_{fix} and $m_G g b_1 x = r m_1 b_1 x = m_1 b_1 x \neq I$. By (A.12), $\psi z = c_2^\# (c_1^\#)^{-1} \phi g b_1 x = c_2^\# (c_1^\#)^{-1} g b_1 x = \sigma\, dx = \beta z$. Now we calculate:

$$m_H z = r m_2 y = m_2 y = r^\# m_2 y = m_W h^\# y = m_W \psi z = m_W \beta z.$$

Case A.2 [otherwise]. By (3.6.2), (3.6.3), and $H = hB_2 \cup c_2 D$ as a set of items, $z = c_2 y$ for y in D with $m_{D,1} y = m_{D,2} y$. Moreover, let $y^\# = (c_1^\#)^{-1} c_1 y$. Then $m_{D,1}^\# y^\# = m_{D,2}^\# y^\#$ can be shown as follows. Suppose 'otherwise. Then $y^\# = d^\# x$ for x in K with $m_1 b_1 x \neq m_2 b_2 x$. By (3.6.3), $m_1 b_1 x$ and $m_2 b_2 x$ are in C_{fix}. But $\phi c_1\, dx = c_1^\# d^\# x = c_1 y \neq c_1\, dx$, so (A.12) implies $r m_1 b_1 x = m_G c_1\, dx = I$. But B_1 lacks I nodes, so $m_1 b_1 x$ is in C_{var}, a contradiction. Therefore $m_{D,1}^\# y^\# = m_{D,2}^\# y^\#$ in addition to $m_{D,1} y = m_{D,2} y$. We calculate:

$$m_H z = m_G c_1 y = m_H^\# c_1 y = m_{D,1}^\# y^\# = m_{D,2}^\# y^\# = m_W c_2^\# y^\# = m_W \sigma y = m_W \beta z.$$

This completes the proof that β is an isomorphism between **H** and **W** in SOUGRAPHS[C].

To show that (A.13) will hold when β is used to identify **H** with **W** in SOUGRAPHS[C], suppose $m_H \mathbf{t}_H x = I$and $\mathbf{t}_W \beta x \neq \beta \mathbf{t}_H x$. We need $\mathbf{t}_W \beta x = \mathbf{t}_W \beta z$ for z the outarc of $\mathbf{t}_H x$. Let $y = t_{\mathrm{H}}\mathrm{x}$. Because ψ preserves targets and β agrees with ψ on arcs and on nodes in $h(B_2 - b_2K)$, we have y in c_2D with $\beta y \neq \psi y$. For $\eta = c_1 c_2^{-1} y$ we have

$$c_2^\# (c_1^\#)^{-1} \eta = \beta y \neq \psi y = c_2^\# (c_1^\#)^{-1} \phi \eta,$$

so (A.12) implies $m_G \eta = I$. The outarc ζ of η in G must be in c_1D because B_1 has no arcs colored *ind* or in C_{var}, so $\mathrm{c}_2\mathrm{c}_1^{-1}\zeta$ is an outarc of y in H. Hence $c_2 c_1^{-1} \zeta = z$ and z is in c_2D with $c_1 c_2^{-1} z = \zeta$. We calculate:

$$\begin{aligned}
\mathbf{t}_W \beta x &= \mathbf{t}_W \psi x = \psi \mathbf{t}_H x = \psi y \\
&= c_2^\# (c_1^\#)^{-1} \phi \eta \\
&= c_2^\# (c_1^\#)^{-1} \mathbf{t}_H^\# \zeta \qquad \text{by (A.10)} \\
&= \mathbf{t}_W c_2^\# (c_1^\#)^{-1} c_1 c_2^{-1} z \\
&= \mathbf{t}_W \beta z.
\end{aligned}$$

This completes the proof of (A.13) and hence of (A.6). The debts incurred while proving Theorem 4.5 have all been paid.

Appendix B: Pushouts. Intuitively, a pushout displays how one large graph results from "gluing together" two smaller ones. In the diagram (B.2) below we "glue" B and D together to form G. The morphisms b and d tell how the parts B and D fit together to form G. For each x in K, the items bx in B and dx in D are fused to form one item $gbx = cdx$ in G. Apart from this gluing, G resembles the disjoint union of B and D. The morphisms g and c have images gB and cD such that anything in G is in one of the images. As a set of items, G satisfies

(B.1) $$G = gB \cup cD.$$

For $K \neq \varnothing$ some items are in both images because $gb = cd$, but G is as close to being a disjoint union as the gluing specified by b and d will allow. The only items in $gB \cap cD$ are those in gbK, and the only departures from injectivity present in g and c are those incidental to gluing. Thus g should not have $gy = gy'$ for $y \neq y'$ unless y and y' are both involved in gluing.

Instead of defining pushouts by an elaborate construction that formalizes the intuition directly, we will say that pushouts *do* for us in the categories of interest here. Specifications of objects in terms of what they do at a categorical level are known as *universal properties*. As is to be expected when category theory is applied rather than pursued for its own sake, this paper is sometimes concerned with interactions between universal properties and the concrete constructions that yield objects with these properties. As in [1, p. 44], [14, p. 139], a *pushout* is any commutative diagram of the form

(B.2) $$\begin{tikzcd} K \arrow[r, "b"'] \arrow[d, "d"'] & B \arrow[d, "g"] \\ D \arrow[r, "c"'] & G \end{tikzcd}$$

such that, whenever an object X and morphisms $\beta: B \to X$, $\delta: D \to X$ satisfy

(B.3) $$\beta b = \delta d,$$

then there is a unique morphism $\gamma: G \to X$ such that

(B.4) $$\gamma g = \beta \quad \text{and} \quad \gamma c = \delta.$$

This definition makes sense in any category, but it leaves open the question of whether there are any such diagrams in the category under discussion. We speak of *pushing out from* the morphisms b and d in (B.12). If there is any pushout from b and d at all, then it is unique up to isomorphism in the usual way [1, p. 45], [14, p. 135]. If there is a pushout from any morphisms b and d that start from the same object, the category under discussion is said to *have pushouts*. This paper considers the category GRAPHS[C] and some other categories that are helpful in proofs, such as the category SETS whose objects are sets and whose morphisms are maps between sets. All the categories we consider have pushouts.

The explicit construction showing that SETS has pushouts is well-known and is reviewed in [21, § 2], which also notes that this construction leads to pushouts in

GRAPHS and GRAPHS[C]. Given $b: K \to B$ and $d: K \to D$ in GRAPHS or GRAPHS[C], we push out in SETS from b and d as maps from nodes to nodes and also from b and d as maps from arcs to arcs. These two pushouts in SETS yield G as a pair (node set, arc set), and the natural choice of source and target maps (defined by the arc set pushout) yields a graph structure. If b and d are originally colored graph morphisms then the natural choice of coloring (defined by the two pushouts in SETS) yields a colored graph structure. It is easy to check that the two diagrams (B.2) in SETS correspond to one diagram (B.2) in GRAPHS or GRAPHS[C] with this structure on G, and that this diagram is a pushout. Writing out the details can be recommended as an exercise in relating explicit constructions to universal properties. The reader should try to construct pushouts in SETS from the intuitive hints about gluing that began this appendix. If necessary, [21, § 2] can be consulted.

Acknowledgment. The authors appreciate the comments on drafts of this paper by J. Staples. Many improvements were suggested by one of the referees in a very detailed and helpful report. A condensation of an earlier version of this paper was presented as [8] at the 4th International Colloquium on Automata, Languages, and Programming, Turku, Finland, July, 1977.

REFERENCES

[1] M. A. ARRIB AND E. G. MANES, *Arrows, Structures, and Functors*, Academic Press, New York, 1975.

[2] L. P. DEUTSCH AND D. G. BOBROW, *An efficient incremental automatic garbage collector*, Comm. ACM, 19 (1976), pp. 522–526.

[3] T. W. DOEPPNER, *Parallel program correctness through refinement*, Proc. 4th ACM Symp. on Principles of Programming Languages, Santa Monica, January 1977, pp. 155–169.

[4] H. EHRIG, *Introduction to the algebraic theory of graph grammars*, Bericht Nr. 78-28, Fachbereich Informatik, Tech. U. Berlin, August 1978; *Proc. Internat. Workshop on Graph Grammars and their Applications to Computer Science and Biology*, Bad Honeff, October 1978, to appear.

[5] H. EHRIG, H. J. KREOWSKI, A. MAGGIOLO-SCHETTINI, B. K. ROSEN AND J. WINKOWSKI, *Deriving structures from structures*, Lecture Notes in Computer Science, 64 (1978), pp. 177–190.

[6] H. EHRIG, M. PFENDER AND H. J. SCHNEIDER, *Graph grammars: An algebraic approach*, Proc. 14th Ann. IEEE Symp. on Switching and Automata Theory, Iowa City, October 1973, pp. 167–180.

[7] H. EHRIG AND B. K. ROSEN, *Commutativity of independent transformations on complex objects*, IBM Research Rep. RC 6251, October 1976.

[8] ———, *The mathematics of record handling*, Lecture Notes in Computer Science, 52 (1977), pp. 206–220.

[9] ———, *Concurrency of manipulations in multidimensional information structures*, Lecture Notes in Computer Science, 64 (1978), pp. 165–176.

[10] ———, *Concurrency of manipulations in multidimensional information structures*, Bericht Nr. 78-13, Fachbereich Informatik, Tech. U. Berlin, May 1978; Revision, *Theor. Comput. Sci.*, to appear.

[11] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 507–518.

[12] D. GRIES, *On believing programs to be correct*, Comm. ACM, 20 (1977), pp. 49–50.

[13] ———, *An exercise in proving parallel programs correct*, Comm. ACM, 20 (1977), pp. 921–930.

[14] H. HERRLICH AND G. STRECKER, *Category Theory*, Allyn and Bacon, Rockleigh, NJ, 1973.

[15] G. HUET, *Confluent reductions : Abstract properties and applications to term rewriting systems*, Proc. 18th Ann. IEEE Symp. on Foundations of Computer Sci., Providence, October 1977, pp. 30–45.

[16] D. B. JOHNSON, *Finding all the elementary circuits of a directed graph*, this Journal, 4 (1975), pp. 77–84.

[17] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1 (2nd ed.), Addison-Wesley, Reading, MA, 1973.

[18] M. J. O'DONNELL, *Computing in systems described by equations*, Lecture Notes in Computer Science, 58 (1977), pp. 1–111.

[19] D. C. OPPEN AND S. A. COOK, *Proving assertions about programs that manipulate data structures*, Proc. 7th Ann. ACM Symp. on Theory of Computing, Albuquerque, May 1975, pp. 107–116.

[20] B. K. ROSEN, *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1973), pp. 160–187.

[21] ———, *Deriving graphs from graphs by applying a production*, Acta Informat., 4 (1975), pp. 337–357.

[22] ———, *Correctness of parallel programs: The Church–Rosser approach*, Theor. Comput. Sci., 2 (1976), pp. 183–207.

[23] R. SETHI, *Testing for the Church–Rosser property*, J. Assoc. Comput. Mach., 21 (1974), pp. 671–679.

[24] P. M. SPIRA AND A. PAN, *On finding and updating spanning trees and shortest paths*, this Journal, 4 (1975), pp. 375–380.

[25] J. STAPLES, *A class of replacement systems with simple optimality theory*, Bull. Austral. Math. Soc., 17 (1977), pp. 335–350.

[26] G. L. STEELE, *Multiprocessing compactifying garbage collection*, Comm. ACM, 18 (1975), pp. 495–508.

[27] H. R. STRONG, A. MAGGIOLO-SCHETTINI AND B. K. ROSEN, *Recursion structure simplification*, this Journal, 4 (1975), pp. 307–320.

[28] R. E. TARJAN, *Finding dominators in directed graphs*, this Journal, 3 (1974), pp. 62–89.

[29] P. L. WADLER, *Analysis of an algorithm for real time garbage collection*, Comm. ACM, 19 (1976), pp. 491–500.

[30] N. WIRTH AND C. A. R. HOARE, *A contribution to the development of ALGOL*, Comm. ACM, 9 (1966), pp. 413–431.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

CONDITIONS FOR OPTIMALITY OF THE HUFFMAN ALGORITHM*

D. STOTT PARKER, JR.†

Abstract. A new general formulation of Huffman tree construction is presented which has broad application. Recall that the Huffman algorithm forms a tree, in which every node has some associated weight, by specifying at every step of the construction which nodes are to be combined to form a new node with a new combined weight. We characterize a wide class of weight combination functions, the quasilinear functions, for which the Huffman algorithm produces optimal trees under correspondingly wide classes of cost criteria. In addition, known results about Huffman tree construction and related concepts from information theory and from the theory of convex functions are tied together. Suggestions for possible future applications are given.

Key words. Huffman algorithm, optimal tree construction, weighted path length, tree height, quasilinear functions, convex functions, Rényi entropy

1. Introduction. Although Huffman's algorithm was primarily developed for a problem in discrete coding as early as 1952 [16], it has recently been undergoing a considerable amount of research as more applications for it are uncovered. Most recently, Itai [17], van Leeuwen [19], Glassey and Karp [12], and Golumbic [13] have presented new perspectives on how the algorithm works and how it can be employed in new ways. Until now, all research has concentrated on two variations of the algorithm, which respectively minimize (i) the weighted path length, and (ii) measures akin to tree height, of the constructed tree. Modern applications for weighted path length minimization include (1) construction of optimal search trees [31], [15], [17], (2) merging of lists [8], [20], (3) minimization of absolute error bounds in sums of positive numbers [6], [29] and relative error bounds in products [26], (4) text file compression [25], and (5) optimal checking for leaky pipelines and water pollution [12]. Applications for tree height minimization include the determination of the minimum execution time for fanning-in data (in limited task-scheduling systems, and in arithmetic/Boolean sum- or product-accumulation, etc.) and problems related to speed in parallel processing [13]. This is by no means a complete listing.

Our interest in the algorithm comes mainly from its import for generating optimal evaluation trees in the compilation of expressions. Not only does the algorithm build optimal trees with respect to execution time, space usage, and roundoff error for many classes of limited expressions, it does so very efficiently. If N is the number of leaves in the tree to be constructed, Huffman's algorithm can be implemented in time $O(N \log N)$ when a fast priority queue is used. Moreover, van Leeuwen has shown that this time bound can be reduced to $O(N)$ if the leaf weights are given in sorted order [19]. (This suggests that the complexity of the algorithm is Θ $(N \log N)$, since sorting is at least that difficult.)

The two variations of the Huffman algorithm mentioned above are based on the same construction process, but use different tree cost functions and node merging methods. In the construction process each node has some associated weight, and these nodes are combined along with their weights to form a new weighted internal (parent) node in the tree. Construction terminates when all nodes have been combined into a final root node. The weighted path-length variation produces parent nodes having

* Received by the editors July 6, 1977, and in final revised form May 14, 1979.

† Department of Computer Science, University of Illinois at Urbana–Champaign. Now at Department of Computer Science, University of California, Los Angeles, California 90024. This research was supported in part by the National Science Foundation under Grants MCS 76-81686 and MCS 75-21758, and by an IBM Fellowship.

weights equal to the *sum* of the weights of its sons, while the tree-height variation uses the *maximum* of the son weights plus some nonnegative constant. This will all be discussed in greater detail below.

It is not immediately apparent why these two apparently unrelated incarnations of the Huffman algorithm both produce optimal trees. From the point of view of compiling it would be nice if we could use instances of the same construction to obtain trees optimal with respect to yet other cost measures besides tree height and path length. For example, suppose we wish to construct parse trees for parallel evaluation of sums of positive numbers, optimal with respect to some measure of both roundoff and time used. Since error bounds in this case correspond to path-length, and execution time to tree height, an optimal parse tree cannot be constructed using the Huffman algorithm unless a node-merging method more complicated than the two above is used. This problem raises the following question addressed and investigated in this paper: *for which problems will the Huffman algorithm produce optimal trees under exactly which cost*? We will identify a wide class of weight combination functions (encompassing the two standard methods above) which all produce optimal tree with the Huffman algorithm under corresponding classes of tree cost functions. We also tie together some results from information theory and the theory of convex functions in studying the optimality in these instances.

2. Basic machinery for Huffman tree construction. This section defines the notation to be used for the rest of the paper. The exposition here is not really introductory and readers seeking more background are referred to [18, § 2.3.4.5] or [1]. For the time being we confine ourselves to *binary* trees until the essential results are established. The extension to r-ary trees follows in an analogous way.

In the "binary tree construction problem" one is given a set of $n+1$ leaves having corresponding weights $\{w_1, w_2, \cdots, w_{n+1}\}$. The weights need not be normalized so that their sum comes out to be unity; we require only that they be nonnegative and given, for convenience, sorted by index:

$$0 \leqq w_1 \leqq w_2 \leqq \cdots \leqq w_{n+1}.$$

Although in some problems a particular ordering is to be enforced on the leaves (e.g., [15]) we presume in this paper that there are no constraints on the final order of the leaves in the constructed tree. Construction of a (full) binary tree on these leaves is then effected by n merges of pairs of "available" nodes. Each node in the pair is marked unavailable after a merge and their father (the result of the merge) is marked available, having as his weight some function F of the weights of its sons. Leaves are initially all marked available, of course. Note that n merges are necessary and sufficient, since binary trees on $n+1$ leaves have n internal nodes.

Each internal node defines the root of a binary subtree of the constructed tree, which implies that tree construction can be defined inductively in terms of *forests* (collection of trees) in the obvious way. The construction begins with a forest of $n+1$ one-node trees and repeatedly reduces the number of trees by 1 via root merge operations until only one tree is left.

Notation.

w_j—jth smallest leaf weight (i.e., w_1 is smallest, w_{n+1} largest);

l_j—path length (distance from the root) of w_j;

W_i—ith smallest internal node weight.

The name of the tree or forest in question will be added in parentheses to each quantity whenever it is not clear from context which tree or forest is meant.

DEFINITION. A *weight space* U for a weighted tree construction problem is a connected interval of the nonnegative reals R_+. All weights in the tree are elements of U.

DEFINITION. A *weight combination function* F: $U^2 \to U$ is any symmetric function which is closed as a binary operator on U. F is used to produce the weight of internal nodes generated by a merge operation in tree construction (cf. Fig. 1).

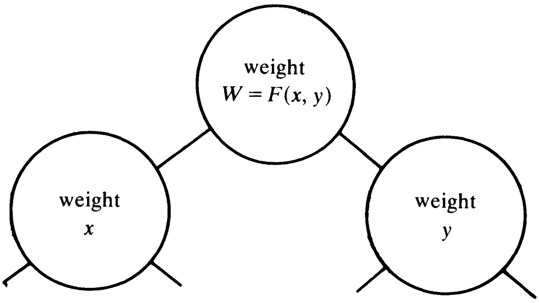


FIG. 1. *Weight combination function* $F(x, y)$.

DEFINITION. A *tree cost function* G: $U^n \to R$ for all trees having n internal nodes is any symmetric mapping of U^n into the real numbers R. For such a tree T, the cost of T will be

$$G(W_1(T), W_2(T), \cdots, W_n(T)),$$

i.e., the value of G applied to the internal node weights of T. Note that if such a tree cost is to be generally useful, it should be extensible to arbitrary numbers of arguments and not be dependent on some fixed value of n.

Huffman's algorithm for binary tree construction is now simple to state: *To build the Huffman tree given a weight combination function F, merge at each step the two available nodes of smallest weight* (with ties resolved arbitrarily) *until only one node is available.*

Example. If $F(x, y) = x + y$ and $G = \text{sum}$ with $U = R_+$, then it is not hard to show that the cost of any tree T in this system is

$$\sum_{1 \leqq j \leqq n+1} w_j(T) l_j(T),$$

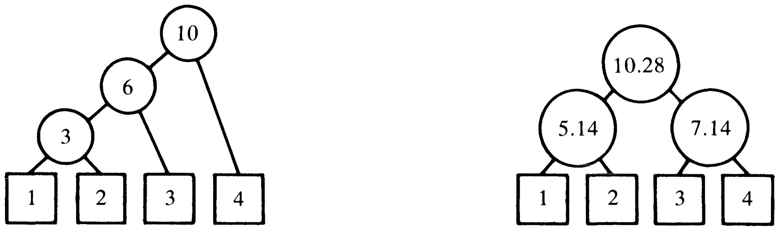
which is called the *weighted path length* of T.

Example. If $F(x, y) = \max(x, y) + c\,(c \geqq 0)$ and $G = \max$ with $U = R_+$, then the cost of any tree T in this system is

$$\max_{1 \leqq j \leqq n+1} (w_j(T) + c \cdot l_j(T)).$$

We call this a *tree-height measure* of T because when $c = 1$ and $w_j = 0$ for $j = 1, \cdots, n+1$ then this cost is exactly the height of T.

The importance of Huffman's algorithm is that it produces optimal trees for both of these examples. The optimality in the weighted path length system (the one originally considered by Huffman) is proved, e.g., by Zimmerman [31]. Zimmerman's proof *mutatis mutandis* will work for the tree-height example as well. Examples of the construction are shown in Fig. 2. In both cases the trees illustrated are the unique optimal-cost trees; note that although they have identical initial weights their structures are entirely different.



(a) *weight* (*root*) = *weight* (*left son*) + *weight* (*right son*)

(b) *weight* (*root*) = max (*weight* (*left son*), *weight* (*right son*)) + 3.14

FIG. 2. *Tree construction.*

3. Motivation: "Quasilinear" weight combination functions. This section explores the properties of quasilinear functions, and shows that whenever F is quasilinear then the Huffman algorithm will produce a "good" tree. This result motivates the work of § 4, in which the behavior of quasilinear F with Huffman's algorithm is analyzed in detail.

Let $\phi: U \to R$ be a real-valued function on the weight space.

DEFINITION. ϕ is:

i) *positive* if $\phi(x) \geqq 0$ for all x in U;

ii) *negative* if $-\phi$ is positive;

iii) *sign-consistent* if ϕ is positive or negative;

iv) *monotone* if $(\phi(x)-\phi(y))(x-y)$ is a positive function of x for every fixed value y in U; and

v) *strictly monotone* if ϕ is monotone and $x \neq y$ implies $\phi(x) \neq \phi(y)$.

DEFINITION. A weight combination function F: $U^2 \to U$ is:

i) *increasing* if $y \leqq z$ implies $F(x, y) \leqq F(x, z)$ for all x, y, z in U;

ii) *path-length monotone* if $y \leqq z$ implies $F(F(x, y), z) \leqq F(F(x, z), y)$ for all x, y, z in U;

iii) *nonshrinking* if $F(x, y) \geqq \max(x, y)$ for all x, y in U;

iv) *strictly shrinking* if $F(x, y) \leqq \min(x, y)$ for all x, y in U; and

v) *quasilinear* if $F(x, y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ where λ is a nonzero constant and ϕ: $U \to R$ is invertible. (Note F is symmetric, and conjugate under ϕ to the linear map $\lambda(x+y)$.)

The quasilinear functions have a number of interesting properties. They have been studied in the context of functional equations by Aczél and others [1]. We will be concerned with their properties as weight combination functions. In particular, when $\lambda = 1$ and $\phi(x) = x$ the quasilinear function F obtained is

$$F(x, y) = x + y,$$

which is the weight combination function for the weighted path-length construction system. Also when $\lambda = \exp(pc)$ $[c \geqq 0]$ and $\phi(x) = \exp(px)$, then

$$\lim_{p \to \infty} F(x, y) = \max(x, y) + c,$$

which is the weight combination function for the tree-height system. Thus the class of quasilinear functions F is broad enough, in the limit at least, to encompass the two known Huffman-optimal ones.

Assuming F is quasilinear, further restrictions must be imposed on it before useful results can be proved. We require F to have the following characteristics:

(1) F must be continuous;
(2) F must map $U \times U$ into U;
(3) F must be path-length monotone.

These conditions lead to restrictions on ϕ and λ. The first is achieved by restricting ϕ to be *continuous*. We may also assume without loss of generality that ϕ is *strictly increasing*, since ϕ must be strictly monotone to be invertible, and F is invariant of changes to the sign of ϕ.

The second and third conditions may be simultaneously satisfied by making the restrictions on ϕ and λ stated in the following Lemma.

LEMMA 1. *Let* $\phi: U \to R$ *be strictly increasing and* $\lambda > 0$. *If* $F(x, y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ *is to satisfy* $F: U^2 \to U$ *and be path-length monotone, then necessarily* $\lambda \geqq 1$, *and* $\phi(U)$ *must be unbounded.*

Proof. Since ϕ is increasing, ϕ^{-1} is also increasing. If F is path-length monotone then for all x, y, z in U with $y \leqq z$ we find

$$F(F(x, y), z) \leqq F(F(x, z), y),$$

$$\lambda^2\phi(x) + \lambda^2\phi(y) + \lambda\phi(z) \leqq \lambda^2\phi(x) + \lambda^2\phi(z) + \lambda\phi(y),$$

$$0 \leqq (\lambda^2 - \lambda)(\phi(z) - \phi(y)).$$

Since ϕ is increasing we must have $(\lambda^2 - \lambda) \geqq 0$, implying $\lambda = 0$ (trivial) or $\lambda \geqq 1$, as desired. Similarly if F: $U^2 \to U$ then we must have $\lambda(\phi(U) + \phi(U)) \subseteq \phi(U)$. Since $\lambda \geqq 1$, $\phi(U)$ must be unbounded.

Taking under consideration the restrictions posed by Lemma 1, we have the following list of properties enjoyed by quasilinear F for all u, v, x, y in U:

QL1. (Symmetry) $F(x, y) = F(y, x)$;
QL2. (Increasingness) $F(u, x) \leqq F(u, y)$ if $x \leqq y$;
QL3. (Path-length monotonicity) $F(F(u, x), y) \leqq F(F(u, y), x)$ if $x \leqq y$;
QL4. (Bisymmetry) $F(F(u, v), F(x, y)) = F(F(u, x), F(v, y))$.

Property QL4 is very important, since it leads to the proof of the following lemma.

LEMMA 2. *F is a quasilinear weight combination function satisfying Lemma* 1 if and only if *F satisfies conditions* QL1–QL4.

Proof. The only if part is now obvious. Aczél establishes the other part of the proof in § 6.4 of [1], by showing that the continuous solutions of the bisymmetry functional equation QL4 are in fact quasilinear functions.

LEMMA 3. *Let F be as in Lemma* 1. *Then F is nonshrinking or strictly shrinking if and only if* ϕ *is sign consistent. Specifically, F is nonshrinking iff* ϕ *is positive* (*increasing*), *and F is strictly shrinking iff* ϕ *is negative* (*increasing*).

Proof. Since ϕ is increasing we have

$$F(x, y) \geqq \max(x, y) \quad \text{iff } \lambda\phi(x) + \lambda\phi(y) \geqq \phi(x) \quad \text{for all } x \geqq y \text{ in } U,$$

$$F(x, y) \leqq \min(x, y) \quad \text{iff } \lambda\phi(x) + \lambda\phi(y) \leqq \phi(x) \quad \text{for all } x \leqq y \text{ in } U.$$

Neither of these can be true if ϕ is not sign-consistent, for then the connectivity of the interval U and the continuity of ϕ imply a neighborhood of zero would exist in $\phi(U)$; consequently we could contradict the first inequality above by selecting

$$\phi(x) > 0, \qquad \phi(y) = -\phi(x), \quad \text{giving } \lambda \cdot 0 = 0 \geqq \phi(x).$$

We conclude that ϕ must be sign-consistent, and find the nonshrinking/strictly shrinking condition of F is satisfied when

$$\lambda \geqq \sup_{x,y} \frac{\phi(x)}{\phi(x)+\phi(y)} \quad \text{and} \quad \begin{array}{l} \phi \text{ is negative } [F \leqq \min], \\ \phi \text{ is positive } [F \geqq \max], \end{array}$$

which is always true since $\lambda \geqq 1$ by Lemma 1.

DEFINITION. A weighted tree is an *F-tree* if the weight of each internal node is the F-value of the weights of its sons.

DEFINITION. A weight combination function F is *Huffman monotone* if for all F-trees T:

(a) When R is an F-tree obtained by interchanging any two of T's leaves which have the same path length (and then recomputing internal node values) the root weight of R equals the root weight of T.

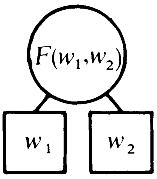
(b) When S is an F-tree obtained from T by interchanging a node having weight u with a node of lesser depth having weight $v \geqq u$ (resp. $v \leqq u$), then the root weight of S is greater (resp. smaller) than the root weight of T.

THEOREM 1. *If F is Huffman monotone, then Huffman's algorithm produces a tree with the minimal root weight.*

Proof. By induction on the number of leaves in the tree. As a basis the theorem is true for 2 leaves, so assume it true for n. Let T be a tree with $n+1$ leaves such that T is better than the Huffman tree S. Assuming the leaf weights are

$$w_1 \leqq w_2 \leqq \cdots \leqq w_{n+1}$$

then T cannot contain the subtree



since the Huffman tree S contains this subtree and has, by hypothesis, the minimal root weight for the n-weight set $\{F(w_1, w_2), w_3, \cdots, w_{n+1}\}$. Let w_r and w_s be the leaf weights of the binary subtree of maximal depth in T. That is, let w_r and w_s have maximal path length in T. Then since F is Huffman monotone the tree $\bar{T}$ obtained from T by first interchanging w_1 and w_r, then w_2 and w_s, cannot have a root weight any greater than T's. But as we just showed the induction hypothesis says the root weight of S is no worse than that of $\bar{T}$. This contradicts the assumption that T was better than S.

THEOREM 2. *F is Huffman monotone* if and only if *F satisfies properties* QL3 *and* QL4.

Proof. The only if part follows by definition. The converse may be shown by direct manipulation of "F-expressions", using induction on the depth of the tree.

Note that QL3 and QL4 give the converse's basis for trees of depth 2. Assume inductively that if u and v are leaf weights of equal depth in an F-tree T of depth n or less, then interchanging u and v does not change T's root weight. Denote this fact by

$$F(T_1[u], T_2[v]) = F(T_1[v], T_2[u]),$$

where T_1 and T_2 are subtrees of T of depth $n-1$ or less containing u and v, respectively. Now consider a tree T' of depth $n+1$ with leaf weights u, v of equal path length $n+1$.

(If their path length is less than this, it is obvious from the induction that they can be exchanged without altering the root weight of T'.) T' then has 4 subtrees T_1, T_2, T_3, T_4 of depth $n-1$ or less. Suppose without loss of generality that T_2 contains u and T_3 contains v (again, if u and v are in the same subtree then the interchange property follows from induction). But then the root weight of T' is

$$\begin{aligned} F(F(T_1, T_2[u]), F(T_3[v], T_4)) &= F(F(T_1, T_4), F(T_2[u], T_3[v])) \\ &= F(F(T_1, T_4), F(T_2[v], T_3[u])) \\ &= F(F(T_1, T_2[v]), F(T_3[u], T_4)) \end{aligned}$$

which says precisely that interchanging u and v does not affect the root weight of T'. This establishes the (a) part of Huffman monotonicity; the (b) part may be proved analogously.

Lemmas 1 and 2 and Theorems 1 and 2 finally give us the following corollary.

COROLLARY 1. *F is Huffman-monotone if and only if F is quasilinear and satisfies the restrictions of Lemma* 1.

The Huffman-monotonicity property is important in that it guarantees (Theorem 1) a minimal root weight for the Huffman tree. If the cost of a tree is determined by its root weight, clearly the Huffman tree will be optimal. Corollary 1 provides the incentive for the examination in the next section of the properties of F-trees for quasilinear F.

4. General characterization of Huffman tree construction. We begin this section with a result from Glassey and Karp [12], and show how it can be extended in a natural way to characterize the weight structure obtained in trees constructed with the general Huffman algorithm.

DEFINITION. A *weight sequence* **a** is a row of nonnegative numbers $[a_1, a_2, \cdots, a_m]$ such that $a_1 \leqq a_2 \leqq a_3 \leqq \cdots \leqq a_m$.

DEFINITION. Given two weight sequences $\mathbf{a} = [a_1, a_2, \cdots, a_m]$ and $\mathbf{b} = [b_1, b_2, \cdots, b_m]$, we write $\mathbf{a} \leqq \mathbf{b}$ if $\sum_{i=1}^k a_i \leqq \sum_{i=1}^k b_i$ holds for all k, $1 \leqq k \leqq m$.

THEOREM 3 (Glassey and Karp). *Let* $\mathbf{W(S)} = [W_1(S), W_2(S), \cdots, W_n(S)]$ *be the weight sequence for the internal nodes in a tree constructed by the binary Huffman algorithm in the weighted path-length system, and let* $\mathbf{W(T)} = [W_1(T), W_2(T), \cdots, W_n(T)]$ *be the weight sequence for the internal nodes of any other tree on the same leaf weights. Then* $\mathbf{W(S)} \leqq \mathbf{W(T)}$.

Glassey and Karp [12, pp. 371–373] prove Theorem 3 for general r-ary tree construction, where r may be greater than 2 and the trees need not be full. The proof establishes by induction on k that $\sum_1^k W_i(S) \leqq \sum_1^k W_i(T)$, for $1 \leqq k \leqq m$. The theorem is a sharpening of the earlier result by Hu and Tucker that "Huffman's algorithm gives an optimal m-sum forest" in the weighted path-length case [15, p. 518]. Anyway it is an important characterization of the traditional Huffman algorithm and will be the clue to most of the more general results in this section.

DEFINITION. A function ϕ: $U \to R$ is *convex* if U is a convex subset of R and for all x, y in U and t in $[0, 1]$, $\phi\ (tx + (1-t)y) \leqq t \cdot \phi(x) + (1-t) \cdot \phi(y)$; ϕ is *concave* if $-\phi$ is convex.

THEOREM 4. *Let* **a** *and* **b** *be two weight sequences of length m such that* $\mathbf{a} \leqq \mathbf{b}$. *If ϕ is any concave, strictly increasing function and we define $\phi(\mathbf{a})$ to be the weight sequence $[\phi(a_1), \cdots, \phi(a_m)]$ and similarly for $\phi(\mathbf{b})$, then $\phi(\mathbf{a}) \leqq \phi(\mathbf{b})$, i.e.,*

$$\sum_{i=1}^{k} \phi(a_i) \leqq \sum_{i=1}^{k} \phi(b_i) \quad \textit{for } 1 \leqq k \leqq m.$$

Proof. This result is typical in the theory of convex functions. An elegant proof can be adapted from that of Fuchs [9] (see also [21]) for the analogous case where ϕ is convex, and will be omitted here.

It is instructive to note that our partial order $\mathbf{a} \leqq \mathbf{b}$ on weight sequences is equivalent to the "majorization" relation $\mathbf{a} > \mathbf{b}$ of [14], which appears widely in the literature, if and only if $\sum_1^m a_i = \sum_1^m b_i$.

We now extend Theorem 3 for tree construction with a quasilinear weight combination function

$$F(x, y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y)).$$

Section 3 showed that when F is of this form and several other conditions (Lemma 1) are met, Huffman's algorithm produces a tree with minimal root weight. In fact we will make a much more general statement concerning *all* of the constructed internal node weights. First, however, we must make some observations about construction with quasilinear functions restricted by Lemmas 1 and 3.

Note that when ϕ is *positive increasing*, the tree constructed by the Huffman algorithm (for any positive λ) on the leaf weights $\{w_1, \cdots, w_{n+1}\}$ is topologically *isomorphic* to the tree that would be built by the Huffman algorithm with

$$F(x, y) = \lambda(x + y)$$

on the leaf weights $\{\phi(w_1), \cdots, \phi(w_{n+1})\}$. (The values of the internal node weights would also differ by ϕ from tree to tree.) If ϕ is *positive decreasing*, by contrast, the tree constructed by the Huffman algorithm on $\{w_1, \cdots, w_{n+1}\}$ is topologically isomorphic to that which would be built by the *anti-Huffman algorithm* (the tree construction procedure in which the two nodes of greatest weight are merged at each step) with $F(x, y) = \lambda(x + y)$ on the leaf weights $\{\phi(w_1), \cdots, \phi(w_{n+1})\}$. This all follows from the "order-preserving" properties of monotone functions. It should be pointed out that when ϕ is positive decreasing and $\lambda \geqq 1$ the Huffman algorithm could always produce the tree in Fig. 3, because then by Lemma 3 $F(x, y) \leqq \min(x, y)$ and the smallest weight is always selected.

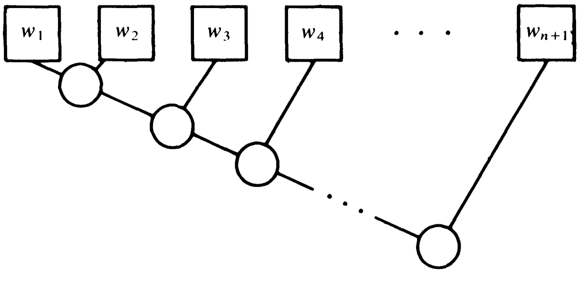


FIG. 3

This is also the structure of the tree that would be produced if ϕ were positive increasing, $\lambda \geqq 1$, and the anti-Huffman algorithm were used, for then we would have $F(x, y) \geqq \max(x, y)$ and the largest weight would always be selected. This type of tree construction is not particularly interesting but will be covered here for the sake of completeness.

Lemma 3 gives the restrictions on ϕ necessary for F to be nonshrinking or strictly shrinking, i.e., $F(x, y) \geqq \max(x, y)$ or $F(x, y) \leqq \min(x, y)$. In some sense the restriction to nonshrinking F is "natural", but here it is made to guarantee that the smallest internal node weights are always found near the leaves of the tree. More precisely, if $F(x, y) \geqq \max(x, y)$, then it is clear that the k smallest internal node weights $[W_1(T), \cdots, W_k(T)]$ of a tree T define a subforest of T. (Suppose some weight $W_i(T)$ in this weight sequence corresponds to an internal node whose son's weight $W_j(T)$ is not also contained in the weight sequence. Then $W_i(T) < W_j(T)$, for otherwise $W_j(T)$ would be contained in the weight sequence. But this is impossible because $F(x, y) \geqq \max(x, y)$ implies $W_i(T) \geqq W_j(T)$.) Thus Lemma 1 asserts that if ϕ is positive (resp. negative) increasing and $\lambda \geqq 1$, the resulting internal node weights will have this subforest characterization: *Every collection of least* (resp. greatest) *node weights define some subforest.*

Lemma 3 also shows that, for nonshrinking or strictly shrinking F, we can assume ϕ is *positive* (and strictly monotone) instead of assuming it is increasing. This is true since F is invariant of sign changes to ϕ, and ϕ must be sign-consistent. This assumption seems to be the natural one to make because of the following result.

THEOREM 5. *Let* $F(x, y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ *be the weight combination function of the tree construction where* ϕ *is* convex, *positive, and strictly monotone and* $\lambda \geqq 1$. *If, as in Theorem* 3, $\mathbf{W(S)}$ *and* $\mathbf{W(T)}$ *are the weight sequences for the internal nodes of the trees S and T, constructed respectively by the Huffman algorithm and by any other way, then*

$$\mathbf{W(S)} \leqq \mathbf{W(T)}.$$

(*The same results hold if* ϕ *is concave, negative, and strictly monotone.*)

Proof. The proof has two cases, accordingly as ϕ is strictly increasing or strictly decreasing.

Case 1. ϕ *convex, positive, and strictly increasing.* We accomplish the proof in two steps: using the notation of Theorem 4, we first show that the weight sequences $\phi(\mathbf{W(S)})$ and $\phi(\mathbf{W(T)})$ satisfy

$$\phi(\mathbf{W(S)}) \leqq \phi(\mathbf{W(T)})$$

and then, since ϕ^{-1} is concave increasing in this case, we can apply Theorem 4 to get $\mathbf{W(S)} \leqq \mathbf{W(T)}$ as desired.

If there are $n+1$ initial leaf weights $w_1, \cdots, w_{n+1}$ we have as above $\mathbf{W(S)} = [W_1(S), \cdots, W_n(S)]$ and $\mathbf{W(T)} = [W_1(T), \cdots, W_n(T)]$ as the internal node weight sequences where W_i is the ith-smallest such weight. In particular since $W_i(S)$ designates the weight of some internal node which is the root of some subtree S_i of the Huffman tree S, if we define

$$\mathcal{S}_i = \{j \mid w_j \text{ is a leaf of } S_i\},$$

$$l_j(S_i) = \text{path length of weight } w_j \text{ in the subtree } S_i,$$

$$a_i = \sum_{j \in \mathcal{S}_i} \lambda^{l_j(S_i)} \phi(w_j)$$

then $W_i(S) = \phi^{-1}(a_i)$. (This equation follows from the remark made after Theorem 4, concerning the topological isomorphism of the Huffman tree here with the Huffman tree using $F(x, y) = \lambda(x+y)$ on the weight set $\{\phi(w_j)\}$.) Defining $\mathcal{T}_i$ and $l_j(T_i)$ in an analogous manner, if we set

$$b_i = \sum_{j \in \mathcal{T}_i} \lambda^{l_j(T_i)} \phi(w_j)$$

then $W_i(T) = \phi^{-1}(b_i)$.

We now claim that $\mathbf{a} = [a_1, \cdots, a_n]$ and $\mathbf{b} = [b_1, \cdots, b_n]$ are weight sequences satisfying $\mathbf{a} \leqq \mathbf{b}$. First of all since ϕ is positive increasing and $\mathbf{W(S)}$, $\mathbf{W(T)}$ are weight sequences, we know that $\mathbf{a}=\phi(\mathbf{W(S)})$ and $\mathbf{b}=\phi(\mathbf{W(T)})$ are weight sequences; in fact since ϕ "preserves order", if $W_i(S)<W_j(S)$ then $a_i=\phi(W_i(S))<\phi(W_j(S))=a_j$, and similarly for $\mathbf{W(T)}$ and $\mathbf{b}$. Second, by Lemma 1 we know that $F(x, y)\geqq x, y$ in this case, so the k smallest node weights $[W_1, \cdots, W_k]$ for either S or T correspond to a subforest F_k of S or T. This implies

$$\begin{aligned}\sum_{i=1}^{k} b_i &= \sum_{i=1}^{k} \sum_{j\in\mathcal{T}_i} \lambda^{l_j(T_i)}\phi(w_j)\\ &= \sum_{j=1}^{n+1} (\lambda+\lambda^2+\cdots+\lambda^{l_j(F_k)})\phi(w_j)\\ &= \begin{cases} \left(\dfrac{\lambda}{\lambda-1}\right)\displaystyle\sum_{j=1}^{n+1} (\lambda^{l_j(F_k)}-1)\phi(w_j) & \text{if } \lambda>1,\\ \displaystyle\sum_{j=1}^{n+1} l_j(F_k)\phi(w_j) & \text{if } \lambda=1,\end{cases}\end{aligned}$$

with a similar expression holding for $\sum_{i=1}^{k} a_i$.

We can now directly apply Glassey and Karp's method of proof for Theorem 3. The proof proceeds by induction on k, proving $\mathbf{a}\leqq\mathbf{b}$ by showing for all k that $\sum_{i=1}^{k} a_i \leqq \sum_{i=1}^{k} b_i$.

Basis: $k=1$ is trivial. Induction step: there are two possibilities, depending on the relationship between $a_1=\phi(W_1(S))$ and $b_1=\phi(W_1(T))$.

Subcase 1. $a_1=b_1$. In this case we know $\phi^{-1}(a_1)=\phi^{-1}(b_1)=F(w_1, w_2)$ and we are reduced to the proof on the set of leaf weights $\{F(w_1, w_2), w_3, \cdots, w_{n+1}\}$, for which we have by induction that

$$\sum_{i=2}^{k} a_i \leqq \sum_{i=2}^{k} b_i.$$

So,

$$\sum_{i=1}^{k} a_i \leqq \sum_{i=1}^{k} b_i.$$

Subcase 2. $a_1<b_1$. As in [GK 76] we will show there is a tree $\bar{T}$ with internal weights $W_i(\bar{T})=\phi^{-1}(c_i)$ where $\mathbf{c}=[c_1, \cdots, c_n]$ satisfies $\sum_{1\leqq i\leqq k} c_i \leqq \sum_{1\leqq i\leqq k} b_i$ and, in addition, $c_1=a_1$ so we have (by reduction to Subcase 1)

$$\sum_{i=1}^{k} a_i \leqq \sum_{i=1}^{k} c_i \leqq \sum_{i=1}^{k} b_i$$

completing the proof that $\mathbf{a}\leqq\mathbf{b}$. Take the forest F_k corresponding to the least k weights $[W_1(T), \cdots, W_k(T)]$ of T and define as before the maximum path length in this forest

$$l_{\max}^{k}=\max_j l_j(F_k).$$

Then choose an internal node having weight $W_p(T)=\phi^{-1}(b_p)=F(w_r, w_s)$ whose 2 (leaf) sons have path length $l_{\max}^{k}$ in F_k and have leaf weights w_r and w_s. Since $a_1<b_1\leqq b_p$ we know $\{w_r, w_s\}\neq\{w_1, w_2\}$. Assuming $w_r\leqq w_s$, let $\bar{T}$ be the tree constructed exactly like T but with the leaf weights w_r and w_1, w_s and w_2 interchanged. Then F_k is still a subforest of $\bar{T}$ (topologically T and $\bar{T}$ are isomorphic) and determines a subset of k of $\bar{T}$'s internal node weights, and consequently some k-subset of the weight sequence $\mathbf{c}$.

Specifically, if we define $\bar{T}_i$, $\bar{\mathscr{T}}_i$, $l_j(\bar{T}_i)$ exactly as above so that

$$c_i = \sum_{j\in\bar{\mathscr{T}}_i} \lambda^{l_j(\bar{T}_i)}\phi(w_j)$$

and $W_i(\bar{T}) = \phi^{-1}(c_i)$, then F_k defines the set

$$\mathscr{I} = \{i \,|\, \phi^{-1}(c_i) \text{ is the weight of some internal node in } F_k\}$$

and $|\mathscr{I}| = k$. Moreover we must have

$$\sum_{i=1}^{k} c_i \leqq \sum_{i\in\mathscr{I}} c_i$$

since the first k weights c_i are the least such weights. But we also have

$$\sum_{i\in\mathscr{I}} c_i \leqq \sum_{i=1}^{k} b_i.$$

To show this we write for convenience

$$\Lambda_1 \equiv \lambda^{l_1(T)} = \lambda^{l_r(\bar{T})}, \qquad \Lambda_2 \equiv \lambda^{l_2(T)} = \lambda^{l_s(\bar{T})},$$

$$\Lambda_m \equiv \lambda^{l^k_{\max}(T)} = \lambda^{l_r(T)} = \lambda^{l_s(T)} = \lambda^{l_1(\bar{T})} = \lambda^{l_2(\bar{T})},$$

$$\phi_r \equiv \phi(w_r), \qquad \phi_s \equiv \phi(w_s), \qquad \phi_1 \equiv \phi(w_1), \qquad \phi_2 \equiv \phi(w_2).$$

Therefore

$$\phi_r \geqq \phi_1 \quad \text{and} \quad \phi_s \geqq \phi_2,$$

and in the case $\lambda > 1$, since

$$l_1(T), l_2(T) \leqq l^k_{\max}(T)$$

we have

$$\Lambda_1 \leqq \Lambda_m \quad \text{and} \quad \Lambda_2 \leqq \Lambda_m.$$

So, if $\lambda > 1$,

$$\begin{aligned}\sum_{i\in\mathscr{I}} c_i - \sum_{i=1}^{k} b_i &= \left(\frac{\lambda}{\lambda-1}\right)[(\Lambda_m\phi_1 + \Lambda_m\phi_2 + \Lambda_1\phi_r + \Lambda_2\phi_s) - (\Lambda_1\phi_1 + \Lambda_2\phi_2 + \Lambda_m\phi_r + \Lambda_m\phi_s)] \\ &= \left(\frac{\lambda}{\lambda-1}\right)[((\Lambda_m - \Lambda_1)(\phi_1 - \phi_r) + (\Lambda_m - \Lambda_2)(\phi_2 - \phi_s))] \\ &\leqq 0.\end{aligned}$$

A trivial modification of this argument gives the proof for $\lambda = 1$, so we omit it here. Thus we have shown

$$\sum_{i=1}^{k} c_i \leqq \sum_{i\in\mathscr{I}} c_i \leqq \sum_{i=1}^{k} b_i$$

but since $c_1 = \phi^{-1}(W_1(\bar{T})) = \phi^{-1}(F(w_1, w_2)) = a_1$ we have, by reduction to Subcase 1, that

$$\sum_{i=1}^{k} a_i \leqq \sum_{i=1}^{k} c_i.$$

Therefore

$$\sum_{i=1}^{k} a_i \leqq \sum_{i=1}^{k} b_i$$

and Theorem 5 follows for Case 1, since we have shown that $\mathbf{a} \leqq \mathbf{b}$, and, since here ϕ^{-1} is concave increasing we can apply Theorem 4 to get immediately

$$\mathbf{W(S)} = \phi^{-1}(\mathbf{a}) \leqq \phi^{-1}(\mathbf{b}) = \mathbf{W(T)}.$$

Case 2. *ϕ convex, positive, strictly decreasing.* Actually in this case a statement stronger than Theorem 5 can be made. We are comparing here the weight sequences

$$\mathbf{W(S)} = [W_1(S), \cdots, W_n(S)] = [\phi^{-1}(a_n), \cdots, \phi^{-1}(a_1)]$$

and

$$\mathbf{W(T)} = [W_1(T), \cdots, W_n(T)] = [\phi^{-1}(b_n), \cdots, \phi^{-1}(b_1)],$$

where $\mathbf{a} = [a_1, \cdots, a_n]$ and $\mathbf{b} = [b_1, \cdots, b_n]$ are weight sequences as in Case 1. From the discussion following Theorem 4 we see that $\mathbf{a}$ would be the internal node weight sequence formed using the anti-Huffman algorithm on the leaf weights $\{\phi(w_1), \cdots, \phi(w_{n+1})\}$. It follows that $a_k \geqq b_k$ for $1 \leqq k \leqq n$, and thus we easily have both $\mathbf{a} \geqq \mathbf{b}$ and $\mathbf{W(S)} \leqq \mathbf{W(T)}$ as consequences.

Theorem 5 is possibly the most general result of its kind. To show what can happen when ϕ is not convex, we consider an example where ϕ is concave positive. Let $\phi(x) = \sqrt{x}$, $\lambda = 1$, and $U = R_+$ so that

$$F(x, y) = (\sqrt{x} + \sqrt{y})^2,$$

and suppose we are to build a tree given the leaf weights $\{1, 2, 3, 4\}$. The Huffman algorithm produces the tree S in Fig. 4a

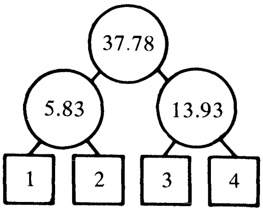


FIG. 4a

for which we have $\sum_{i=1}^{3} W_i(S) = 5.83 + 13.93 + 37.78 = 57.73$, while the tree T in Fig. 4b

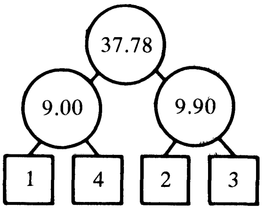


FIG. 4b

has $\sum_{i=1}^{3} W_i(T) = 9 + 9.90 + 37.78 = 56.68$, so $\mathbf{W(S)} \not\leqq \mathbf{W(T)}$. That this phenomenon will always happen when ϕ is not convex is a result of the *converse* of Theorem 4, which says that

$$\sum_{i=1}^{m} \phi(a_i) \leqq \sum_{i=1}^{m} \phi(b_i) \quad \text{for all } \mathbf{a} \leqq \mathbf{b} \Rightarrow \phi \text{ concave increasing.}$$

The proof is easy and we omit it.

An interesting problem is to determine, given only an expression for $F(x, y)$, whether F satisfies the conditions of Theorem 5. Lemma 2 can be used to establish whether F is quasilinear; but it may be hard to determine whether the function ϕ defined by F is convex positive, since explicit form for ϕ may be hard to produce. A good solution to this problem remains to be found. Currently the only method known is to derive a power series for ϕ^{-1} either by repeated differentiation of the functional equation

$$F(\phi^{-1}(x), \phi^{-1}(x)) = \phi^{-1}(2\lambda x)$$

followed by equating of coefficients, or (better) by using iterative methods like the ones in [3] to converge to truncated series, and finally to analyze the sign of this series and of its second derivative. This approach assumes, of course, that F and ϕ are analytic functions.

There is a test that can be made, however. We claim that if F is differentiable then

THEOREM 6. *$\partial F/\partial x$ and $\partial F/\partial y$ must be bounded on U if F is to satisfy the conditions of Theorem* 5.

Proof.

$$\frac{\partial F}{\partial x}(x, y) = \frac{d\phi^{-1}}{dx}(\lambda\phi(x) + \lambda\phi(y)) \cdot \lambda \frac{d\phi}{dx}(x)$$

$$= \lambda \frac{d\phi}{dx}(x) \Big/ \frac{d\phi}{dx}(F(x, y))$$

because $d\phi^{-1}/dx = 1/(d\phi/dx)(\phi^{-1}(x))$. If ϕ is strictly increasing positive then (Lemma 3) $F(x, y) \geqq \max(x, y)$ and $d\phi/dx > 0$; if ϕ is strictly decreasing positive then $F(x, y) \leqq \min(x, y)$ and $d\phi/dx < 0$. So if ϕ is *convex* increasing positive then $d\phi/dx$ is positive increasing, in which case

$$\lambda \frac{d\phi}{dx}(x) \Big/ \frac{d\phi}{dx}(F(x, y)) \leqq \lambda \frac{d\phi}{dx}(x) \Big/ \frac{d\phi}{dx}(\max(x, y)) \leqq \lambda.$$

If ϕ is convex decreasing positive we obtain the same bound using $\min(x, y)$ since then $|d\phi/dx|$ is positive decreasing.

Unfortunately, the condition of Theorem 6 does not imply ϕ *must* be convex, since it is true for lumpy, but near-convex, functions. It does appear to be a fairly potent test, however; for the example in Fig. 4, we find $\partial F/\partial x = 1 + (y/x)^{1/2}$, which is unbounded on $U = R_+$. This condition seems to characterize when the Huffman algorithm works: If F grows too quickly, then the algorithm makes mistakes in its "greedy" selection of nodes to merge.

Theorem 5 and Theorem 1/Corollary 1 will be exploited in § 5. We finish this section by formulating the above characterization for the r-ary case.

THEOREM 7. *Let everything be defined as in Theorem* 5, *with the exception that we let* $F: U^r \to U$ *be the r-ary function* $(r \geqq 2)$,

$$F(x_1, x_2, \cdots, x_r) = \phi^{-1}\left(\lambda \sum_{i=1}^{r} \phi(x_i)\right).$$

Then the results of Theorem 5 *and Theorem* 1/*Corollary* 1 *still hold.*

We omit the proof, which is virtually identical to that of these theorems, with the changes that we must now define F on less than its full r arguments in the natural way. In the binary case all constructed trees are full, but that is no longer true in the r-ary case. If $n+1$ leaf weights are provided, the Huffman algorithm selects exactly the $2 + [(n-1) \bmod (r-1)]$ smallest weights for the first weight combination, and this quantity is not necessarily equal to r; however choosing this many weights guarantees that all future weight combinations can merge r weights. The details of showing that a tree $\bar{T}$ gives us inequalities like $\mathbf{W(S)} \leqq \mathbf{W(\bar{T})} \leqq \mathbf{W(T)}$ are slightly more complicated but no different in method. These details are covered in Glassey and Karp's proof in [12].

5. Cost functions under which Huffman trees are optimal. In §§ 3 and 4 we described the properties of the internal node weights in Huffman trees with quasilinear weight combination functions F. In this section we exploit these results as much as possible and exhibit several classes of tree cost functions for which the appropriate Huffman trees are optimal. As indicated above in § 2, we are considering cost a function of the constructed internal node weights, so formally

$$\text{Cost}(T) = G(\mathbf{W(T)}) = G(W_1(T), \cdots, W_n(T)).$$

Thus $G: U^n \to R$ is to be a function under which Huffman internal node weight sequences have smallest image. We show now that cost functions that are "Schur concave" (defined momentarily) are important when all the internal node weights $W_i(T)$ are to be taken into consideration. If one is only interested in max $W_i(T)$ or min $W_i(T)$ (thus: $W_n(T)$ or $W_1(T)$, depending on whether $|\phi|$ is increasing or decreasing, where ϕ is the usual function defining F), then the cost function need only be increasing. These cost functions are apparently the most general possible for the Huffman construction to be optimal when the weight combination function F is quasilinear. Applications will be discussed in the next section.

DEFINITION. A function $G: U^n \to R$ is *Schur concave* if

$$(x_i - x_j)\left(\frac{\partial G}{\partial x_i} - \frac{\partial G}{\partial x_j}\right) \leqq 0 \quad \text{for all } x_i, x_j \in U, \quad i, j \in \{1, \cdots, n\}.$$

THEOREM 8. (Schur and Ostrowski). $G(\mathbf{a}) \leqq G(\mathbf{b})$ *for all weight sequences* $\mathbf{a} \leqq \mathbf{b}$ *if and only if G is Schur concave.*

A proof adapted from [27] and [23] appears in Appendix A. It is worth mentioning that all strictly concave functions G (so $G'' < 0$) are Schur concave—see [27, p. 12]. Generally speaking the importance of this theorem has not been properly appreciated; recently Wong and Yue have found a number of uses for it in storage applications. See for example [30].

The next three theorems follow as corollaries from Theorem 8 and §§ 3 and 4. In each we compare the cost of trees S and T built using a quasilinear weight combination function F, where S is the tree built by the Huffman algorithm and T is any other tree. As usual, $\mathbf{W(S)} = [W_1(S), \cdots, W_n(S)]$ and $\mathbf{W(T)} = [W_1(T), \cdots, W_n(T)]$ denote the internal node weight sequences for these trees.

THEOREM 9. *Let F be as in Theorem 5. Then the Huffman tree will have least cost when G is any Schur concave function of the internal node weights.*

Proof. This is a simple corollary of Theorem 8, since Theorem 5 guarantees $\mathbf{W}(\mathbf{S}) \leqq \mathbf{W}(\mathbf{T})$, so $G(\mathbf{W}(\mathbf{S})) \leqq G(\mathbf{W}(\mathbf{T}))$.

THEOREM 10. *Let $F(x, y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ with $\lambda \geqq 1$ and ϕ positive monotone continuous, as in Lemma 3. Then the Huffman tree will have least cost when G is a function of the following form:*

If ϕ is increasing, $G = \tilde{G} \cdot \phi$ where $\tilde{G}$ is Schur concave.

If ϕ is decreasing, $G = \tilde{G} \cdot \phi$ where $\tilde{G}$ is monotone decreasing (i.e., $\tilde{G}(x_1, \cdots, x_i, \cdots, x_n) \leqq \tilde{G}(x_1, \cdots, x_i', \cdots, x_n)$ if $x_i \geqq x_i'$).

Proof. Note $G(\mathbf{W}(\mathbf{T})) = \tilde{G}(\phi(\mathbf{W}(\mathbf{T}))) = \tilde{G}([\phi(W_1(T)), \cdots, \phi(W_n(T))])$. Using an argument as in the proof of Theorem 5, it is clear that if ϕ is increasing then $\phi(\mathbf{W}(\mathbf{S})) \leqq \phi(\mathbf{W}(\mathbf{T}))$, and, if ϕ is decreasing, then not only $\phi(\mathbf{W}(\mathbf{S})) \geqq \phi(\mathbf{W}(\mathbf{T}))$ but also $\phi(W_i(S)) \geqq \phi(W_i(T))$ for $i = 1, \cdots, n$. Theorem 8 gives us the first part of the theorem; the second follows from the monotonicity of $\tilde{G}$.

THEOREM 11. *Let F be as in Lemma 1. The Huffman tree will have least cost when G is of the form $G(\mathbf{W}(\mathbf{T})) = \psi(\max W_i(T))$ or $G(\mathbf{W}(\mathbf{T})) = \psi(\min W_i(T))$, where ψ is any monotone increasing function.*

Proof. Immediate from Theorem 1/Corollary 1.

Although Schur concave G's are the only cost function we discuss here, it should be clear that there may be other Huffman-optimal ones. The functions here prey on the properties of the Huffman tree internal node weights; discovery of other properties could lead to other cost criteria favorable for Huffman trees. We emphasize also that varying the weight space U can greatly affect the performance of the Huffman algorithm. Consider the weight combination function $F(x, y) = xy$. On $U = [0, 1]$ we can take $\phi(x) = -\log(x)$, a positive *convex decreasing* function (the base of the logarithm is immaterial); from Theorem 9 we know that under cost functions like $G = \text{sum}$, Huffman trees will be optimal. However, on $U = [1, \infty)$ we have $\phi(x) = +\log(x)$, a positive *concave increasing* function, so under the cost $G = \text{sum}$ there is no guarantee that a Huffman tree will be best. Even worse, if we chose $U = [0, \infty)$ there is then no sign-consistent, strictly monotone function ϕ determined by F. This shows that some of the above theorems are more restrictive than they appear at first.

6. Applications and open problems. We have just shown that for wide classes of tree construction systems the Huffman algorithm produces optimal trees. We shall now discuss applications.

We *prove* first that Huffman construction in the tree height system

$$F(x_1, \cdots, x_r) = \max(x_1, \cdots, x_r) + c \qquad (c > 0),$$

$$G(\mathbf{W}(\mathbf{T})) = \max W_i(T)$$

is optimal. The demonstration was hinted at in §3: Consider the family of functions

$$F(x_1, \cdots, x_r) = \phi^{-1}\left(\lambda \sum_{i=1}^{r} \phi(x_i)\right)$$

and

$$G(\mathbf{W}(\mathbf{T})) = \phi^{-1}(\sum \phi(W_i(T))) \quad \text{or} \quad G(\mathbf{W}(\mathbf{T})) = \max W_i(T),$$

where $\phi(x) = r^{px}$, $\lambda = r^{pc}$. Then ϕ is convex increasing and $\lambda \geqq 1$, so Theorems 10 or 11 imply Huffman trees will have least cost. Since in the limit as $p \to \infty$ we approach the max functions F and G of the tree height construction system, we have established that Huffman's algorithm is in this case optimal.

Another application of Huffman tree construction is the generation of *codes* which are optimal under criteria other than Huffman's original one, equivalent to weighted path-length [16]. A moderate literature has grown up around this subject ([4], [5], [2], [22], etc.). It is surprising that no corresponding analogue of Huffman's algorithm has also been developed. We outline several known results, including interesting bounds on average codeword length like that of the noiseless coding theorem, and then present these Huffman analogues.

In the context of coding, the leaf weights $\{w_1, \cdots, w_{n+1}\}$ are probabilities (so $\sum w_j = 1$), representing the relative frequencies of occurrence of a set of $(n+1)$ messages which are to be encoded into r-ary codewords ($r \geqq 2$). Let the length of the message with probability w_j be called l_j; we are then interested in minimizing the "quasiarithmetic mean codeword length" [2], [5],

$$L(\mu, \{w_j\}, \{l_j\}) = \mu^{-1}\left(\sum_{j=1}^{n+1} w_j\mu(l_j)\right)$$

or some similar code cost measure; here μ is a continuous, strictly increasing function on R_+. For example, when $\mu(x) = x$ we get the traditional weighted path-length; other "translative" forms of L have been considered in [5], [2] and [22]. Although this measure of codeword length is quite general, most special cases treated in the literature can be handled by the extended Huffman construction presented here. We consider three cases one by one; each is based on Rényi's entropy of order α

$$H_\alpha(w_1, \cdots, w_{n+1}) = \frac{1}{1-\alpha} \log_r\left(\sum_{j=1}^{n+1} w_j^\alpha\right).$$

Here r is the size of the code letter alphabet, i.e., codewords can be viewed as r-ary numbers. Rényi's entropy has the interesting property that its limit, as $\alpha \to 1$, is the usual Shannon entropy

$$H(w_1, \cdots, w_{n+1}) = -\sum_{j=1}^{n+1} w_j \log_r (w_j).$$

Campbell [4] now defines an exponential codeword length average $L(t)$ by setting $\mu(x) = r^{tx}$ so that

$$L(t) = \frac{1}{t} \log_r \left(\sum w_j r^{tl_j}\right) = \log_\lambda \left(\sum w_j \lambda^{l_j}\right),$$

where $t > 0$ and $\lambda = r^t > 1$. He then proves that

(1) $$\lim_{t \to 0} L(t) = \sum_j w_j l_j,$$

(2) $$\lim_{t \to \infty} L(t) = \max_j l_j,$$

(3) $$H_\alpha(w_1, \cdots, w_{n+1}) \leqq L(t) \quad \text{where } \alpha = \frac{1}{1+t} = \frac{1}{1+\log_r(\lambda)}$$

with equality holding when $r^{-l_j} = w_j^\alpha / (\sum w_j^\alpha)$. Now consider general Huffman construction as discussed in § 2 with $F(x, y) = \lambda(x+y)$ and $G(\mathbf{W(T)}) = \log_\lambda (W_n(T))$. Then

$$\text{Cost}(T) = G(\mathbf{W(T)}) = L(t) = L(\log_r(\lambda)),$$

so Huffman construction with this weight combination function F produces optimal exponential-length-cost trees by Theorem 11.

Aczél [2], besides citing results of Campbell for the degenerate case $t<0$ $(\lambda<1)$ above, considers the result when $\mu(x)=(\lambda^x-1)/(\lambda-1)$ (again, $\lambda=r^t$) and shows that

$$L(\mu)=\mu^{-1}\left(\sum_{j=1}^{n+1} w_j\mu(l_j)\right)$$

satisfies $((\sum w_j^\alpha)^{1/\alpha}-1)/(\lambda-1)\leqq\mu(L(\mu))$, where again $\alpha=1/(1+t)=1/(1+\log_r\lambda)$. But notice that when $F(x,y)=\lambda(x+y)$ and $G(\mathbf{W}(\mathbf{T}))=\mu^{-1}((1/\lambda)\sum W_i(T))$, then because $\mu(m)=1+\lambda+\cdots+\lambda^{m-1}$ $(m\in\mathbb{Z}_+)$,

$$\text{Cost}\,(T)=G(\mathbf{W}(\mathbf{T}))=L(\mu).$$

So, by Theorem 9, since G is Schur concave, Huffman construction with this function F again produces the optimal code tree (identical to the one constructed for Campbell's average codeword length).

Lastly, Nath has come up with nice results by defining what he calls the average codeword length of order $(\alpha>1)$ [22],

$$L(\alpha)=(\alpha-1)^{-1}\log_r\left(\sum_{j=1}^{n+1} w_j^\alpha r^{(\alpha-1)l_j}\Big/ w^\alpha\right)$$
$$=\log_\lambda\left(\sum_{j=1}^{n+1} w_j^\alpha\lambda^{l_j}\Big/ w^\alpha\right),$$

where $w^\alpha=\sum w_j^\alpha$ and $\lambda=r^{(\alpha-1)}$. He shows that $H_\alpha(w_a,\cdots,w_{n+1})\leqq L(\alpha)$ with equality iff $w_j=r^{-l_j}$ for all j. Now when $F(x,y)=(\lambda x^\alpha+\lambda y^\alpha)^{1/\alpha}$ and $G(\mathbf{W}(\mathbf{T}))=\log_\lambda(W_n(T)^\alpha/w^\alpha)$ we find Cost $(T)=G(\mathbf{W}(\mathbf{T}))=L(\alpha)$. So by Theorem 11 Huffman construction with this function F produces optimal trees here, i.e., produces code trees of least average length $L(\alpha)$.

In addition to this unification of abstract coding problems, we have the following bounds on the costs of Huffman trees:

THEOREM 12. *If $F(x_1,\cdots,x_r)=\sum x_i$, then the weighted path length $\sum w_jl_j=\sum W_i$ of the Huffman tree satisfies*

$$w\cdot H<\sum w_jl_j<w\cdot(H+1),$$

where $H=H(w_1/w, w_2/w,\cdots,w_{n+1}/w)$, $w=\sum w_j$ and $l_1,l_2,\cdots,l_{n+1}$ are the path lengths of $w_1,\cdots,w_{n+1}$ in the Huffman tree. Equality on the left is achieved iff $w_j=r^{-l_j}$ for all j.

Proof. This is the noiseless coding theorem. See, e.g., [10, pp. 50–55].

THEOREM 13. *If W is the root node weight produced by Huffman construction with*

$$F(x_1,\cdots,x_r)=\phi^{-1}\left(\lambda\sum_{i=1}^{r}\phi(x_i)\right)\qquad(\lambda>1),$$

where ϕ is a positive, increasing function, then

$$\phi^{-1}(w^\phi\lambda^H)\leqq W\leqq\phi^{-1}(w^\phi\lambda^{H+1}),$$

where $H=H_\alpha(\phi(w_1)/w^\phi,\cdots,\phi(w_{n+1})/w^\phi)$, $\alpha=1/(1+\log_r(\lambda))$ and $w^\phi=\sum\phi(w_j)$, with equality holding iff

$$r^{-l_i}=\phi(w_i)^\alpha\Big/\left(\sum_{j=1}^{n+1}\phi(w_j)^\alpha\right)\quad\text{for all } i.$$

Proof. Given in [24], it follows easily from Campbell's proof of his lower bound in [5].

COROLLARY 2. *If W is the root node weight produced by r-ary Huffman construction with the weight combination function $F(x_1, \cdots, x_r) = \max(x_1, \cdots, x_r) + c$, then*

$$B \leqq W < B + c,$$

where $B = \log_r ((\sum_{j=1}^{n+1} r^{w_j/c})^c)$. Moreover, equality holds on the left if and only if

$$l_j = (B - w_j)/c.$$

Proof. Take $\phi(x) = r^{px}$, $\lambda = r^{pc}$ in Theorem 13 and let $p \to \infty$. This extends the work of Golumbic [13], who derived this bound for the case where $c = 1$ and all weights w_j are nonnegative integers.

Another application of this generalized construction is in the generation of optimal *search* trees. Hu and Tucker [15] showed that a constrained version of the Huffman algorithm using the sum weight combination function determines the structure of an optimal binary search tree. Tamaki [28] has extended this result by proving an analogue of Theorem1/Corollary 1/Theorem 11: If cost is determined by the tree's root weight, then the Hu-Tucker modification of Huffman's algorithm produces an optimal binary search tree whenever a nonshrinking weight combination function F satisfying conditions QL1, QL3, QL4 of § 3 is used. Thus (Lemma 2) when F is quasilinear, Huffman-like construction still produces the best binary tree.

Other possible applications of this theory being investigated currently include the construction of optimal restricted-height trees (a harder problem than that of restricted-height search trees discussed in [11] and [17], since no obvious dynamic programming solution exists) and the construction of optimal weighted trees where the weights are vectors with multiple components.

There are several open problems. First it would be nice if there were some criterion better than Theorem 6 which would enable us to determine whether F satisfies the requirements of Theorem 5 without having to know explicitly what the conjugation function ϕ is. Secondly, it is natural to ask whether there are other nontrivial construction systems, apart from those considered here, which are optimal under the Huffman algorithm—or whether we have categorized the most general circumstances under which Huffman construction is optimal. That F must *necessarily* be quasilinear if G is Schur concave, etc., is plausible from § 3 yet seems difficult to prove.

Appendix A.

Proof of Theorem 8. $G(\mathbf{a}) \leqq G(\mathbf{b})$ is true for all $\mathbf{a} \leqq \mathbf{b} \Leftrightarrow G$ Schur concave.

($\Rightarrow$(Schur)). Select any $\mathbf{a} = [a_1, \cdots, a_n]$ such that $a_1 \leqq \cdots \leqq a_n$, and set $b_1 = (1-\varepsilon)a_1 + \varepsilon a_2$, $b_2 = \varepsilon a_1 + (1-\varepsilon)a_2$, and $b_i = a_i$ for $i > 2$. Then for $\varepsilon \leqq 1/2$ we have $b_1 \leqq b_2$ and $\mathbf{a} \leqq \mathbf{b}$. Moreover,

$$\frac{G(\mathbf{b}) - G(\mathbf{a})}{\varepsilon} = \frac{G((1-\varepsilon)a_1 + \varepsilon a_2, \varepsilon a_1 + (1-\varepsilon)a_2, a_3, \cdots) - G(a_1, a_2, a_3, \cdots)}{\varepsilon}$$

$$= \frac{1}{\varepsilon}((\partial G/\partial x_1)(\alpha_1, a_2, a_3, \cdots, a_n) \cdot \varepsilon(a_2 - a_1)$$

$$+ (\partial G/\partial x_2)((1-\varepsilon)a_1 + \varepsilon a_2, \alpha_2, a_3, \cdots) \cdot \varepsilon(a_1 - a_2)),$$

where $\alpha_1 \in [a_1, (1-\varepsilon)a_1 + \varepsilon a_2]$ and $\alpha_2 \in [a_2, \varepsilon a_1 + (1-\varepsilon)a_2]$, by the mean value theorem. As ε approaches zero the right hand side approaches

$$-(a_2-a_1)\cdot\left(\frac{\partial G}{\partial x_2}(\mathbf{a})-\frac{\partial G}{\partial x_1}(\mathbf{a})\right).$$

Thus if we are to have $G(\mathbf{a})\leqq G(\mathbf{b})$ this quantity must be positive, so G must be Schur concave, since this argument can be repeated for all pairs of indices i and j (not just 1 and 2).

($\Leftarrow$ (Ostrowski)). Given G is Schur concave, fix $\mathbf{b}$ and assume that there is an $\mathbf{a}\leqq\mathbf{b}$ such that $G(\mathbf{a})>G(\mathbf{b})$. In particular there will be a maximum such $\mathbf{a}$—so assume without loss of generality that $G(\mathbf{a})$ is a maximum. Select indices k, l, i, and j such that $b_k>b_l$ and $a_i\leqq a_j$, and define a weight sequence $\bar{\mathbf{a}}$ such that $\mathbf{a}\leqq\bar{\mathbf{a}}\leqq\mathbf{b}$ by setting $\bar{a}_i=a_i+\varepsilon(b_k-b_l)$ $\bar{a}_j=a_j+\varepsilon(b_l-b_k)$, and $\bar{a}_m=a_m$ for $m\neq i, j$. [Note: if there are no indices k and l such that $b_k>b_l$, we can construct a new weight sequence $\bar{\mathbf{b}}$ such that $\mathbf{a}\leqq\bar{\mathbf{b}}\leqq\mathbf{b}$ which does have such indices and which can be used to replace $\mathbf{b}$ in this proof.] Now set $\phi(\varepsilon)=G(\bar{\mathbf{a}})$. Then $\phi'(\varepsilon)=(b_k-b_l)((\partial G/\partial x_i)(\bar{\mathbf{a}})-(\partial G/\partial x_j)(\bar{\mathbf{a}}))>0$, which contradicts the supposition that $\mathbf{a}$ was a maximal point. So there can be no point $\mathbf{a}\leqq\mathbf{b}$ such that $G(\mathbf{a})>G(\mathbf{b})$—we must have $G(\mathbf{a})\leqq G(\mathbf{b})$.

Acknowledgment. The author gratefully acknowledges the aid of Dr. David Kuck, who posed the problem examined here, pointed out its pitfalls, and encouraged all the research along these lines. Dr. C. K. Wong made me aware of the results of Schur functions. Thanks are also due to Drs. A. H. Sameh and M. B. Pursley for many enlightening discussions, and to one of the referees whose outstanding suggestions produced Theorems 1 and 2 of the current § 3 and significantly improved the organization of this paper.

REFERENCES

[1] J. ACZÉL, *Lectures on Functional Equations and their Applications*, Academic Press, New York, 1966.

[2] ———, *Determination of all additive quasiarithmetic mean codeword lengths*, Z. Wahrscheinlichkeitstheorie und Verw. Gebiete, 29 (1974), pp. 351–360.

[3] R. BRENT AND H. T. KUNG, *Fast algorithms for manipulating formal power series*, J. Assoc. Comput. Mach., 25 (1978), pp. 581–595.

[4] L. L. CAMPBELL, *A coding theorem and Rényi's entropy*, Information and Control, 8 (1965), pp. 423–429.

[5] ———, *Definition of entropy by means of a coding problem*, Z. Wahrscheinlichkeitstheorie und Verw. Gebiete, 6 (1966), pp. 113–118.

[6] O. CAPRANI, *Roundoff errors in floating-point summation*, Nordisk Tidskr. Informationsbehandling (BIT), 15 (1975), pp. 5–9.

[7] S. EVEN, *Algorithmic Combinatorics*, Macmillan, New York, 1973, Ch. 7.

[8] W. D. FRAZER AND B. T. BENNETT, *Bounds on optimal merge performance, and a strategy for optimality*, J. Assoc. Comput. Mach., 19 (1972), pp. 641–648.

[9] L. FUCHS, *A new proof of an inequality of Hardy-Littlewood-Polya*, Mat. Tidsskr., B(1947), pp. 53–54.

[10] R. G. GALLAGER, *Information Theory and Reliable Communication*, Wiley, New York, 1968.

[11] M. R. GAREY, *Optimal Binary Search Trees of Restricted Maximal Depth*, this Journal, 3 (1974), pp. 101–110.

[12] C. R. GLASSEY AND R. M. KARP, *On the optimality of Huffman Trees*, SIAM J. Appl. Math., 31 (1976), pp. 368–372.

[13] M. C. GOLUMBIC, *Combinatorial merging*, IEEE Trans. Computers, TC-25 (1976), pp. 1164–1167.

[14] G. H. HARDY, J. E. LITTLEWOOD AND G. POLYA, *Inequalities*, Cambridge University Press, Cambridge, 1934.

[15] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[16] D. A. HUFFMAN, *A method for the construction of minimum-redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.

[17] A. ITAI, *Optimal alphabetic trees*, this Journal, 5 (1976), pp. 9–18.

[18] D. E. KNUTH, *Fundamental algorithms: The art of computer programming, Vol.* 1, Addison-Wesley, Reading, MA, 1968.

[19] J. VAN LEEUWEN, *On the construction of Huffman trees*, Proc. 3rd International Colloquium on Automata, Languages, and Programming, Edinburgh, July 1976, pp. 382–410.

[20] J. W. S. LIU, *Algorithms for parsing search queries in inverted file document retrieval systems*, ACM Trans. Database Systems, 1 (1976), pp. 299–316.

[21] D. S. MITRINOVIC, *Analytic Inequalities*, Springer-Verlag, New York, 1970.

[22] P. NATH, *On a coding theorem connected with Rényi's entropy*, Information and Control, 29 (1975), pp. 234–242.

[23] A. OSTROWSKI, *Sur quelques applications des fonctions convexes et concaves au sens de I. Schur (offert en homage à P. Montel)*, J. Math. Pures Appl., 31 (1952), pp. 253–292. (In French.)

[24] D. S. PARKER, *Combinatorial merging and Huffman's algorithm*, IEEE Trans. Computers, TC-28 (1979), pp. 365–367.

[25] R. RUBIN, *Experiments in text-file compression*, Comm. ACM, 19 (1976), pp. 617–623.

[26] A. H. SAMEH, Unpublished manuscript.

[27] I. SCHUR, *Über eine Klasse von Mittelbildungen mit Anwendungen auf die Determinantentheorie*, Sitzungsber. Berl. Math. Ges., 22 (1923), pp. 9–20. (In German.)

[28] J. K. TAMAKI, *Optimal binary trees and sequences realized by Eulerian triangulations*, Ph.D. thesis, Dept. of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, October 1977.

[29] K. Ushijima, *Steps to an efficient program for floating-point summation*, Software-Practice and Experience, 7 (1977), pp. 759–769.

[30] C. K. WONG AND P. C. YUE, *A majorization theorem for the number of distinct outcomes in N independent trials*, Discrete Math., 6 (1973), pp. 391–398.

[31] S. ZIMMERMAN, *An optimal search procedure*, Amer. Math. Monthly, 66 (1959), pp. 690–693.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0003 $01.00/0

STORAGE MODIFICATION MACHINES*

A. SCHÖNHAGE†

Abstract. In 1970 the author introduced a new machine model [A. Schönhage, *Universelle Turing Speicherung*, Dörr, Hotz, eds., Automatentheorie und Formale Sprachen, Bibliogr. Institut, Mannheim, 1970, pp. 369–383] now called storage modification machine (SMM). This paper gives a comprehensive presentation of our present knowledge of SMMs. It contains a complete description of the SMM model and its real time equivalence to the so-called successor RAMs. The preliminary version [A. Schönhage, *Real-time simulation of multidimensional Turing machines by storage modification machines*, Technical Memorandum 37, M.I.T. Project MAC, Cambridge, MA, 1973] of our proof for the real time simulation of multidimensional Turing machines is now worked out in its details. Moreover we show the existence of an SMM that performs integer-multiplication in linear time. The final discussion contains a brief comment on the relationship between SMMs and Kolmogorov algorithms [A. N. Kolmogorov and V. A. Uspenskii, *On the definition of an algorithm*, Uspehi Mat. Nauk, 13 (1958), pp. 3–28; AMS Transl. 2nd ser. vol. 29 (1963), pp. 217–245].

Key words. Turing machines, random access machines, real time simulation, integer-multiplication

Introduction. What is the "true" concept of computability?—In the qualitative sense this question is answered by the general consensus about Church's thesis that effective calculability should be defined by recursiveness or, equivalently, by λ-definability (cf. [2, p. 346]). For the quantitative analysis of algorithms the Turing machine concept is especially suited to provide intuitively appealing measures for the time and space requirements of computations, and it seems in fact to lead to a satisfactorily universal notion of complexity, as long as we adopt the rather rough measure of polynomial reducibilities, since it is well known that complexity classes like P or NP remain unaffected within a wide range of variations of the underlying machine model.

With regard to lower order complexity, however, the situation is quite different. Many of the concrete algorithms given in the literature are (at least implicitly) designed for multitape Turing machines, sometimes the higher flexibility of random access machines (with a variety of instruction sets) is required, and frequently it is totally left to the reader's imagination what the model of computation should look like. We may say that so far no unified and generally accepted measure for the time complexity of algorithmic problems has been established.

In 1970 the present author introduced a new machine model (cf. [11]) now called *storage modification machine* (SMM) and posed the intuitive thesis that this model possesses extreme flexibility and should therefore serve as a basis for an adequate notion of time complexity. In [12] a sketchy proof was supplemented that in fact real time simulation of multidimensional Turing machines by SMMs is possible; Schnorr's observation that SMMs are real time equivalent to so-called successor RAMs (see [10, Chap. 8]) added further support to our thesis. As was pointed out to us by several colleagues Kolmogorov and Uspenskii have introduced a machine model very similar to the SMM model much earlier (in 1958, cf. [7]) already. The main object of their paper was to reach extreme flexibility in a descriptive sense. The clumsy presentation of Kolmogorov algorithms makes it very elaborate, however, to handle them in a detailed way. Therefore we will restrict ourselves to a brief comment on their relationship to SMMs in the final discussion. At the moment we should merely state that Kolmogorov

* Received by the editors February 15, 1979, and in revised form June 24, 1979. This research was supported in part by the National Science Foundation under Grant GJ-34671.

† Mathematisches Institut, Universität Tübingen, Tübingen, Germany.

algorithms are different from SMMs and that they are certainly not stronger but perhaps weaker than SMMs.

After that preliminary phase of consolidation the aim of this paper is now to give a comprehensive presentation of our present knowledge about storage modification machines. For reference purposes it seems to be overdue to have a complete description which replaces the fragmentary information scattered in the literature. So §§ 1–4 will cover the notion of Δ-structures which serve as storage devices, the instruction set of SMMs, the related successor RAMs and the corresponding real time equivalences. Section 5 contains our proof for the real time simulation of multidimensional Turing machines worked out in its details. In § 6 we present our most recent result that there exists an SMM which can perform integer-multiplication in linear time. Some further comments will follow in the final discussion.

Before going into details we should make clear our general attitude towards the problems under consideration. No attention is paid to the question whether a physical realization of our constructs is possible. Such investigations would require a totally different approach. It is the atomistic nature of logical steps which will guide our analysis. Thus, for instance, a Δ-structure can be regarded as a collection of a bounded number of finite functions, which can be modified step by step for single arguments. Our notion of "real" time does not relate to seconds or years. Since it is arbitrary to some extent what we call a single step of a machine, we will admit bounded slow-down. For comparing rather heterogeneous machine models our notion of *simulation* will refer to their input/output behavior only.

DEFINITION A. A machine M' is said to *simulate* another machine M *in real time*, denoted by $M \overset{r}{\rightarrow} M'$, if there exists a constant c such that for every input sequence x the following holds: if x causes M to read an input symbol, or to print an output symbol, or to halt at time steps $0 = t_0 < t_1 < \cdots < t_l$, respectively, then x will cause M' to act in the very same way with regard to those external instructions at time steps $0 = t'_0 < t'_1 < \cdots < t'_l$, where $t'_j - t'_{j-1} \leqq c(t_j - t_{j-1})$ for $1 \leqq j \leqq l$.

DEFINITION B. For machine classes $\mathcal{M}, \mathcal{M}'$ *real time reducibility* $\mathcal{M} \overset{r}{\rightarrow} \mathcal{M}'$ is defined by the condition that for each $M \in \mathcal{M}$ there exists an $M' \in \mathcal{M}'$ such that $M \overset{r}{\rightarrow} M'$. *Real time equivalence* $\mathcal{M} \overset{r}{\leftrightarrow} \mathcal{M}'$ means $\mathcal{M} \overset{r}{\rightarrow} \mathcal{M}'$ and $\mathcal{M}' \overset{r}{\rightarrow} \mathcal{M}$.

Using these terms we can restate our main thesis in the following way: $\mathcal{M} \overset{r}{\rightarrow}$ SMM holds for all *atomistic* machine models $\mathcal{M}$. Of course we are not able to give an ultimate definition of this intuitive notion. The idea is to exclude all kinds of machines which are equipped with powerful compound instructions in any unfair manner.

1. Δ-structures. Conceptually the storage device of a Turing machine consists of *squares* statically arranged in a regular pattern, thus forming a tape, or several tapes, a rectangular grid, or other configurations of higher dimension. Each square can store one symbol of the *working alphabet.* Associated with the storage pattern there is another alphabet containing the *shift instructions* for the work head(s) of the machine, e.g. {R, L} for "right" and "left" on a tape or {N, W, S, E} for the directions "north" etc. on a plane.

Similarly, the data structures to be processed by a storage modification machine are based upon a finite alphabet Δ of directions, i.e. Δ is an individual feature of that particular machine. Such a Δ-*structure* is defined as a finite digraph, where the arcs are labeled with the elements of Δ in such a way that, in 1-1 correspondence to Δ, exactly $\#\Delta$ many *pointers* originate from each node (we always assume $\#\Delta \geqq 2$). In addition, there is one distinguished node, called the *center* of the structure. It allows immediate access from the outer world in analogy to the head on a tape of a Turing machine. In contrast to the storage squares of a Turing machine, however, here the nodes do not bear any extra information; it is the particular pattern of the pointers in the Δ-structure which comprises the stored information.

Formally, a Δ-structure is defined as a triple $S=(X, a, p)$, where X denotes the finite set of nodes, $a \in X$ is the center of S, and $p=(p_\alpha | \alpha \in \Delta)$ is the family of pointer mappings $p_\alpha : X \to X$; $p_\alpha(x)=y$ means that the pointer with label α originating from x goes to y. The center can also be considered as the endpoint of an additional pointer coming from outside.

In a natural way every word $W \in \Delta^*$ determines a path in the structure S starting at the center and following the labeled arcs according to the sequence of symbols in W. This induces a map p^*: $\Delta^* \to X$ recursively defined by

$$(1.1) \qquad \begin{aligned} &p^*(\Box)=a, \qquad \text{where } \Box \text{ denotes the empty word,} \\ &p^*(W\alpha)=p_\alpha(p^*(W)) \quad \text{for all } \alpha \in \Delta,\ W \in \Delta^*. \end{aligned}$$

It can happen that the image $p^*(\Delta^*)$ is a proper subset of X; but in this case, by definition of the operating mode of SMMs, the rest will remain inaccessible forever. Therefore we will admit only structures $S=(X, a, p)$ with full accessibility $p^*(\Delta^*)=X$.

Furthermore the map p^* induces an equivalence relation in Δ^* defined by

$$(1.2) \qquad U \sim V \text{ iff } p^*(U)=p^*(V),$$

with the obvious property

$$(1.3) \qquad U \sim V \quad \Rightarrow \quad UW \sim VW \quad \text{for all } U, V, W \in \Delta^*.$$

Conversely, any finite equivalence relation in Δ^* that fulfils condition (1.3) can be understood as a Δ-structure: the equivalence classes are the nodes, the class containing $\Box$ is the center, and up to isomorphisms all Δ-structures are representable in this way.

Let $\xi(d, n)$ denote the number of different Δ-structures with n nodes, where $d=\#\Delta$. We do not know how to express these numbers in a simple closed form, but a fairly simple combinatorial analysis shows how to compute them from certain recurrences ($\xi(2, n)$, for instance, takes the values 1, 12, 216, 5248, 161175, $\cdots$). In order to estimate how much *information* can be stored in n nodes by using an alphabet with d elements we use the bounds

$$(1.4) \qquad n^{nd-n+1} \leqq \xi(d, n) \leqq \frac{1}{dn+1}\binom{dn+1}{n} n^{nd-n+1}.$$

They are obtained by the observation that $n-1$ of the dn many pointers must form a labeled spanning tree for the n nodes while each of the $dn-n+1$ remaining pointers can be directed arbitrarily to one of the n nodes. Taking logarithms to the base 2 in (1.4) now yields the desired measure of information:

$$(1.5) \qquad \log \xi(d,n)=(d-1)n \log n+O(n \log d).$$

2. The SMM model. The storage of a storage modification machine (SMM) with internal alphabet Δ is at any time represented by one Δ-structure accessible via its center. The finite control of the SMM is given as a *program* written in a formal language like ALGOL, on a very basic level, however. Syntactically such a program is a sequence of *labels* and *instructions*. Labels are written as in ALGOL. Their names can be used in goto statements and similar instructions which transfer control to other places in the program. If a name λ is used in this way it has to occur exactly once followed by a colon, whereas an instruction always ends with a semicolon.

Each instruction begins with one of the following *codes*:

(2.0) ***input, output, goto, halt, new, set, if,***

in most cases followed by additional specifications. The machine starts its work with the first instruction of the program and an empty storage, i.e. the initial Δ-structure has only one node. The effect of the program has to be defined recursively by explaining the effect of the different instructions on the instantaneous Δ-structure, or on the flow of control, respectively. At first we explain the *common* instructions, which will be same for all machine models considered in this paper.

(2.1) ***input*** λ_0, λ_1;
this branching instruction causes the next input symbol $\beta \in \{0, 1\}$ to be read and transfers control to label λ_β; if the input string is exhausted, the next instruction is executed.

(2.2) ***output*** β;
$\beta \in \{0, 1\}$ is sent to the output.

(2.3) ***goto*** λ;
control is transferred to label λ.

(2.4) ***halt***;
the machine stops working; implicitly this also happens when the control passes the end of the program.

In addition there are the following *internal* instructions:

(2.5) ***new*** W;
The instantaneous storage structure $S = (X, a, p)$ is modified by creating a new node y; the resulting structure is $\bar{S} = (\bar{X}, \bar{a}, \bar{p})$, where $\bar{X} = X \cup \{y\}$. $W \in \Delta^*$ determines where the new node shall be located. If $W = \square$, then $\bar{a} = y$, and $\bar{p}_\delta(y) = a$ for all $\delta \in \Delta$. If $W = U\alpha$ with $U \in \Delta^*$, $\alpha \in \Delta$, then $\bar{a} = a$, $\bar{p}_\alpha(p^*(U)) = y$, and $\bar{p}_\delta(y) = p^*(W)$ for all $\delta \in \Delta$. All other pointers remain unchanged.

(2.6) ***set*** W ***to*** V;
here $S = (X, a, p)$ is modified into $\bar{S} = (\bar{X}, \bar{a}, \bar{p})^*$ by new assignment of a pointer. W and $V \in \Delta^*$ determine which pointer shall be directed where, and it is $\bar{X} = X$. The asterisk indicates that $\bar{S}$ is obtained by reduction to that part of $\bar{X}$ (and $\bar{p}$) which is accessible from $\bar{a}$. If $W = \square$, then $\bar{a} = p^*(V)$, $\bar{p} = p$. If $W = U\alpha$, then $\bar{a} = a$, $\bar{p}_\alpha(p^*(U)) = p^*(V)$, and again all other pointers remain unchanged.

(2.7) ***if*** $U = V$ ***then*** σ;
Here σ denotes an instruction of one of the previous types which is executed iff in the instantaneous storage structure $p^*(U) = p^*(V)$ is fulfilled (otherwise σ is skipped).

(2.8) ***if*** $U \neq V$ ***then*** σ;
here σ is executed iff $p^*(U) \neq p^*(V)$.

Obviously this list of instructions is redundant to some extent. The test $U \neq V$, for instance, could easily be accomplished by means of the test $U = V$; similarly it would be sufficient to have the instruction ***new*** with $W=\square$ only, but there is no point in doing without such conveniences.

With respect to the descriptions of the codes ***new*** and ***set*** it is quite tempting to expect that after execution of ***new*** W (or ***set*** W ***to*** V) the relations $\bar{p}^*(W)=y$ (or $\bar{p}^*(W)=p^*(V)$, respectively) will hold, but this is not true in general. A counterexample is provided by the program

start: ***new*** AA; ***set*** BB ***to*** A; ***halt***;

it produces the $\{A, B\}$-structure $S=(\{a, y\}, a, p)$ with $p_A(a)=p_B(a)=y$, $p_A(y)=p_B(y)=a$; this implies $p^*(AA)=p^*(BB)=a\neq y=p^*(A)$.

Dependent on the input string we define the *running time* of an SMM program as the number of instructions performed including ***halt***. The conditional instructions ***if*** $\cdots$ ***then*** $\cdots$ are always counted regardless of the outcome of the test. A more precise and realistic measure could be obtained by weighing instructions like ***set*** W ***to*** V proportional to one plus the length of W plus the length of V, but for each particular program this would increase the time by at most a constant factor and therefore lead to an equivalent concept.

3. Related RAM models. Another concept which allows us to get rid of the restrictions imposed by the inflexibility of Turing storage devices is that of a *random access machine* (RAM). We assume the reader to be familiar with at least one of the RAM models given in the literature (see e.g. [1, pp. 5–11]). Since, however, the computational power of a RAM model seems to depend rather sensitively on the scope of its instruction set, we nevertheless will have to go into detail.

One important principle will be to admit only such instructions which can be said to be of an *atomistic* nature. We will describe two versions of the so-called *successor* RAM, with the successor function as the only arithmetic operation. Our RAM1 model follows the common pattern of all RAM concepts, whereas the RAM0 version deserves special attention for its extreme simplicity; its instruction set consists of only a few one letter codes, without any (explicit) addressing.

Both models have countably many storage locations with addresses $0, 1, 2, \cdots$ (and one or two extra registers) each of which can store an arbitrary natural number. This seems to be an unrealistic assumption, as long as we think of integers in binary representation. We prefer, however, to consider such a storage in terms of pointers. Then each location n is a node with two pointers, one leading to the successor $n+1$, the other pointing to an arbitrary node in this infinite chain, which is denoted by $\langle n\rangle$ usually called the *contents* of location n. With this perception in mind we must mistrust our every day intuition, as may be demonstrated by an example: Whether $\langle n\rangle$ is even or odd can no longer be decided in one step but requires some extra programming.

Programs for our RAM models can be written in the same manner as for an SMM. We adopt the common instructions for input, output, and control, as described in (2.1)–(2.4). Further explanations have to be given for the internal instructions of our two models. In order to avoid double addressing the RAM1 has an extra register (without an address) which serves as an *accumulator*. Its current contents is denoted by z. Initially $z=0$, and $\langle n\rangle=0$ for all n. Each of the internal instructions begins with an operation code which is followed by an explicit address n written in decimal notation. A complete listing of the internal instructions together with their meaning is given in Table 3.1.

In order to avoid any explicit addressing the RAM0 has the accumulator with contents z and an additional address register with current contents n (initially 0). The internal instructions are expressed by one letter codes, given in Table 3.2.

TABLE 3.1
*Internal RAM*1 *instructions.*

Instruction	Effect	Verbal explanation
LDA n	$z := n$	*l*oa*d* *a*ddress
LDD n	$z := \langle n\rangle$	*l*oa*d* *d*irectly
LDI n	$z := \langle\langle n\rangle\rangle$	*l*oa*d* *i*ndirectly
STD n	$\langle n\rangle := z$	*st*ore *d*irectly
STI n	$\langle\langle n\rangle\rangle := z$	*st*ore *i*ndirectly
COM n	$z = \langle n\rangle?$	*com*pare: the next instruction is executed iff equality holds*
SUC	$z := z+1$	compute successor

* For simplicity it is assumed that COM n must not be followed by a label or by another COM instruction.

TABLE 3.2
*Internal RAM*0 *instructions.*

Code	Effect	Verbal explanation
Z	$z := 0$	set to *z*ero
A	$z := z+1$	*a*dd one
N	$n := z$	set address *n*
L	$z := \langle z\rangle$	*l*oad
S	$\langle n\rangle := z$	*s*tore
C	$z = 0?$	*c*ompare: the next instruction is executed iff equality holds*

* C must not be followed by a label or by C.

4. Real time equivalences. Now we are ready to present the first main result.

THEOREM 4.1. *The machine models SMM, RAM*1, *RAM*0 *are real time equivalent.*

Proof. The proof is by showing the real time reducibilities

$$\text{RAM1} \overset{r}{\to} \text{RAM0} \overset{r}{\to} \text{SMM} \overset{r}{\to} \text{RAM1}, \tag{4.1}$$

according to Definition B.

We begin with the perhaps most surprising fact that the rudimentary RAM0 model offers (at least) the same computational power (up to a constant factor) as the RAM1 model. For this let M^* be a given RAM1. We have to exhibit a RAM0 M that simulates M^* in real time. In order to be distinct let $\langle k\rangle^*$ denote the contents of the kth memory location of M^*, z^* the contents of its accumulator, whereas without asterisk the corresponding data of M are meant.

Since both models have the same common instructions (2.1)–(2.4), it is sufficient to show how to replace each internal RAM1 instruction by a finite sequence of RAM0 instructions in such a way that the outcome of the implicit branching by comparisons COM n is simulated correctly. This is achieved by a suitable mapping of the stored information of M^* into the storage of M. During the step by step simulation the relations

$$2\cdot z^* = \langle 1\rangle, \tag{4.2}$$

$$2\cdot \langle n\rangle^* = \langle 2n\rangle \quad \text{for all } n \in \mathbb{N} \tag{4.3}$$

will always be preserved, i.e. all M^*-addresses are doubled and stored in storage locations of M with doubled addresses; location 1 is used to simulate the accumulator of M^*, and the locations 3, 5, $\cdots$ will serve to simulate comparisons. The technical details of the simulation are given in Table 4.1. The instruction σ^* after COM n has to be skipped in case of $\neq$; correspondingly its simulation σ will be skipped iff the C-instruction finds $z=0$. This explains the technique employed for the simulation of COM n. For the real time reduction RAM0 $\overset{\mathrm{r}}{\to}$ SMM we similarly show how to represent an instantaneous storage configuration of a RAM0 by a suitable Δ-structure, where we choose $\Delta=\{A, B\}$, and then how to simulate the internal RAM0 instructions (cf. Table 3.2). The Δ-structure consists of nodes $0, 1, \cdots, m$, which represent a finite section of the RAM0 storage to be extended when required, and two extra nodes c and a, where c is the center. The pointer mappings are given by

TABLE 4.1
Simulation of internal RAM1 instructions.

M^*-instruction	Simulation	Effect of M
LDA n	$ZANZA^{2n}S$	$\langle 1\rangle := 2n$
LDD n	$ZANZA^{2n}LS$	$\langle 1\rangle := \langle 2n\rangle$
LDI n	$ZANZA^{2n}LLS$	$\langle 1\rangle := \langle\langle 2n\rangle\rangle$
STD n	$ZA^{2n}NZALS$	$\langle 2n\rangle := \langle 1\rangle$
STI n	$ZA^{2n}LNZALS$	$\langle\langle 2n\rangle\rangle := \langle 1\rangle$
COM n; σ^*	$ZALA^3NZS$	$\langle\langle 1\rangle+3\rangle := 0$
	$ZA^{2n}LA^3NS$	$\langle\langle 2n\rangle+3\rangle := x \quad (x \geqq 3)$
	$ZALA^3L$	$z := \langle\langle 1\rangle+3\rangle$
	C; *goto* λ; σ; λ:	
SUC	$ZANLAAS$	$\langle 1\rangle := \langle 1\rangle+2$

$$
\begin{aligned}
&p_A(c)=0, \quad p_B(c)=a, \quad p_A(a)=z, \quad p_B(a)=n,\\
(4.4)\qquad &p_A(k)=k+1 \quad \text{for } 0\leqq k<m, \quad p_A(m)=0,\\
&p_B(k)=\langle k\rangle \quad \text{for } 0\leqq k\leqq m.
\end{aligned}
$$

Table 4.2 shows the simulation of the internal RAM0 instructions by updating the Δ-structure in order to preserve the conditions (4.4). Initially the SMM performs the steps ***new***; ***new*** *B*; the first one creates c and puts $\langle 0\rangle := 0$, the second one creates a with $z := n := 0$. Extension of the Δ-structure is required when the RAM0 instruction A has to be simulated and $z=m$ holds.

TABLE 4.2
Simulation of internal RAM0 instructions.

RAM0 code	SMM simulation
Z	***set*** *BA* ***to*** *A*;
A	***if*** *BAA* = *A* ***then new*** *BAA*; ***set*** *BA* ***to*** *BAA*;
N	***set*** *BB* ***to*** *BA*;
L	***set*** *BA* ***to*** *BAB*;
S	***set*** *BBB* ***to*** *BA*;
C	***if*** *BA* = *A* ***then*** $\cdots$

Finally we give a sketchy outline for the reduction SMM $\overset{\mathrm{r}}{\to}$ RAM1. Without restriction we can assume that the alphabet of the given SMM is $\Delta=\{0, 1, \cdots, d-1\}$ with some $d \geqq 2$. Each node of the instantaneous Δ-structure is represented by means of

d consecutive storage locations of the simulating RAM1. More precisely the node with serial number $k \geqq 0$ is put into locations $kd+4$, $kd+5, \cdots, (k+1)d+3$, and $kd+4$ is its address. For any $\delta \in \Delta$ the destination of the δ-pointer from node k is stored in location $kd+4+\delta$. The address of the center of S is always stored in 0, and location 1 contains the current address of the free storage domain to be increased by d with each ***new*** instruction. Location 2 and 3 are for intermediate use to build up the addresses of the nodes $p^*(U)$, $p^*(V)$, when $U, V \in \Delta^*$ occur in SMM instructions of type (2.5)–(2.8).

The RAM1 begins the simulation by setting up its initial configuration

$$\langle 0\rangle = 4, \langle 1\rangle = 4+d, \langle 4+\delta\rangle = 4 \quad \text{for } 0 \leqq \delta < d.$$

Then all SMM instructions of type (2.1)–(2.4) can be replaced by the same RAM1 instructions, and it should be obvious now how to simulate each ***new***, ***set***, or ***if*** instruction of the given SMM program by a finite sequence of RAM1 instructions. In this way real time simulatibn is achieved.

Since the loop of reductions (4.1) can also be started with an arbitrary SMM, we obtain its real time reducibility to an SMM with alphabet $\{A, B\}$. This observation completes the proof of Theorem 4.1 by justifying that we need not specify the size of the alphabet when talking about the SMM model(s).

5. Real time simulation of Turing machines. It is a fairly simple task to design a RAM1 real time simulation for multitape Turing machines. Things are quite different, however, if TMs with storage devices of higher dimension are admitted. The main difficulty is encountered with a two-dimensional storage plane already; its typical nature is well illustrated by the *self-crossing problem*:

Over the alphabet of directions N(orth), W, S, E every infinite sequence of symbols describes a path in the plane. How can a machine, when scanning symbol after symbol, decide in real time whether the present position has already been visited before? There is an obvious solution, if we use a TM with a two-dimensional storage which is empty at the outset of the computation, but what about the other machine models? How can we program an everyday computer to solve this problem efficiently for long input sequences?—From our main result, TM $\overset{r}{\rightarrow}$ SMM, as given below we conclude that the self-crossing problem has also an SMM real time solution. In the same way our simulation technique can facilitate the construction of efficient algorithms for other problems that are adaptable to higher dimensional storage configurations.

At first we have to specify the class of TMs to be simulated. The storage of such a machine may consist of one or several *components* of finite dimension, each of which is isomorphic to $\mathbb{Z}^k$ with some individual $k \geqq 1$ and can be accessed by one or several work heads. Moves of each head position are restricted to an increase or decrease of one of its integer coordinates by one. Initially all the storage locations are blank, and all heads are positioned at the origin of their own storage component (some standardizing of this kind seems to be necessary in order to avoid "preinformation" that otherwise could be hidden in positional relations of several heads on one component). Without restriction we can further assume that for input, output, and internal work the alphabet $\{0, 1\}$ is used.

Again the finite control is given as a finite sequence of instructions and labels. In addition to ***input***, ***output***, ***goto***, ***halt*** (see (2.1)–(2.4)) there are the following internal instructions:

head ν; here $\nu \geqq 1$ denotes an integer (written in decimal notation); the effect of this instruction is that subsequent instructions refer to the work head with number ν (the *active* head), until the next ***head*** instruction is performed. At the beginning the head with number 1 is active.

write α; $\alpha \in \{0, 1, b\}$, where b denotes "blank"; the inscription of the square under the active head is changed into α.

read λ_0, λ_1; if the square under the active head contains 0 or 1, then control is transferred to label λ_0 or λ_1, respectively. In case of a blank the next instruction is performed.

move δ; the active head position is shifted in direction δ, where δ has to belong to the set of directions associated with the storage component of the active head.

Now we are ready to state the main result of this paragraph.

THEOREM 5.1. *Every Turing machine as specified above can be simulated by a suitable SMM in real time.*

Proof. We begin our proof with the observation that a TM with r heads on several storage components of dimensions $k_1, k_2, \cdots, k_l$ respectively can apparently be simulated in real time by a suitable TM with r heads on just one storage component isomorphic to $\mathbb{Z}^d$, where $d = 1 + \max\{k_1, \cdots, k_l\}$. Furthermore we will restrict the presentation of our simulation technique to the case $d = 2$, in spite of the fact that the first reduction usually leads to some $d > 2$; in the end it will become obvious to the reader how the method can be adapted to other values of d.

In addition to the four directions of the plane we need pointers for ***up*** and ***down***. Therefore the SMM for simulating a given TM uses the alphabet $\Delta = \{\mathrm{N, W, S, E, U, D}\}$. During the simulation the current configuration of the TM and further data needed to achieve the real time simulation are encoded in a rather complicated Δ-structure, which is composed of the following parts:

(a) The contents of the storage plane together with the geometric information which areas of the plane have already been visited by one of the work heads are encoded in the *pyramidal structure P.*

(b) For each of the work heads (with numbers $j = 1, 2, \cdots, r$) there is a substructure C_j, called a *counter*, from which, corresponding to the current head position, the pyramidal structure is accessed properly.

(c) The counters $C_1, \cdots, C_r$ are accessed from the *headquarter* H; it contains the center A of the whole structure and serves to select the counter of the active head according to the state of the TM each time when an internal TM instruction is simulated.

In order to describe the simulation process we now have to explain these parts in detail.

5.1. The pyramidal structure. At any stage of the simulation the pyramidal structure P is a finite portion of the following infinite structure $\mathcal{P}$ with nodes $M_k(x, y)$ for all $k \in \mathbb{N}$, $(x, y) \in \mathbb{Z}^2$. We consider k as the *vertical* coordinate. Each *level* $\mathcal{L}_k = \{M_k(x, y) | (x, y) \in \mathbb{Z}^2\}$ is structured as a rectangular grid by the *horizontal* pointers labeled with N, W, S, E leading from each $M_k(x, y)$ to $M_k(x, y+1)$, $M_k(x-1, y)$, $M_k(x, y-1)$, $M_k(x+1, y)$ respectively. The subset $\{M_k(\xi, \eta) \,|\, |\xi - x| \leqq 1,\ |\eta - y| \leqq 1\}$ together with the 24 horizontal pointers connecting any two of these nodes is called the *vicinity* of $M_k(x, y)$ (see Fig. 5.1; in this definition nothing is assumed about the other pointers).

The ground level $\mathcal{L}_0$ is intended to represent the storage plane of the TM, but we have to keep in mind that this infinite set of nodes does not preexist. It has to be constructed in finite parts during the simulation. Thus the pyramidal structure P

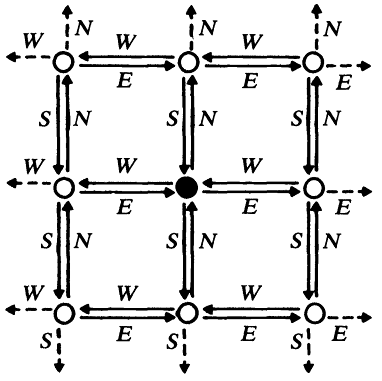


FIG. 5.1. *The vicinity of node ●.*

contains some finite subset L_0 of $\mathscr{L}_0$, which, roughly speaking, represents just those areas of the TM storage which have been visited by one of the work heads so far. The main problem arises when L_0 has to be extended: how to decide in case of a pointer not yet properly installed whether the node to which it should point has already been constructed.

In order to overcome this difficulty we use the higher levels $\mathscr{L}_1, \mathscr{L}_2, \cdots$ to represent the storage plane on successively reduced scales in linear ratio $1:3:9:\cdots$. $M_{k+1}(x, y)$ represents the vicinity of $M_k(3x, 3y)$; correspondingly connections between $\mathscr{L}_k$ and $\mathscr{L}_{k+1}$ are provided by D-pointers (downward) from $M_{k+1}(x, y)$ to $M_k(3x, 3y)$ and by U-pointers (upward) from $M_k(x, y)$ to $M_{k+1}(\lfloor(x+1)/3\rfloor, \lfloor(y+1)/3\rfloor)$, so for all k, x, y. On level $\mathscr{L}_0$ the D-pointers will be used to encode the information stored in the TM storage.

Now we are ready to state some basic facts about the pyramidal structure P; further explanations will be given in the next subsection. There are finite subsets $L_k \subseteq \mathscr{L}_k$, $0 \leqq k \leqq m$, with a unique m such that

$$P = L_0 \cup L_1 \cup \cdots \cup L_m, \tag{5.1}$$

and L_m contains exactly one node, namely $M_m(0, 0)$, called the *top node* of P. Our simulation will step by step preserve the following conditions for P.

(i) Any pointer originating from a node $K \in P$ is directed either to the same node as in the infinite structure $\mathscr{P}$, or to the center A in the headquarter. In the former case the pointer is said to be *properly installed* and then its destination must also belong to P (except for the D-pointers from $K \in L_0$). Otherwise the destination A indicates that the pointer is not yet properly installed (this may happen even when its true. destination already belongs to P).

(ii) For each $K \in P$ different from the top node the U-pointer is properly installed, and for its destination, denoted by KU, the D-pointer is properly installed, and so is the whole vicinity of KUD.

According to the latter condition P can be viewed as an agglomeration of elementary pyramids each of which consists of ten nodes with all their internal connections.

5.2. The counters. During the simulation for each work head of the given TM we have to keep track of its position on the storage plane. Its coordinates being (x, y), this can be achieved by providing a pointer to the node $H_0 = M_0(x, y)$ on the ground level of the pyramidal structure. In order to meet the requirements of real time simulation, however, we will sometimes need immediate access also to the higher level representations of H_0, namely (following the U-pointers) to H_0U, H_0UU, $\cdots$ which, of course, must belong to P already. For this purpose we provide additional pointers to certain nodes $H_1, H_2, \cdots, H_l$ belonging to the levels $L_1, L_2, \cdots, L_l$ of the pyramidal structure, which at any time represent the actual position of this particular head *approximately*:

(iii) If K belongs to the vicinity of H_j, then KU belongs to the vicinity of $H_{j+1} (0 \leqq j < l)$.

In addition we will always have $H_l = M_l(0, 0)$, still beneath the top node of P.

For each head these pointers are updated dynamically according to its movements by means of a special device C, called a *counter*. It controls a sequence of *stages* with numbers $1, 2, 3, \cdots$. Stages 1 and 2 are performed at the outset of the simulation; stages $2r+1$ and $2r+2$ occur, when the rth move of the corresponding head is simulated. More precisely the stages with odd numbers are used for updating H_0, while the stages with even numbers refer to the updating on higher levels. They are scheduled according to the same principle: only every other stage is used for updating H_1; again the remaining half is saved for the updating on higher levels, etc. This results in the following schedule:

At stage t level k is serviced, where (uniquely)

$$t = (2j-1) \cdot 2^k. \tag{5.2}$$

The corresponding function $t \mapsto k(t)$ can be computed recursively in real time by use of

$$k(2t-1) = 0, \qquad k(2t) = k(t)+1 \text{ for } t \geqq 1. \tag{5.3}$$

Therefore we design the counter C such that, after completion of stage $2t$, it embodies the graph of this function for all arguments τ with $1 \leqq \tau \leqq 2t$. It contains the nodes $F_1, F_2, \cdots, F_{2t}$ and $G_0, G_1, \cdots, G_l$, where

$$l = \max \{k(\tau) | \tau \leqq 2t\} = \lfloor \log_2 (2t) \rfloor. \tag{5.4}$$

The connections among these nodes are as shown in Fig. 5.2 (where all the meaningless pointers have been omitted). In particular, the S-pointer from F_τ leads to $G_{k(\tau)}$, and the E-pointer from G_k leads to the approximate head position H_k on level k of the pyramidal structure. At stage $2t$ the counter itself is accessed from its *base* B_ν (belonging to the headquarter) via the N-pointer to F_{2t}, the E-pointer to F_t, and the S-pointer to G_0. While the νth head of the TM is active, the S-pointer from the center A leads to B_ν, thus selecting this particular counter $C = C_\nu$.

5.3. Updating. Now we describe our SMM simulation of the TM instruction ***move*** δ, where $\delta \in \{\mathrm{N, W, S, E}\}$. Let us assume that this is the tth move of the νth work head. As can be seen from Fig. 5.2, the proper shift of H_0 is simply achieved by just one SMM instruction, namely ***set SSE to SSEδ***, provided the δ-pointer from H_0 is properly installed already. That will be true by proper initialization for the first step, and also later on, since in addition to rule (iii) our simulation will preserve the condition

(iv) After each simulation step the pyramidal structure contains the vicinities of all the nodes H_j for all work heads.

The simulation of the ***move*** instruction consists of the stages $2t+1$ and $2t+2$ of counter C_ν including the updating on level 0 and on level $k(2t+2) = k(t+1)+1$. In general, *updating* on level j consists of *preparation* and *adjustment*.

Preparation means making sure that the vicinity of H_j belongs to the pyramidal structure P. In view of condition (i) it can easily be checked whether parts of the vicinity of H_j are missing. If this is the case, the missing elements can be constructed in bounded time such that (i) and (ii) are maintained. The corresponding algorithm is based upon

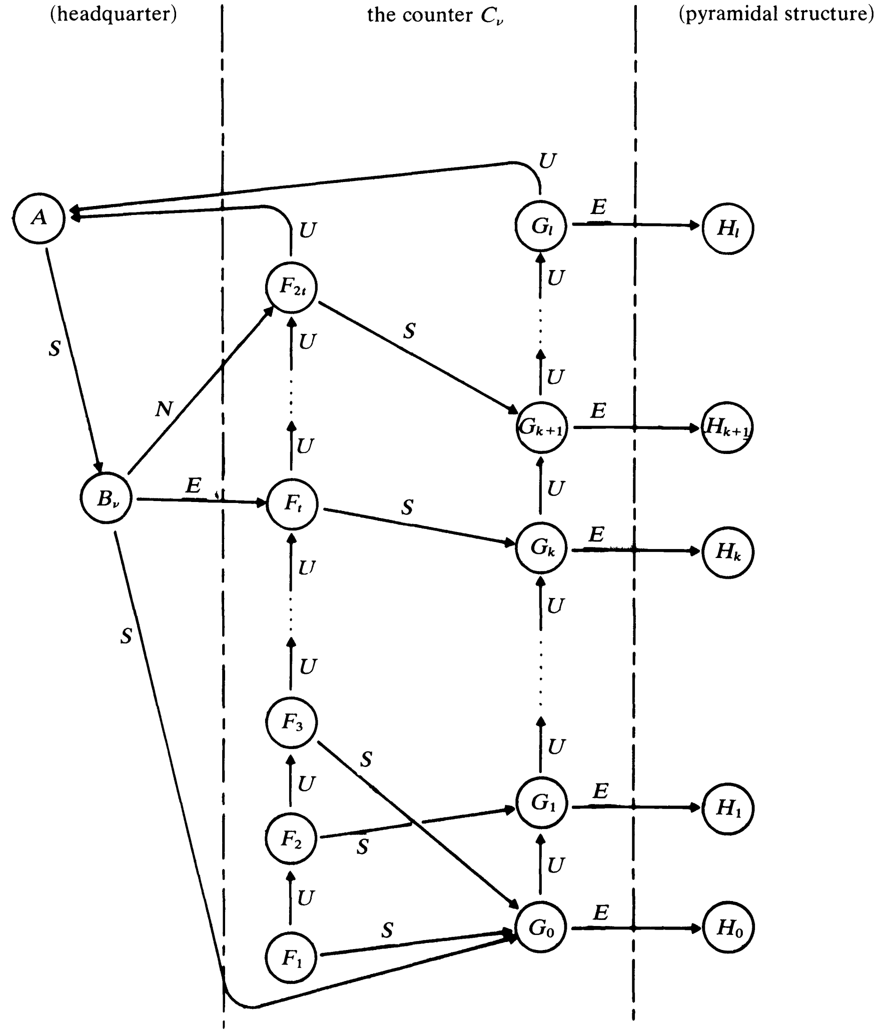


FIG. 5.2. *The counter C_ν after completion of stage 2t and its connections with the headquarter and the pyramidal structure.*

the following argument: If $j = l$, then $H_l = M_l(0, 0)$, and this node has its vicinity in P because of (ii), except for the case that H_l is the top node of P. This is only possible, if l has just been increased by one, and is recognizable by checking whether the U-pointer from H_l leads to A. Then $M_{l+1}(0, 0)$ is created as the new top node of P together with the vicinity of H_l.

Otherwise we are left with the case $j < l$. Here condition (iii) guarantees that all nine nodes under preparation must be located directly beneath the vicinity of H_{j+1} which already exists (cond. (iv)). From there full information is obtainable on how to fill in the missing elements. If, for instance, the W-pointer from H_j is not yet properly installed, then H_j must belong to the western part of the vicinity of H_jUD, i.e. H_j coincides with one of the three nodes H_jUDWN, H_jUDW, H_jUDWS, e.g. $H_j = H_jUDWS$. There are two possibilities: If $H_jUWD \neq A$, then the W-pointer can be immediately directed to its proper destination H_jUWDES which, by reason of (i), (ii), must exist in P already. Otherwise at first the nine nodes at the base of the elementary pyramid under its top node H_jUW have to be constructed with all their internal connections. From this example it should be obvious now that preparation is possible in a bounded number of SMM steps.

Adjustment on level j means to shift H_j appropriately. For $j=0$ this means to replace H_0 by $H_0\delta$, and for $j>0$, to replace H_j by $H_{j-1}U$.

Within each stage adjustment precedes preparation, but this has to be understood in the following way. When level j is serviced for the first time, which happens at step 2^j, then l has just been increased, and $j=l$. In this case adjustment means that $H_j=M_j(0,0)$ is introduced. Otherwise preparation at stage $(2i-1)2^j$ prepares the adjustment at stage $(2i+1)2^j$, which is the next time level j is serviced. In the meantime exactly two adjustments on level $j-1$ occur, namely at the stages $(4i\pm 1)2^{j-1}$. So we obtain by induction on j that adjustment on level j can shift H_j by not more than a king's move in chess, and that even after two such shifts condition (iii) is still satisfied.

With reference to Fig. 5.2 we finally present a sequence of SMM instructions for the simulation of ***move*** δ. The "macro" ***prepare*** followed by the address of a node H_j is a pseudo instruction which has to be replaced by a piece of program which peforms the preparation around H_j as described above.

```
SMM simulation of the TM instruction move $\delta$;
set SSE to SSE$\delta$; prepare SSE; (updating on level 0)
new SNU; set SNUS to SS; ($k(2t+1)=0$)
new SNUU; (creates $F_{2t+2}$)
set SE to SEU; set SN to SNUU;
if SESU = □ then new SESU; (increases $l$)
set SNS to SESU; ($k(2t+2)=k(t+1)+1$)
set SNSE to SESEU; (updating on level $k(2t+2)$)
prepare SNSE;
```

5.4. Conclusion. In order to explain the simulation of the other internal TM instructions we first have to describe the headquarter. In addition to the center A and the base nodes $B_1, B_2, \cdots, B_r$ for the counters it contains two other nodes A_0, A_1 (cf. Fig. 5.3) which are used to encode the stored information: if the square of the storage plane with coordinates x, y contains the inscription $\alpha\in\{0,1,b\}$ and if $M_0(x,y)$ exists in P already then the D-pointer from $M_0(x,y)$ leads to A_α in the headquarter (in case of a blank A_b means A). Consequently in creating new nodes on the ground level of P their D-pointers have to be directed to the center A, since the TM storage is assumed to be empty initially.

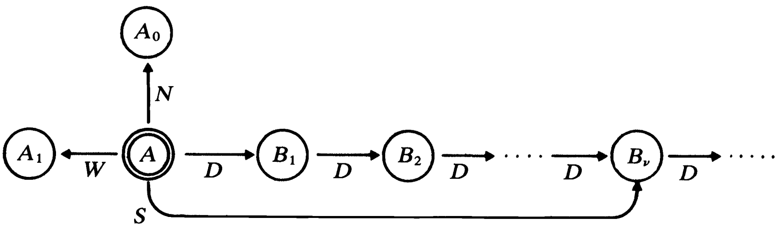


FIG. 5.3. *The headquarter with center A, when the νth head is active.*

Our simulations of the other internal TM instructions are (cf. Fig. 5.2 and Fig. 5.3):

```
set S to $D^\nu$;                 for head $\nu$;
set SSED to $\beta$;              for write $\alpha$;
    (where $\beta = N, W, □$ for $\alpha = 0, 1, b$)
if SSED = N then goto $\lambda_0$;
if SSED = W then goto $\lambda_1$;  for read $\lambda_0, \lambda_1$.
```

The whole simulation process is *initialized* by creating all nodes of the headquarter, the nodes F_1, F_2, G_0, G_1 for each of the counters, and the initial pyramidal structure consisting of the vicinities of $M_0(0, 0)$, $M_1(0, 0)$, and its top node $M_2(0, 0)$, together with all their connections properly installed. For all counters the E-pointers from G_0, G_1 are directed to $H_0 = M_0(0, 0)$, $H_1 = M_1(0, 0)$ respectively.

All this also requires only bounded time. Therefore we have constructed an SMM which simulates the given TM in real time. So far we have restricted our considerations to the two-dimensional case; it should be obvious now how to generalize our method to higher dimensions: With dimension d we will have $2d$ different directions plus the extra pointers U and D, the vicinity of a node will consist of 3^d many nodes, and the pyramidal structure will be embedded in $\mathbb{N} \times \mathbb{Z}^d$.

6. Integer-multiplication in linear time. Still the best upper bound we know for multiplying two N-bit numbers on a multitape Turing machine is $O(N \log N \log \log N)$. The slightly improved bound $O(N \log N)$ can be obtained for certain other machine models (including all kinds of RAMs) with facilities for rapid table look-up by use of a multiplication table (cf. [5, Chap. 4.3.3, p. 275]).

For RAMs with ***addition*** at unit cost the even better bound $O(N \log \log N)$ is possible by simply applying the general speed-up theorem given in [14, p. 63], or [9, p. 104], which allows to save a factor of order $\log t$ when simulating a TM with time bound t. We do not know, however, whether our restricted RAM1 model (cf. § 3) does imply any such speed-up at all. On the other hand the SMMs are sufficiently flexible that we can gain something when a multitape Turing machine has to be simulated simultaneously for a large number of inputs of the same length. Then we can exploit the fact that, within sufficiently small time-intervals, the same performance of the Turing machine will occur in many of the computations running in parallel. A general theorem of this type will be given at the end of our paper without proof. Here we prefer to explicate this idea in detail by establishing a definite result for a prominent problem.

THEOREM 6.1. *There exists an SMM which performs integer-multiplication in linear time, i.e. there is a constant c such that any $2N$-bit input is read as the concatenation of two N-bit integers x, y, and after at most cN many steps their product $z = xy$ is output in binary representation.*

The proof is based upon the application of the FFT over the field of complex numbers.

The computations are carried out in numerical approximation by using fix-point numbers of sufficient length which, however, will not be handled in their binary representation. It is the economical use of other number systems which makes the SMM so fast.

6.1. SMM implementation of digital operations. Binary strings will be split into blocks of equal length $b \geqq 1$. Correspondingly, arithmetic computations will be realized by performing sequences of *basic digital operations* in the B-ary number system, where $B = 2^b$. Our SMM will use the alphabet $\Delta = \{P, Q, S, W\}$. Its Δ-structure will contain a substructure called *B-scale*. It consists of nodes $D_0, D_1, \cdots, D_{B-1}$ cyclically linked by their S-pointers, i.e.

$$D_jS = D_{j+1}, D_{B-1}S = D_0. \tag{6.1}$$

We will always keep to the convention that a B-ary digit p is represented by a pointer to D_p. For doubling and halving, the B-scale contains pointers properly installed according to the identities

$$D_jP = D_{2j}, D_{2j}Q = D_{2j+1}Q = D_j \tag{6.2}$$

for $0 \leqq j < B/2$ (see Fig. 6.1). The W-pointers will later be used to transfer digital information from the B-scale to other places.

There is a simple SMM program which, for a given b, constructs the B-scale in time $O(B)$. Furthermore there are obvious algorithms for translating a B-ary digit into its binary representation (a chain of length b with some convenient bit encoding), and vice versa, both in time $O(b)$, since residues mod 2 are easily obtained by checking whether $D_jQP = D_j$ holds.

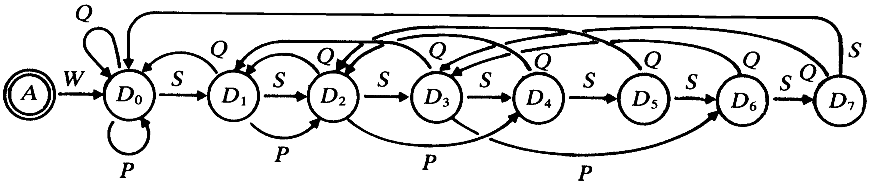


FIG. 6.1. *B-scale for $b = 3$, hooked to the center A via the W-pointer.*

Our SMM performs B-ary arithmetic by means of the following basic digital operations:

$$p + q = r + sB \qquad (0 \leqq s \leqq 1), \tag{6.3}$$

$$p - q = r \cdot (-1)^s \qquad (0 \leqq s \leqq 1, s = 0 \text{ for } p = q), \tag{6.4}$$

$$p * q = r + sB. \tag{6.5}$$

The transition from a given pair (p, q) of digits to the digits r, s which constitute the result is achieved by suitable subroutines in time $O(b)$ for addition/subtraction, and $O(b^2)$ for multiplication, if intermediate translation to the binary system is used, where simple algorithms exist. Comparisons between B-ary digits are possible by subtraction.

For the division of B-ary numbers a further basic operation would be needed, for instance

$$(p, q) \mapsto \left\lfloor \frac{pB}{q+1} \right\rfloor \quad \text{for } 1 \leqq p \leqq q. \tag{6.6}$$

6.2. Mass production. At times our SMM will operate in an interpreting mode. Then it processes programs for numerical computations in B-ary arithmetic which are built up from basic digital operations. They have to be encoded as parts of the Δ-structure in some feasible manner. For each of these programs, which are processed simultaneously, an instruction pointer must be maintained. We may assume that everything is organized such that within one *sweep* at a time each of the programs is promoted by one basic step. Now the decisive point is that these digital operations are not performed immediately. Instead, they are collected in three lines, additions, subtractions, multiplications separately. To be specific, let us for example consider the *collector line* for the multiplications. For the ith multiplication of digits p_i, q_i it contains a node C_i. Its pointers are directed acording to

$$C_iP = D_{p_i}, \quad C_iQ = D_{q_i}, \quad C_iW = R_i, \quad C_iS = C_{i+1}, \tag{6.7}$$

where R_i denotes the destination where to deliver the result of this multiplication, i.e. R_i is some node in one of the programs. In general, this collecting is restricted to a single sweep, since the programs usually must be furnished with the results, before the next sweep can be organized.

LEMMA 6.2. *For m simultaneous programs each sweep can be performed in time* $O(m+b^2B^2)$.

This is our crucial result. The idea is first to *sort* the collector lines and then to execute each of the $3B^2$ different digital operations once at most. To distribute the results will require not more than $O(m)$ steps.

It remains to show that sorting (with respect to equality only, which is sufficient here) is possible in linear time. Therefore let us assume that the collector line to be sorted (cf. (6.7)) is accessed from the center A by its S-pointer, the endpoint C_μ has its S-pointer back to A, and $AW = D_0$ (see Fig. 6.1), while all other parts of the Δ-structure are reached via AQ. Then sorting with respect to equal operands q_i is achieved by the following SMM program in linear time.

```
        new P; set PS to □; set WW to W;
empty:  set WW to WWS; set WWW to P;
        if WW $\neq$ P then goto empty;
        goto test;
insert: set PW to S; set S to SS;
        set PWS to PWQWS;
        set PWQWS to PW;
        set PWQW to PW;
test:   if S $\neq$ □ then goto insert;
        set S to PS;
```

(during insertion, for any j the W-pointer of node D_j is always directed to the last node C of the new line with $CQ = D_j$ or to the auxiliary node P if such a C has not been inserted so far).

Nearly the same program can be applied for sorting with respect to equal operands p_i (all Q's have to be replaced by P). After this second path repeated occurrences of the same pair of operands will always appear in succession (actually, we have implemented a variant of radix sorting, see [6, Chap. 5.2.5]).

6.3. Integer-multiplication. In order to find out the actual value of N (on which a suitable choice of b will depend) our SMM reads the input and stores it in binary representation. Then it chooses n minimal as a multiple of 3 such that $n \cdot 2^n \geqq 2N$. Both factors are filled up to length $n \cdot 2^n$ by preceding zeros. Then they are split into pieces of length n according to

$$x = 2^{2n} \sum_{j=0}^{2^n-1} x_j 2^{nj}, \quad y = 2^{2n} \sum_{j=0}^{2^n-1} y_j 2^{nj}, \tag{6.8}$$

where $0 \leqq x_j < 2^{-n}$, $0 \leqq y_j < 2^{-n}$, and these numbers are integer multiples of 2^{-2n} (observe that certainly $x_j = y_j = 0$ for $j \geqq 2^{n-1}$).

Our goal is to compute the numbers

$$z_j = \sum_{\mu+\nu \doteq j} x_\mu y_\nu < 2^{-n}, \qquad (0 \leqq j < 2^n) \tag{6.9}$$

(later needed in (6.12)) which must be integer multiples of 2^{-4n}. Therefore it suffices to compute approximate values which differ from the z_j's by less than 2^{-4n-1}. This is achieved by means of the FFT essentially as described in § 3 of [13]. In our present notation formula (3.4) of [13] becomes

$$\hat{x}_k \hat{y}_k = \left(\sum_\mu x_\mu w_n^{\mu k}\right)\left(\sum_\nu y_\nu w_n^{\nu k}\right) = \sum_j z_j w_n^{jk} = \hat{z}_k,$$

where $w_n = \exp(2\pi i/2^n)$. A careful analysis (similar to [13, p. 286]) of the round-off errors in the FFTs and other steps involved shows that sufficient precision is obtained if the numerical calculations are carried out, for instance, by using fix-point numbers of binary length $6n$ and modulus $\leqq 1$ throughout (provided $n \geqq 6$).

With regard to these theoretical considerations our SMM chooses $b = n/3$, constructs the B-scale and translates the x_j, y_j into B-ary fix-point numbers with 18 fractional digits and one additional leading digit (in order to avoid overflow). All this is done in time $O(N)$, since we have

$$\text{(6.10)} \qquad \begin{aligned} n &= O(\log N), & 2^n &= O(N/\log N), \\ b &= O(\log N), & B &= O(N^{1/3}). \end{aligned}$$

For the FFTs the roots of unity $w_\nu = \exp(2\pi i/2^\nu)$ and their powers w_ν^κ are needed. In an initial phase $w_1 = -1$, $w_2 = i$, and $w_3, \cdots, w_n$ are prepared in full precision by means of the formula $w_{\nu+1} = (1+w_\nu)/|1+w_\nu|$. This computation can be carried out in any crude manner, since it requires only $O(\log N)$ many divisions and square-roots involving numbers of binary length $O(\log N)$. Then, after translation into B-ary form, the powers are obtained in $n-2$ *stages* by use of

$$\text{(6.11)} \qquad w_{\nu+1}^{2\kappa} = w_\nu^\kappa, \qquad w_{\nu+1}^{2\kappa+1} = w_\nu^\kappa \cdot w_{\nu+1}. \qquad (2 \leqq \nu < n)$$

Each stage requires at most $O(2^n)$ many multiplications of complex numbers, which can be done simultaneously. Here the technique of § 6.2 will be applied. At first, for each multiplication the SMM constructs a program, which consists of a bounded number of basic digital operations, since the real and the imaginary part of each of the factors have 19 digits only. Then each stage requires only a bounded number of *sweeps* in the interpreting mode. By Lemma 6.2. each sweep is performed in time $O(2^n + B^2b^2) = O(N/\log N)$, hence (cf. (6.10)) all the $n-2$ stages cost at most $O(N)$ many steps.

The same analysis applies to the FFTs. The transformations $(x_j) \mapsto (\hat{x}_j)$, $(y_j) \mapsto (\hat{y}_j)$ consist of n stages, in each of which $O(2^n)$ many operations of the form $h = f + g \cdot w_\nu^\kappa$ are to be performed simultaneously. Again the corresponding digital programs have bounded length. Therefore we obtain the time bound $n \cdot O(1) \cdot O(2^n + B^2b^2) = O(N)$. The simultaneous multiplications $\hat{z}_j = \hat{x}_j\hat{y}_j$ together with the back transformation $(\hat{z}_j) \mapsto (z_j)$ require $O(N)$ once more.

Finally the z_j's are translated into binary form and added up to yield the product

$$\text{(6.12)} \qquad xy = z = 2^{4n} \sum_{j=0}^{2^n-1} z_j 2^{nj},$$

which then is output. Again only $O(N)$ many steps are needed. This completes our proof of Theorem 6.1.

7. Final discussion. The existence of an SMM for integer-multiplication in linear time shows that any nonlinear lower bound for integer-multiplication on Turing machines will require some special argument referring to the specific nature of Turing machines and cannot be obtained by merely analyzing properties of multiplication.

Readers who have got the feeling that the result in Theorem 6.1 is inconsistent with their intuition will perhaps be pleased with a corollary easily obtainable by combining Theorems 6.1 and 4.1. In $O(N)$ many steps a RAM0 can internally produce numbers of binary length $O(\log N)$ at most. Therefore we get (cf. [1, p. 12] for the definition of logarithmic cost)

THEOREM 7.1. *There exists a successor RAM which performs integer-multiplication of N-bit numbers at logarithmic cost $O(N \log N)$.*

Such a smooth bound of course invites us to conjecture its optimality.

As announced the techniques of § 6.2 for economic parallel execution of programs allow us to derive a more general result which we state here without further proof.

THEOREM 7.2. *If a multitape Turing machine has time complexity $\leqq t(n)$ and outputs of length $\leqq L(n)$ for all inputs of length n, then its application to m many inputs of length n simultaneously can be simulated by a suitable SMM in time*

$$O\left(m\left(\frac{t(n)}{\log m}+L(n)+n\right)\right).$$

Finally we have to compare our SMM model with the *Kolmogorov-Uspenskii machines* (KUM) [7] for which we may assume similar input/output instructions. At first glance it may seem that both models are essentially the same, and it is indeed fairly obvious how to construct a real time simulation that shows KUM $\overset{r}{\to}$ SMM, but we are not sure about the reverse. The difficulty arises from the fact that KUMs use *undirected* graphs of bounded degree as storage elements. Therefore the nodes have bounded fan-out and bounded fan-in, whereas during an SMM computation the fan-in of some nodes will possibly increase with the size of the Δ-structure. If SMM $\overset{r}{\to}$ KUM is true, then this will lend further support to our main thesis; it its not true, we will nevertheless prefer the much more feasible Δ-structures.

In [3] an example is constructed which shows that there are KUMs which cannot be simulated in real time by any multidimensional Turing machine. By combining this with KUM $\overset{r}{\to}$ SMM we immediately get that SMM $\overset{r}{\to}$ TM is not true. Our Theorems 6.1 and 7.2 suggest the slightly stronger hypothesis that even SMM $\overset{l}{\to}$ TM must be wrong, where $\overset{l}{\to}$ denotes reducibility by simulation in linear time.

Acknowledgment. We are indebted to D. E. Knuth for reading the first draft of this paper and for contributing many improvements. In particular he has brought to the author's attention that the SMM model coincides with a special type of "linking automata" briefly explained in volume one of his book (cf. [4, pp. 462–463]) in 1968 already. Now he suggests calling them "pointer machines" which, in fact, seems to be the adequate name for these automata.

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. CHURCH, *An unsolvable problem of elementary number theory*, Amer. J. Math., 58 (1936), pp. 345–363.

[3] D. YU. GRIGORYEV, *Kolmogorov algorithms are stronger then Turing machines*, Investigations in Constructive Mathematics and Mathematical Logic VII, Matijasevič, Slisenko, eds., Izdat. Nauka, Leningrad, 1976, pp. 29–37. (In Russian).

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

[5] ———, *The Art of Computer Programming, vol.* 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 2nd ed. 1971.

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

[7] A. N. KOLMOGOROV AND V. A. USPENSKII, *On the definition of an algorithm*, Uspehi Mat. Nauk, 13 (1958), pp. 3–28; AMS Transl. 2nd ser. vol. 29 (1963), pp. 217–245.

[8] M. V. KUBINEC, *Recogniton of the self-intersection of a plane trajectory by Kolmogorov's algorithm*, Investigations in Constructive Mathematics and Mathematical Logic V, Matijasevič, Slisenko, eds., Izdat. Nauka, Leningrad, 1972, pp. 35–44. (In Russian).

[9] W. J. PAUL, *Komplexitätstheorie*, Teubner, Stuttgart, 1978.

[10] C. P. SCHNORR, *Rekursive Funktionen und ihre Komplexität*, Teubner, Stuttgart, 1974.

[11] A. SCHÖNHAGE, *Universelle Turing Speicherung*, Automatentheorie und Formale Sprachen, Dörr, Hotz, eds., Bibliogr. Institut, Mannheim, 1970, pp. 369–383.

[12] ———, *Real-time simulation of multidimensional Turing machines by storage modification machines*, Technical Memorandum 37, M.I.T. Project MAC, Cambridge, MA, 1973.

[13] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation großer Zahlen*, Computing, 7 (1971), pp. 281–292.

[14] J. E. HOPCROFT, W. J. PAUL AND L. G. VALIANT, *On time versus space and related problems*, Proc. 16th Ann. IEEE Symp. Foundations Comp. Sci. (Berkeley, CA), 1975, pp. 57–64.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0004 $01.00/0

A CORRECT PREPROCESSING ALGORITHM FOR BOYER–MOORE STRING-SEARCHING*

WOJCIECH RYTTER†

Abstract. We present the correction to Knuth's algorithm [2] for computing the table of pattern shifts later used in the Boyer–Moore algorithm for pattern matching.

Key words. algorithm, pattern-matching, string, overlap

The key to the Boyer–Moore algorithm for the fast pattern matching is the application of the table of pattern shifts which is denoted in [1] by Δ_2 and in [2] by dd'. Let us denote this table by D.

Assume that the pattern is given by the array pattern $[1:n]$, so D is given as an array D $[1:n]$. For every $1 \leqq j \leqq n$, $D[j]$ gives the minimum shift $d > 0$ such that the pattern with the right end placed at the position $k+d$ of the processing string is compatible with the part of string scanned before, where k is the last scanned position in the string and j is the last scanned position in the pattern.

The formal definition of D given in [2] is:

$$D[j] = \text{MIN}\,\{s+n-j \mid s \geqq 1 \text{ and } (s \geqq j \text{ or pattern } [j-s] \neq \text{pattern } [j]) \\ \text{and } ((s \geqq i \text{ or pattern } [i-s] = \text{pattern } [i]) \text{ for } j < i \leqq n)\}.$$

Algorithm A given by Knuth is:

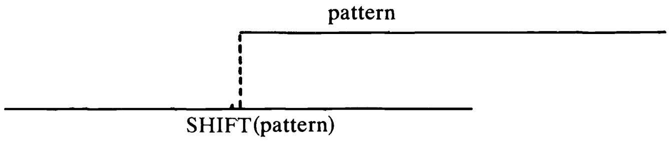
```
A1. for $k := 1$ step 1 until $n$ do $D[k] := 2^*n - k$;
A2. $j := n; t := n+1$;
    while $j > 0$ do
    begin
      $f[j] := t$;
      while $t \leqq n$ and pattern $[j] \neq$ pattern $[t]$ do
      begin
                          $D[t] := \text{MIN}\,(D[t], n-j)$;
                            $t := f[t]$;
      end
      $t := t-1; j :-j-1$;
    end;
A3. for $k := 1$ step 1 until $t$ do

                 $D[k] := \text{MIN}\,(D[k], n+t-k)$;
```

Algorithm A computes also the auxiliary table $f[0:n]$, for $j<n$ defined as follows: $f[j] = \min\{i \mid j < i \leqq n$ and pattern $[i+1] \cdots$ pattern $[n] =$ pattern $[j+1] \cdots$ pattern $[n+j-i]\}$; the final value of t corresponds to $f[0]$. $f[0]$ is the minimum nonzero shift of pattern on itself; let us denote this value by SHIFT (pattern).

* Received by the editor January 18, 1979, and in revised form May 25, 1979.

† Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, Universidad Nacional Autónoma de México Apartado Postal 20-726, México 20, D.F. On leave of absence from Institute of Informatics, Department of Mathematics, Warsaw University, Warsaw, Poland.



Take as inputs to Algorithm A the following two strings: pattern 1 = aaaaaaaaaa and pattern 2 = abaabaabaa. Denoting by def$D$ and D' respectively the value of D according to the definition and computed by Algorithm A we obtain the following results:

j	=	1	2	3	4	5	6	7	8	9	10
pattern 1[j]	=	a	a	a	a	a	a	a	a	a	a
Def$D[j]$	=	10	10	10	10	10	10	10	10	10	10
$D'[j]$	=	10	18	17	16	15	14	13	12	11	10
SHIFT(pattern 1) = 1.											
pattern 2[j]	=	a	b	a	a	b	a	a	b	a	a
Def$D[j]$	=	12	11	10	12	11	10	12	11	2	2
$D'[j]$	=	12	11	10	16	15	14	13	12	2	2
SHIFT(pattern 2) = 3.											

The disagreement between Def$D$ and D' demonstrates explicitly that Knuth's algorithm is incorrect.

There are three cases which are considered in the design of Algorithm A for computing the value of $D[j]$:

Case (1). $D[j]=2*n-j$. This is the most simple case computed in the part A1 of Algorithm A.

Case (2). $D[j]<n$ and pattern $[l]\neq$ pattern $[j]$, where $l=n-D[j]$. In this case $D[j]$ is computed in the part A2.

Case (3). $n\leqq D[j]<2*n-j$ and $j\leqq$ SHIFT(pattern) $=f[0]=t$. In this case $D[j]$ is computed in the part A3 of Algorithm A.

However, another case occurs which is not covered by Cases (1), (2) and (3):

Case (4). $n<D[j]<2*n-j$ and $j>$ SHIFT(pattern). For example it occurs for pattern = pattern 2 and $j=5$. To correct Algorithm A, we have to consider not only the minimal nonzero shift of the string on itself but all shifts, namely all i such that $0<i\leqq n$ and pattern $[i+1]\cdots$ pattern $[n]=$ pattern $[1]\cdots$ pattern $[n-i]$. Let us denote the set of all such i by ALLSHIFTS(pattern). Using the method of computing the failure function in the pattern-matching algorithm of Knuth, Morris and Pratt [2], we give below a correct version of the algorithm, where A1, A2 denote the corresponding parts of Algorithm A.

ALGORITHM B.

```
A1; A2;
$q := t$; $t := n+1-q$; $q1 := 1$;
    B1. $j1 := 1$; $t1 := 0$;
        **while** $j1 \leqq t$ **do**
        **begin**
        $f1[j1] := t1$;
        **while** $t1 \geqq 1$ **and** pattern $[j1] \neq$ pattern $[t1]$ **do** $t1 := f1[t1]$;
            $t1 := t1+1$; $j1 := j1+1$;
        **end**;
```

```
B2. while $q<n$ do
    begin
    for $k := q1$ step 1 until $q$ do $D[k] :=$ Min $(D[k], n+q-k)$;
    $q1 := q+1$; $q := q+t-f1[t]$;
        $t := f1[t]$; end;
```

The part B1 computes the auxiliary table $f1[1:t']$ where $t'=n+1-\text{SHIFT(pattern)}$, and the part B2 computes the values of $D[j]$ for both Cases (3) and (4).

$$f1[1]=0 \quad \text{and} \quad \text{for } 1<j\leqq t',$$

$$f1[j]=\max\{i|1\leqq i<j \text{ and pattern } [j-i+1]\cdots \text{pattern } [j-1] = \text{pattern } [1]\cdots \text{pattern } [i-1]\}.$$

The correctness of the part B2 follows from the following: If ALLSHIFTS(pattern) = $\{i_1, i_2, \cdots, i_k\}$ and $i_1=\text{SHIFT(pattern)}$ and $i_1<i_2<\cdots<i_k$ and $t_1=n+1-i_1$, $t_{p+1}=f1[t_p]$ for $p=1,2,\cdots,(k-1)$ then $i_{p+1}=i_p+t_p-t_{p+1}$ for $p=1,2,\cdots,(k-1)$.

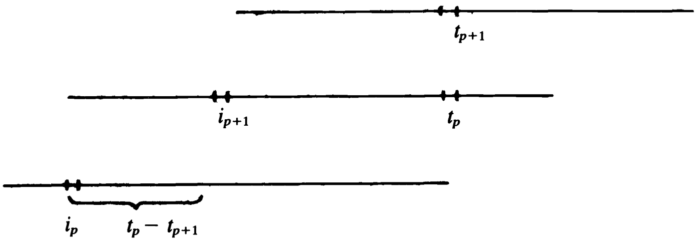


FIG. 1. *The graphical representation of the computation of* i_{p+1}.

Remark 1. The same table space can be used for f and $f1$.

Remark 2. The tables f and $f1$ are related in the following way: Let pattern' be the string resulting from reversing the string pattern and $f1$ be computed for the string pattern and f be computed for pattern'.

Then

$$f1[i]=n-f[n-i+1]+1 \quad \text{for } i=1,2,\cdots,(n+1).$$

Remark 3. Denote OVR(pattern) = $n-$SHIFT(pattern). So OVR(pattern) gives the maximum overlap of the pattern with itself. The difference in the time complexity of Algorithms A and B is proportional to OVR(pattern) which can be linear with respect to n. However, on the average it is very small for alphabets of the size greater than 1. Let $V(n,k)$ denotes the average value of OVR(pattern) taken over the set of all patterns of the length n over the same alphabet of the size k.

The rounded values of $V(n,2)$ for $n\leqq 14$ computed on B6700 are shown in Table 1.

TABLE 1

n	1	2	3	4	5	6	7
$V(n,2)$	0	0.5	0.75	1.0	1.125	1.281	1.375

n	8	9	10	11	12	13	14
$V(n,2)$	1.453	1.500	1.545	1.574	1.595	1.607	1.618

LEMMA. 1. *If* $k > 1$ *then* $V(n, k) < k/(k-1)^2$.
2. $V(n, 2) < 2$.
3. $V(n, k) < 1$ *for* $k > 2$.

Proof. Fix n and k and assume that $k > 1$. Let a_j be the number of patterns such that OVR(pattern) $= j$ for $j = 1, 2, \cdots, (n-1)$. Every pattern with OVR(pattern) $= j$ is determined by its prefix of the length $n-j$. So $a_j \leqq k^{n-j}$. Hence $V(n, k) = (\sum_{j=1}^{n-1} j \cdot a_j)/k^n \leqq \sum_{j=1}^{n-1} j \cdot (1/k)^j \leqq \sum_{j=1}^{\infty} j \cdot (1/k)^j = k/(k-1)^2$. Parts 2 and 3 of the lemma follow from 1. This ends the proof.

REFERENCES

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

[2] D. E. KNUTH, J. H. MORRIS, JR. AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0005 $01.00/0

THE PEBBLING PROBLEM IS COMPLETE IN POLYNOMIAL SPACE*

JOHN R. GILBERT,† THOMAS LENGAUER‡ AND ROBERT ENDRE TARJAN§

Abstract. In this paper we study a pebbling problem that models the storage requirements of various kinds of computation. Sethi has shown this problem to be *NP*-hard and Lingas has shown a generalization to be *P*-space complete. We prove the original problem *P*-space complete by using a modification of Lingas's proof. The pebbling problem is an example of a *P*-space complete problem not exhibiting any obvious quantifier alternation.

Key words. computational complexity, *P*-space completeness, pebbling, register allocation

1. Introduction. In this paper, we consider the following *pebble game*. Let G be a directed acyclic graph, all of whose vertices have at most two predecessors.[1] Given a collection of pebbles, we wish to place a pebble on a distinguished vertex of G, called the *goal*, starting with no pebbles on the graph, by applying the following rules;

(i) A pebble may be removed from a vertex at any time.

(ii) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

(iii) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be moved from a predecessor of v to v.

We shall consider time to be divided into unit steps. At each time step, one of rules (i)–(iii) is applied once. The *space* required by the pebbling is the maximum number of pebbles ever on the graph at one time; the *time* required is the number of applications of rules (i)–(iii).

Although it is convenient to allow use of the "sliding" rule (iii), the pebble game is not affected substantially if we omit this rule. The following lemma formalizes this observation. An independent proof of the lemma has been given by van Emde Boas and van Leeuwen [2], who also investigate the effect of the sliding rule on the pebbling time.

LEMMA 1. *For all $s > 0$, a graph G can be pebbled with s pebbles using rules* (i)–(iii) *if and only if it can be pebbled with $s+1$ pebbles using only rules* (i) *and* (ii).

Proof. Suppose G can be pebbled with s pebbles using rules (i)–(iii). We can replace each use of rule (iii) to move a pebble from a vertex x to a vertex y with two steps, first placing a pebble on y and then removing the pebble from x. This transformation allows us to pebble G with $s+1$ pebbles using only rules (i) and (ii).

Conversely, suppose a scheme exists for pebbling G with $s+1$ pebbles using rules (i) and (ii). We can pebble G with s pebbles as follows. Suppose there are $s+1$ pebbles at time t_0. Then the move at t_0 must be to place a pebble on some vertex y. If this is not

* Received by the editors May 17, 1979. This research was supported in part by the National Science Foundation under Grant MCS-75-22870-A02 and by the Office of Naval Research Contract N00014-76-C-0688.

† Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by a Hertz Fellowship.

‡ Bell Telephone Laboratories, Murray Hill, New Jersey 07974. The work of this author was supported in part by the German Academic Exchange Service.

§ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by a Guggenheim Fellowship.

[1] We shall use the following graph-theoretic terminology. A directed graph $G = (V, E)$ is a collection of *vertices* V and a collection of *edges* E. Each edge is an ordered pair (v, w) of distinct vertices. If (v, w) is an edge, v is a *predecessor* of w and w is a *successor* of v. A *source* is a vertex with no predecessors; a *sink* is a vertex with no successors. A *path* from v to w is a sequence of vertices $v = v_1, v_2, \cdots, v_k = w$ such that v_{i+1} is a successor of v_i for $1 \leqq i < k$. A *cycle* is a path of at least two vertices from v to v. A graph is *acyclic* if it has no cycles.

the final move, the move at t_0+1 must be to remove a pebble from some vertex x. If x is a predecessor of y we replace these two moves with a single use of rule (iii), sliding a pebble from x to y. If x is not a predecessor of y we reverse the order of the moves, first removing the pebble from x and then placing it on y. If t_0 is the final move we slide a pebble from any predecessor of y to y. (If y has no predecessors we just pebble it, using one pebble, since $s>0$.) If we apply this transformation to all times at which $s+1$ pebbles are on G, we get a scheme to pebble G with s pebbles using rules (i)–(iii). □

The pebble game has been used to model register allocation [14], to study flowcharts and recursive schemata [9], and to analyze the relative power of time and space as Turing machine resources [1], [6]. Our interest lies in determining the computational complexity of the following problem, which we call the *pebbling problem*: given a graph G, can a given vertex v in G be pebbled using no more than s pebbles? This problem is not necessarily in *NP*,[2] since the number of moves necessary to pebble G with s pebbles may not be polynomially bounded [10]. However, the problem is in polynomial space, since a sequence of moves can be guessed and checked by a nondeterministic machine; only polynomial space is necessary to remember a single arrangement of pebbles on the graph (or *configuration*). By Savitch's theorem [12], such a nondeterministic machine can be converted into a deterministic machine for which the space bound is at most squared.

Many of the known P-space complete problems, such as the quantified Boolean formula problem [15] and various game problems [3], [4], [7], [13] possess an obvious quantifier alternation not present in the pebbling problem. Thus we might expect difficulties in showing the pebbling problem P-space complete. Sethi [14] was able to show the problem *NP*-hard, and *NP*-complete in the special case that each vertex can be pebbled only once. Lingas [8] generalized the problem by allowing "or" vertices (an "or" vertex can be pebbled if at least one of its predecessors is pebbled) and proved the generalized version P-space complete. We shall prove the original pebbling problem P-space complete by modifying Lingas's construction. The next section of the paper contains the proof. The concluding section mentions some additional consequences of our construction.

2. The construction.

Quantified Boolean formulas. In order to prove the pebbling problem P-space complete, we must reduce a known P-space complete problem to the pebbling problem. For this purpose we choose the quantified formula problem (QBF) [15]: Determine whether a quantified formula of the form $Q_1x_1Q_2x_2\cdots Q_nx_nF$ is true, where each x_i is a Boolean variable, each Q_i is either an existential or a universal quantifier, and F is an unquantified Boolean formula involving only the variables x_i, in conjunctive normal form with exactly three literals per clause. From the quantified formula we construct a graph G with a goal vertex q_1 and a number of pebbles s such that the quantified formula is true if and only if q_1 can be pebbled with s pebbles. It will be evident that the transformation from formula to graph can be accomplished in logarithmic space; it follows that the pebbling problem is log-space complete in P-space.

We need a notation to denote substitution of truth values in F. For technical reasons we substitute for the literals rather than for the variables. We use $F(e_1, e_2, \cdots, e_{2k-1}, e_{2k})$ to denote the formula obtained from F by replacing each occurrence of x_i by e_{2i-1} and each occurrence of $\bar{x}_i$ by e_{2i}, for $1 \leqq i \leqq k$, where each e_j is either true or false. Thus F (true, false) denotes making x_1 true (and $\bar{x}_1$ false), F (false, true) denotes making

[2] We use standard concepts from complexity theory without defining them. For a thorough discussion of *NP*, *P*-space, and completeness, see [5].

x_1 false (and $\bar{x}_1$ true), and F (false, false) denotes the "double false" substitution making x_1 false and $\bar{x}_1$ also false. (We shall have no need to consider the "double true" substitution F (true, true).) Note that if $F(e_1, \cdots, e_{2k-2}, \text{false}, \text{false}, e_{2k+1}, \cdots, e_{2n})$ is true, then *both* $F(e_1, \cdots, e_{2k-2}, \text{true}, \text{false}, e_{2k+1}, \cdots, e_{2n})$ and $F(e_1, \cdots, e_{2k-2}, \text{false}, \text{true}, e_{2k+1}, \cdots, e_{2n})$ are true. Thus if $Q_{k+1}x_{k+1} \cdots Q_n x_n F(e_1, \cdots, e_{2k-2}, \text{false}, \text{false})$ is true, so is $\forall x_k Q_{k+1}x_{k+1} \cdots Q_n x_n F(e_1, \cdots, e_{2k-2})$.

Preliminary observations. An important building block in our construction is the "pyramid" graph shown in Fig. 1, which we shall abbreviate with a triangle as indicated in the figure. Cook [1] has proved that the sink (or *apex*) of a pyramid with k sources can be pebbled if and only if at least k pebbles are used. We can use a pyramid to lock a pebble on a given vertex for a given time interval. We do this by making the vertex the apex of a pyramid which is so large that in order to repebble the vertex, so many pebbles have to be taken off the graph for use on the pyramid that the results achieved after the vertex was first pebbled are lost.

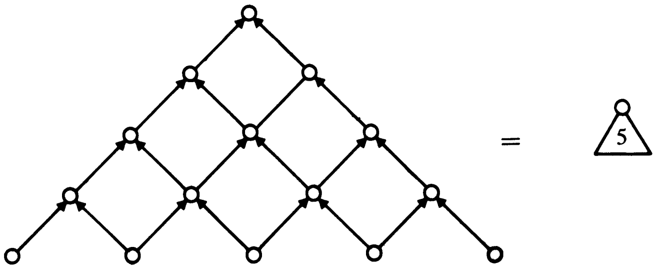


FIG. 1. *A 5-pyramid.*

Note also that if any source of a k-pyramid contains a pebble that cannot be moved, then the apex can be pebbled with $k-1$ additional pebbles.

We now make some general remarks about pebbling strategies that are similar to those of Pippenger [11]. We partition the pebble placements into *necessary* and *unnecessary* placements as follows. The first placement on the goal vertex is necessary; all other placements on the goal vertex are unnecessary. A placement on any other vertex v is necessary if and only if the pebble placed remains on v until a necessary placement occurs on a successor of v. The necessary placements are well-defined since the graph is acyclic. Deletion of all unnecessary placements from a pebbling strategy results in another pebbling strategy. We call a pebbling strategy with no unnecessary placements *frugal*. The following statements are true of any frugal pebbling strategy.

(i) At all times after the first placement on a vertex v, some path from v to the goal vertex contains a pebble.

(ii) At all times after the last placement on a vertex v, all paths from v to the goal vertex contain a pebble. (This is true also of nonfrugal pebbling strategies.)

(iii) The number of placements on a nongoal vertex is bounded by the total number of placements on its successors.

We call a pebbling strategy *normal* if it is frugal and if it pebbles each pyramid P in G as follows: after the first pebble is placed on P, no placement or removal occurs outside P until the apex of P is pebbled and all other pebbles are removed from P. No new placement occurs on P until after the pebble on the apex of P is removed.

LEMMA 2. *If the goal vertex is not inside a pyramid, any pebbling strategy can be transformed into a normal pebbling strategy without increasing the number of pebbles used.*

Proof. Consider any pebbling strategy. First obtain a frugal strategy by deleting all unnecessary placements; this does not increase the number of pebbles used. Then let t_1 be a time at which a pebble is placed on a k-pyramid P. Let $[t_0, t_2]$ be the largest time interval containing t_1 such that P is never pebble-free during $[t_0, t_2]$. Since the pebbling strategy is frugal and the goal vertex is not in P, the only pebble on P at time t_2 is on the apex of P. Since at time t_0-1 no pebbles are on P, there must be a time t_3 during $[t_0, t_2]$ at which k pebbles are on P. Modify the pebbling strategy as follows. Delete all placements and removals from P during $[t_0, t_2]$. Insert at t_3 a continuous sequence of moves that pebbles the apex of P using k pebbles and then removes all pebbles on P except the one on the apex. This transformation results in a pebbling strategy since no vertex in P has a predecessor outside P, and the only vertex in P that precedes vertices outside P is the apex. If the inserted sequence contains no unnecessary placements, then the transformed strategy is frugal. Furthermore it uses no more pebbles than the original strategy. Repeating this transformation for each placement on a pyramid results in a normal strategy. □

Details of the construction. To describe the construction we need a little more notation. Recall that n is the number of quantifiers. The number of pebbles we allow is $s=3n+3$. For $1 \leqq i \leqq n+1$, let $s_i = s-3i+3$; thus $s_1 = s$ and $s_{n+1}=3$. Roughly speaking, we use three pebbles to keep track of each quantifier and its associated variable, and three more to check the validity of the clauses of F under a given assignment to the variables. Let F contain m clauses $(l_{j1} \vee l_{j2} \vee l_{j3})$ for $1 \leqq j \leqq m$, where each l_{jk} is a literal. For any variable x, we shall regard $\bar{\bar{x}}$ as synonymous with x.

The graph G to be constructed consists of $n+m$ blocks of vertices, one for each quantifier and its associated variable, and one for each clause in F. The quantifier block for Q_ix_i includes four vertices to represent the variable x_i, as illustrated in Fig. 2. Two pebbles placed on this subgraph encode the truth values of x_i and $\bar{x}_i$ as illustrated in Fig. 2(b)–(d). The remainder of a quantifier block depends on the quantifier.

Figure 3 illustrates a universal quantifier block. The way this block works is as follows. There are essentially two ways to pebble q_i with s_i pebbles: (i) pebble q_{i+1} twice with s_{i+1} pebbles, each time with three pebbles fixed on the ith quantifier block, once representing $\bar{x}_i$ true and once representing x_i true (the third pebble is fixed on d_i or a_i respectively); or (ii) pebble q_{i+1} once with s_{i+1} pebbles, while three pebbles representing x_i false and $\bar{x}_i$ false are fixed on the ith quantifier block.

Figure 4 illustrates an existential quantifier block. The only way to pebble q_i with s_i pebbles is to pebble q_{i+1} with s_{i+1} pebbles, while three pebbles representing one of the three possible truth assignments to x_i and $\bar{x}_i$ are fixed on the ith quantifier block (the third pebble is fixed on d_i).

Figure 5 illustrates the block of vertices representing a clause. After $s-3$ pebbles are used on the quantifier blocks to fix an assignment to the literals, the remaining three pebbles are available to pebble the clause blocks. For each literal l_{jk}, there is a fixed pebble on vertex l_{jk} if the literal is true, or on vertex l'_{jk} if the literal is false. Thus if F is valid, the clause pyramids can be pebbled in the order $p_0, p_1, \cdots, p_m = q_{n+1}$ with three pebbles; however, if some clause $(l_{j1} \vee l_{j2} \vee l_{j3})$ is false, p_j is the apex of an empty 4-pyramid and cannot be pebbled with three pebbles.

Figure 6 illustrates the entire construction. Note that p_0 is a single vertex, and that $p_m = q_{n+1}$.

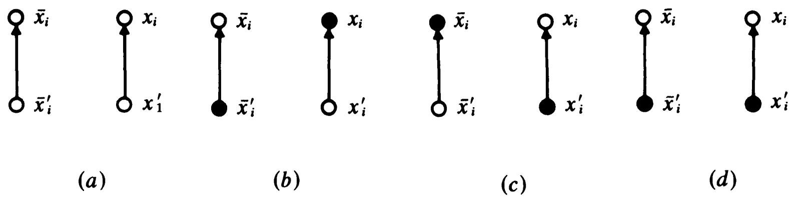


FIG. 2. (a) *Vertices representing a variable*; (b) *true configuration*; (c) *false configuration*; (d) *double false configuration.*

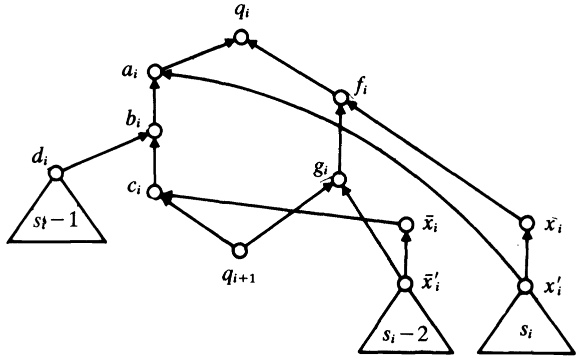


FIG. 3. *Universal quantifier block. Vertex q_{i+1} is part of the $i+1$-st quantifier block.*

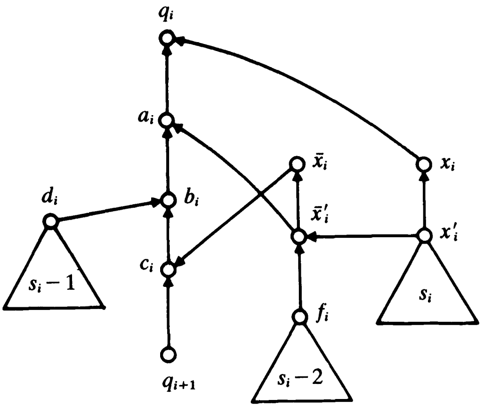


FIG. 4. *Existential quantifier block. Vertex q_{i+1} is part of the $i+1$-st quantifier block.*

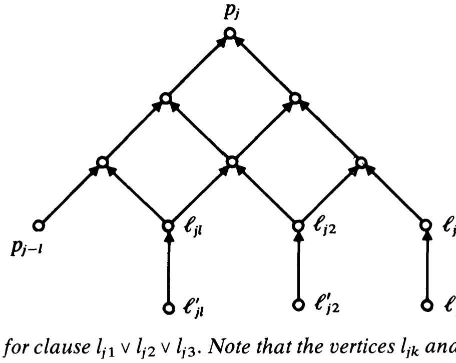


FIG. 5. *Block of vertices for clause $l_{j1} \vee l_{j2} \vee l_{j3}$. Note that the vertices l_{jk} and l'_{jk} occur among the quantifier blocks. Vertex p_{j-1} is part of the $j-1$-st clause block; p_0 is a single vertex and $p_m = q_{n+1}$.*

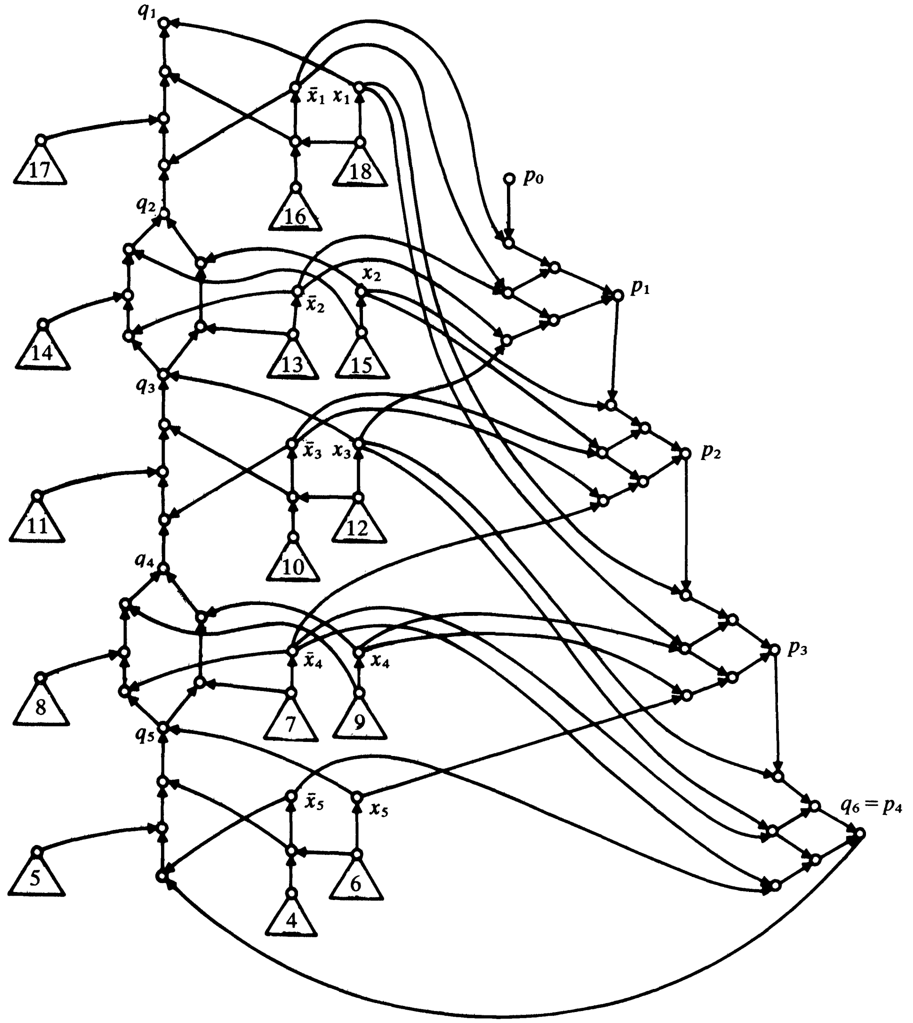


FIG. 6. *Graph for* $E=\exists x_1\forall x_2\exists x_3\forall x_4\exists x_5(\bar{x}_1\vee\bar{x}_2\vee x_3)\wedge(x_2\vee\bar{x}_3\vee\bar{x}_4)\wedge(x_1\vee x_4\vee x_5)\wedge(x_3\vee\bar{x}_4\vee\bar{x}_5)$. *Number of pebbles* $=5\cdot 3+3=18$.

Proof of the reduction. Our main result is as follows.

THEOREM 1. *The quantified Boolean formula* $Q_1x_1Q_2x_2\cdots Q_nx_nF$ *is true if and only if vertex* q_1 *in the graph* G *constructed as above can be pebbled with* $s=3n+3$ *pebbles.*

We prove this theorem by means of two lemmas which state that if we use $s-s_i$ pebbles to fix truth values for the literals corresponding to the first $i-1$ variables, then we can pebble q_i with the remaining s_i pebbles if and only if the quantified formula is valid after making the appropriate substitution. The lemmas are proved by induction on i. For $1\leqq i\leqq n+1$, we define N_i to be the set of configurations fixing truth values for the literals corresponding to the first $i-1$ variables. An arrangement of exactly $s-s_i$ pebbles on G is in N_i if and only if, for $1\leqq j<i$, two conditions hold:

(1) If $Q_j=\forall$ there are exactly three pebbles on the jth quantifier block, on one of the following three sets of vertices:

(a) $\{a_j, x_j, \bar{x}'_j\}$, indicating x_j true;

(b) $\{d_j, \bar{x}_j, x'_j\}$, indicating x_j false, or

(c) $\{d_j, x'_j, \bar{x}'_j\}$, indicating double false.

(2) If $Q_j = \exists$, there are exactly three pebbles on the jth quantifier block, on one of the following three sets of vertices:

(a) $\{d_j, x_j, \bar{x}'_j\}$, indicating x_j true;

(b) $\{d_j, \bar{x}_j, x'_j\}$, indicating x_j false, or

(c) $\{d_j, x'_j, \bar{x}'_j\}$, indicating double false.

Note that N_1 contains only the configuration with no pebbles on the graph, and N_{n+1} contains all configurations in which a truth assignment has been made to each literal and three pebbles remain to test whether the assignment makes F true.

LEMMA 3. *Let $1 \leqq i \leqq n+1$. Suppose the graph is initially in a configuration in N_i. For $1 \leqq j < i$, let e_{2j-1} be the truth assignment defined for x_j by that configuration, and let e_{2j} be the truth assignment defined for $\bar{x}_j$. If $Q_i x_i \cdots Q_n x_n F(e_1, e_2, \cdots, e_{2i-3}, e_{2i-2})$ is true, then vertex q_i can be pebbled with s_i additional pebbles without moving any of the $s - s_i$ pebbles initially on the graph.*

Proof. Proof is by induction on i from $n+1$ to 1.

Basis. Let $i = n+1$ and suppose that the assignment defined by the N_i configuration makes F true. We must show that vertex $q_{n+1} = p_m$ can be pebbled with $s_{n+1} = 3$ additional pebbles without moving any of the pebbles of the N_i configuration.

For each clause $(l_{j1} \vee l_{j2} \vee l_{j3})$ of F, there is a pebble of the configuration on l_{j1} or l_{j2} or l_{j3}, and if there is not a pebble on l_{jk} then there is a pebble on l'_{jk}, for $1 \leqq k \leqq 3$. It follows that with three additional pebbles we can pebble $p_0, p_1, \cdots, p_m$ in turn as described earlier. Note that we need at least three additional pebbles, since each p_j for $j \geqq 1$ is the apex of a three-source pyramid initially containing no pebbles.

Inductive step. Suppose that the lemma holds for $i+1$, and that the assignment defined by the N_i configuration makes the substituted formula $Q_i x_i \cdots Q_n x_n F(e_1, e_2, \cdots, e_{2i-3}, e_{2i-2})$ true.

Case 1 (universal quantifier). Suppose $Q_i = \forall$. Then

$$Q_{i+1}x_{i+1} \cdots Q_n x_n F(e_1, \cdots, e_{2i-2}, \text{true}, \text{false}) \quad \text{and}$$

$$Q_{i+1}x_{i+1} \cdots Q_n x_n F(e_1, \cdots, e_{2i-2}, \text{false}, \text{true}) \quad \text{are both true.}$$

Vertex q_i can be pebbled with s_i pebbles as follows. First use all s_i pebbles to pebble x'_i, leaving a pebble there. Then use the remaining $s_i - 1$ pebbles to pebble d_i, leaving a pebble there, and the remaining $s_i - 2$ pebbles to pebble $\bar{x}_i$, leaving a pebble there. The current configuration is in N_{i+1}, representing x_i false. Applying the induction hypothesis, pebble q_{i+1} with the remaining $s_{i+1} = s_i - 3$ pebbles. Move the pebble on q_{i+1} to c_i, b_i and a_i. Move the pebble on x'_i to x_i. Leaving pebbles on a_i and x_i, pick up the rest of the pebbles and use the $s_i - 2$ free pebbles to pebble $\bar{x}'_i$, leaving a pebble there. The current configuration is in N_{i+1}, representing x_i true. Applying the induction hypothesis, pebble q_{i+1} again. Finish by moving the pebble on q_{i+1} to g_i, f_i, and q_i.

If $Q_{i+1}x_{i+1} \cdots Q_n x_n F(e_1, \cdots, e_{2j-2}, \text{false}, \text{false})$ is true, there is a way to pebble q_i that only pebbles q_{i+1} once. First pebble x'_i, d_i, and $\bar{x}'_i$, which gives a configuration in N_{i+1} representing x_i and $\bar{x}_i$ both false. Applying the induction hypothesis, pebble q_{i+1}. There are now $s_i - 4 \geqq 2$ free pebbles. Place one on $\bar{x}_i$ and move it to c_i, b_i, and a_i. Move the pebble on $\bar{x}'_i$ to g_i, and finish by moving the pebble on x'_i to x_i, f_i, and q_i.

Case 2 (existential quantifier). Suppose $Q_i = \exists$. Then either

$$Q_{i+1}x_{i+1} \cdots Q_n x_n F(e_1, \cdots, e_{2i-2}, \text{true}, \text{false}) \quad \text{or}$$

$$Q_{i+1}x_{i+1} \cdots Q_n x_n F(e_1, \cdots, e_{2i-2}, \text{false}, \text{true}) \quad \text{is true.}$$

Suppose first that the former is the case. Vertex q_i can be pebbled with s_i pebbles as follows. First pebble x'_i, leaving a pebble there. Then pebble d_i and f_i, leaving pebbles there. Move the pebble on f_i to $\bar{x}'_i$ and move the pebble on x'_i to x_i. The current configuration is in N_{i+1}, representing x_i true. Applying the induction hypothesis, pebble q_{i+1} with the remaining $s_{i+1} = s_i - 3$ pebbles. There are now $s_i - 4 \geqq 2$ free pebbles. Place one on $\bar{x}_i$ and finish by moving it to c_i, b_i, a_i, and q_i.

Alternatively, suppose that $Q_{i+1}x_{i+1} \cdots Q_nx_nF(e_1, \cdots, e_{2i-2}, \text{false}, \text{true})$ is true. To pebble q_i with s_i pebbles, begin by pebbling x'_i, d_i, and f_i in turn, leaving pebbles there. Move the pebble on f_i to $\bar{x}'_i$ and $\bar{x}_i$, which gives a configuration in N_{i+1} representing x_i false. Applying the induction hypothesis, pebble q_{i+1}. Move the pebble on q_{i+1} to c_i and b_i. Pick up all the pebbles except those on b_i and x'_i, and use the $s_i - 2$ free pebbles to pebble f_i. Move the pebble on f_i to $\bar{x}'_i$ and a_i, and finish by moving the pebble on x'_i to x_i and q_i. □

LEMMA 4. *Let $1 \leqq i \leqq n+1$. Suppose the graph is initially in a configuration in N_i. For $1 \leqq j < i$, let e_{2j-1} be the truth assignment defined for x_j by that configuration, and let e_{2j} be the truth assignment defined for $\bar{x}_j$. If vertex q_i can be pebbled with s_i additional pebbles without moving any of the $s - s_i$ pebbles initially on the graph, then $Q_ix_i \cdots Q_nx_nF(e_1, e_2, \cdots, e_{2i-3}, e_{2i-2})$ is true.*

Proof. Again, proof is by induction on i from $n+1$ to 1.

Basis. Let $i = n+1$ and suppose $q_i = p_m$ can be pebbled with $s_i = 3$ pebbles without moving any pebbles in the N_i configuration. Then each pyramid of size four representing a clause of F must contain at least one pebble of the N_i configuration, corresponding to a true literal; that is, the assignment defined by the N_i configuration must make F true.

Inductive step. Suppose that the lemma holds for $i+1$, and that there is a strategy which pebbles q_i with s_i pebbles without moving any pebbles in the N_i configuration. By Lemma 2 we can assume that the strategy is normal.

Case 1 (universal quantifier). Suppose $Q_i = \forall$. By frugality, each of q_i, a_i, b_i, c_i, d_i, f_i, and g_i is only pebbled once.

Let t_0 be the last time s_i pebbles appear on the s_i-pyramid. After t_0, x'_i is only pebbled once. At t_0 no pebbles are on vertices outside the s_i-pyramid. Since the pebbling is frugal, no placement before t_0 occurs outside the s_i-pyramid. Thus x'_i is only pebbled once, and this occurs before anything else happens. Let t_1 be the time x'_i is pebbled. From t_1 until q_i is pebbled, a pebble is on x'_i, x_i, or f_i. From t_1 until a_i is pebbled, a pebble is on x'_i.

To pebble a_i requires pebbling d_i. This requires removing all pebbles from the graph except the one on x'_i. By normality, therefore, d_i is pebbled before anything other than x'_i, and a pebble remains on d_i until b_i is pebbled. To pebble b_i requires pebbling c_i and hence $\bar{x}'_i$. This requires removing all pebbles except those on x'_i and d_i. Therefore $\bar{x}'_i$ is pebbled immediately after d_i and a pebble remains on $\bar{x}'_i$ or $\bar{x}_i$ until c_i is pebbled, which happens before b_i is pebbled. By normality, all pebbles except the one on $\bar{x}'_i$ are removed from the $s_i - 2$-pyramid as soon as $\bar{x}'_i$ is pebbled. Let t_2 be the time these pebbles are removed, and let t_3 be the first time after t_2 that q_{i+1} is pebbled.

At t_2 there are pebbles on x'_i, d_i, and $\bar{x}'_i$. Pebbles must remain on x'_i and d_i until t_3, and a pebble must be on either $\bar{x}'_i$ or $\bar{x}_i$ until t_3. Suppose first that a pebble remains on $\bar{x}'_i$ from t_2 until t_3. The configuration at t_2 is in N_{i+1} with a double false assignment to x_i, and none of the pebbles on the graph at t_2 can be removed until t_3. Therefore the induction hypothesis says that $Q_{i+1}x_{i+1} \cdots Q_nx_nF(e_1, \cdots, e_{2i-2}, \text{false}, \text{false})$ is true, so $\forall x_iQ_{i+1}x_{i+1} \cdots Q_nx_nF(e_1, \cdots, e_{2i-2})$ is true and the lemma holds in this case.

Alternatively, suppose that the pebble on $\bar{x}_i'$ does not remain until t_3. In this case we will argue that q_{i+1} must be pebbled twice, first with a false assignment to x_i and then with a true assignment to x_i.

Either $\bar{x}_i'$ or $\bar{x}_i$ must have a pebble from t_2 to t_3. The only successors of $\bar{x}_i'$ are $\bar{x}_i$ and g_i, and g_i cannot be pebbled before t_3. Therefore we can rearrange the strategy so that at t_2+1 the pebble on $\bar{x}_i'$ is moved to $\bar{x}_i$, where it must remain until t_3. The configuration at t_2+1 is then in N_{i+1} with a false assignment to x_i, and none of the pebbles on the graph at t_2+1 can be removed until t_3. By the induction hypothesis, $Q_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2},\text{false},\text{true})$ is true.

At t_3, there are pebbles on d_i, $\bar{x}_i$, x_i', and q_{i+1}. Vertices q_i, a_i, b_i, c_i, f_i, and g_i are vacant because they can't be pebbled before q_{i+1} is pebbled. Vertex $\bar{x}_i'$ couldn't have been repebbled between t_2+1 and t_3 because three pebbles were fixed on d_i, $\bar{x}_i$, and x_i' during that interval; thus $\bar{x}_i'$ and (by normality) the entire s_i-2-pyramid are also vacant at t_3. There may or may not be a pebble on x_i at t_3.

We will now show that immediately after t_3, a configuration in N_{i+1} with a true assignment to x_i is created, and that q_{i+1} must be repebbled while the pebbles in the configuration are fixed.

By frugality, the pebble on q_{i+1} at t_3 remains until either c_i or g_i is pebbled. Vertex q_{i+1} cannot retain a pebble until g_i is pebbled, because to pebble g_i requires placing all but two of the pebbles on the s_i-2-pyramid, and in addition to the pebble on q_{i+1}, two pebbles are fixed, one on x_i', x_i, or f_i and the other on d_i, b_i, or a_i, until q_i is pebbled. Thus the pebble on q_{i+1} at t_3 remains until c_i is pebbled and is removed before g_i is pebbled. Since $\bar{x}_i$ has a pebble at t_3, we can rearrange the strategy so that the pebble on q_{i+1} is moved to c_i at t_3+1.

Now the only successors of c_i and b_i are b_i and a_i respectively, and since d_i and x_i' both contain pebbles at t_3+1, we can rearrange the strategy so that the pebble on c_i is moved to b_i at t_3+2 and to a_i at t_3+3. A pebble must then remain on a_i until q_i is pebbled. Since a_i is only pebbled once and is the only successor of x_i' except x_i, we can further rearrange the strategy so that the pebble on x_i' is moved to x_i at t_3+4 (or is picked up if there is already a pebble on x_i).

At t_3+4, a_i contains a pebble that will remain until q_i is pebbled, and x_i contains a pebble that will remain until f_i is pebbled. Vertex $\bar{x}_i'$ must be repebbled before f_i is pebbled, which must happen before q_i is pebbled. To pebble $\bar{x}_i'$ requires all the pebbles except the ones on a_i and x_i, so by normality $\bar{x}_i'$ is pebbled immediately after t_3+4, and is only pebbled once before f_i is pebbled. Let t_4 be the time all the pebbles except the one on $\bar{x}_i'$ are removed from the s_i-2-pyramid after $\bar{x}_i'$ is first pebbled after t_3+4. At t_4 there are pebbles on a_i, x_i, and $\bar{x}_i'$, and nowhere else on the ith quantifier block. This configuration is in N_{i+1} with a true assignment to x_i, and none of the pebbles on the graph at t_4 can be removed until after q_{i+1} is repebbled. By the induction hypothesis $Q_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2},\text{ true},\text{ false})$ is true, Therefore $\forall x_iQ_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2})$ is true. This finishes the inductive step for a universal quantifier.

Case 2 (existential quantifier). Suppose $Q_i=\exists$. By frugality, each of q_i, a_i, b_i, c_i, d_i, and q_{i+1} is only pebbled once. Exactly as in Case 1, normality implies that x_i' is only pebbled once, and is pebbled before anything else happens. A pebble remains on x_i' or x_i until q_i is pebbled, and a pebble remains on x_i' or $\bar{x}_i'$ until a_i is pebbled. To pebble a_i requires pebbling d_i, which requires removing all pebbles from the graph except one on x_i'. Thus d_i is pebbled before anything else except x_i', and a pebble remains on d_i until b_i is pebbled.

To pebble b_i requires pebbling c_i and hence f_i. To pebble f_i requires removing all pebbles except those on x_i' and d_i. Thus f_i is pebbled only once before b_i is pebbled, and this happens immediately after d_i is pebbled. A pebble remains on f_i, $\bar{x}_i'$, or $\bar{x}_i$ until c_i is pebbled.

The only successor of f_i is $\bar{x}_i'$, and a pebble remains on x_i' until $\bar{x}_i'$ is pebbled, so we can rearrange the strategy so that the first move after picking up the pebbles on the s_i-2-pyramid (except the one on f_i) is to move the pebble on f_i to $\bar{x}_i'$. Let t_1 be the time of this move, and let t_2 be the time q_{i+1} is pebbled. Note that since f_i is not repebbled between t_1 and t_2, neither is $\bar{x}_i'$. At t_1 there are pebbles on x_i', $\bar{x}_i'$, and d_i, and until t_2 there must be pebbles on x_i' or x_i, x_i' or $\bar{x}_i'$, $\bar{x}_i'$ or $\bar{x}_i$, and d_i.

Case 2a. The pebble on x_i' is removed before t_2. Since the only successors of x_i' are x_i and $\bar{x}_i'$, and $\bar{x}_i'$ is not repebbled before t_2, we can rearrange the strategy so that the pebble on x_i' is moved to x_i at t_1+1. The configuration at t_1+1 is then in N_{i+1} with a true assignment to x_i, and none of the pebbles can be removed until t_2. By the induction hypothesis,

$$Q_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2},\text{true},\text{false})\text{ is true.}$$

Case 2b. A pebble remains on x_i' until t_2, and the pebble on $\bar{x}_i'$ is removed before t_2. We can rearrange the strategy so that the pebble on $\bar{x}_i'$ is moved to $\bar{x}_i$ at t_1+1. The configuration at t_1+1 is in N_{i+1} with a false assignment to x_i, and no pebble can be removed until t_2. By the induction hypothesis,

$$Q_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2},\text{false},\text{true})\text{ is true.}$$

Case 2c. Pebbles remain on x_i' and $\bar{x}_i'$ until t_2. The configuration at t_1 is in N_{i+1} with a double false assignment to x_i, and no pebble is removed until t_2. By the induction hyothesis,

$$Q_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2},\text{false},\text{false})\text{ is true.}$$

In each of subcases 2(a)–(c), $\exists x_iQ_{i+1}x_{i+1}\cdots Q_nx_nF(e_1,\cdots,e_{2i-2})$ is true. This completes the inductive step for an existential quantifier, and the proof of the lemma. □

Proof of Theorem 1. Theorem 1 is simply the case $i=1$ of Lemmas 3 and 4. □

3. Remarks. Variants of our construction give a couple of additional interesting results. Lingas [8] exhibited an infinite family of graphs with the following property: pebbling an n-vertex graph in the family with the minimum number of pebbles requires $2^{\Omega(n^{1/3})}$ time, but allowing two additional pebbles reduces the time to $O(n)$. P. van Emde Boas and van Leeuwen [2] independently obtained a similar result; in their construction only one additional pebble is necessary to reduce the pebbling time to $O(n)$.

We can obtain such a result as follows: Select any value of k. Let $s=3k+2$. Construct a graph G_k corresponding to the formula

$$\forall x_1\forall x_2\cdots\forall x_k(x_1\vee\bar{x}_1)\wedge(x_2\vee\bar{x}_2)\wedge\cdots\wedge(x_k\vee\bar{x}_k)$$

as described in § 2, representing each clause by a three-source pyramid as in Fig. 7 instead of by a four-source pyramid as in Fig. 5. G_k has $O(k^3)$ vertices and requires at least s pebbles, since it contains a pyramid of size s. Since the formula is true, G_k can be pebbled with s pebbles, but only in $\Omega(2^k)$ time, since any double false substitution makes the formula false. With $s+1$ pebbles, G_k can be pebbled in $O(k^3)$ time by selecting the double false assignment for all variables and using the remaining three pebbles to pebble the clause pyramids. This construction improves the result of Lingas

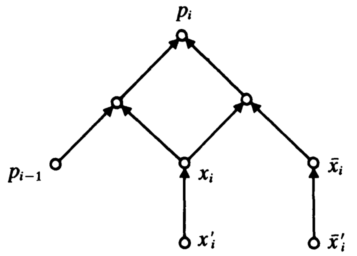


FIG. 7. *Block of vertices for clause $x_i \vee \bar{x}_i$.*

because only one extra pebble is needed to reduce the pebbling time to linear, and it improves the result of van Emde Boas and van Leeuwen because the graph size is $O(s^3)$ rather than $O(s^4)$, where s is the minimum number of pebbles necessary to pebble the graph.

Another variant of our construction shows the following problem to be P-space complete: given a graph G and a number of pebbles s sufficient to pebble a given vertex v, can v be pebbled within a specified time bound t? We assume t is expressed in binary notation; if t is expressed in unary, it is immediate from Sethi's result [14] that the problem is NP-complete. We shall reduce the quantified Boolean formula problem to this problem of pebbling with a time bound.

Let $E = Q_1x_1 \cdots Q_nx_nF$ be a quantified Boolean formula. Construct a new formula $E' = \exists y_1 \cdots \exists y_nQ_1x_1 \cdots Q_nx_nF'$, where F' is formed from F by adding clauses $x_i \vee \bar{x}_i \vee y_i$ and $x_i \vee \bar{x}_i \vee \bar{y}_i$ to F for $1 \leqq i \leqq n$. The new formula E' is true if and only if E is true, but a double false substitution for any universally quantified variable in E' makes F' false. Let m be the number of clauses on F' (note that $m \geqq 2n$), and let $n_\forall$ be the number of universally quantified variables in E'. Construct a formula $E'' = \forall z_1 \forall z_2 \cdots \forall z_k \exists y_1 \cdots \exists y_n Q_1x_1 \cdots Q_nx_nF''$ from E', where F'' is formed from F' by replacing every clause $l_{j1} \vee l_{j2} \vee l_{j3}$ by the set of clauses $\{l_{j1} \vee l_{j2} \vee l_{j3} \vee z_i \vee \bar{z}_i | 1 \leqq i \leqq k\}$. Here k is a suitably large integer whose value we shall select later.

Let $s = 3k + 6n + 5$. Construct a graph G corresponding to the new formula as in § 2, using a pyramid of size six to represent each clause. Since E'' is true, G can be pebbled with s pebbles. If a double false substitution is made for some variable z_i in E'', the resulting formula is true if and only if the original formula E is true. Thus if E is false, pebbling G requires $\Omega(mk2^{k+n_\forall})$ time. If E is true, G can be pebbled in $O((mk + (2n + k)^3)2^{n_\forall})$ time by selecting the double false assignment for every variable z_i. Thus if k is sufficiently large ($k = m$ suffices for large m), there is a time $t(m)$ such that G can be pebbled with s pebbles in time $t(m)$ if and only if E is true.

Acknowledgment. We would like to thank Michael Loui for suggesting the existential quantifier block in Fig. 4, which considerably simplified our original construction. A preliminary version of this paper was presented at the Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, Georgia, 1979.

REFERENCES

[1] S. COOK, *An observation on time-storage trade off,* Proc. Fifth Annual ACM Symp. on Theory of Computing, Austin, TX, 1973, pp. 29–33.

[2] P. VAN EMDE BOAS AND J. VAN LEEUWEN, *Move rules and trade-offs in the pebble game,* Proc. Fourth GI Conf. on Theoretical Computer Science, Springer Lecture Notes on Computer Science, 67, 1979, pp. 101–112.

[3] S. EVEN AND R. E. TARJAN, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.

[4] 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. Nineteenth Annual Symp. on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 55-64.

[5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[6] J. E. HOPCROFT, W. PAUL AND L. VALIANT, *On time versus space*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.

[7] D. LICHTENSTEIN AND M. SIPSER, *GO is P-space hard*, Proc. Nineteenth Annual Symp. on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 48–54.

[8] A. LINGAS, *A PSPACE complete problem related to a pebble game*, Automata, Languages, and Programming, Fifth Colloquium, Springer Lecture Notes on Computer Science 62, 1978, pp. 300–321.

[9] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Record of Project MAC Conference on Concurrent Systems and Parallel Computation, Cambridge, MA, 1970, pp. 119–128.

[10] W. PAUL AND R. E. TARJAN, *Time-space trade-offs in a pebble game*, Acta Informat., 10 (1978), pp. 111–115.

[11] N. PIPPENGER, *A time-space trade-off*, J. Assoc. Comput. Mach., 25 (1978), pp. 509–515.

[12] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[13] T. J. SCHAEFER, *Complexity of decision problems based on finite two-person perfect-information games*, Proc. Eighth Annual ACM Symp. on Theory of Computing, Hershey, PA, 1976, pp. 41–49.

[14] R. SETHI, *Complete register allocation problems*, this Journal, 4 (1975), pp. 226–248.

[15] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: preliminary report*, Proc. Fifth Annual ACM Symp. on Theory of Computing, Austin, TX, 1973, pp. 1–9.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0006 $01.00/0

VECTOR ITERATION IN POINTED ITERATIVE THEORIES*

STEPHEN L. BLOOM†, CALVIN C. ELGOT‡ AND JESSE B. WRIGHT‡

Abstract. This paper is a sequel to a previous paper (S. L. Bloom, C. C. Elgot and J. B. Wright, Solutions of the iteration equation and extensions of the scalar iteration operations, SIAM J. Comput., 9 (1980), pp. 25–45. In that paper it was proved that for each morphism $\perp : 1 \to 0$ in an iterative theory J there is exactly one extension of the scalar iteration operation in J to all scalar morphisms such that $I_1^\dagger = \perp$ and all scalar iterative identities remain valid. In this paper the scalar iteration operation in the pointed iterative theory $(J, \perp)$ is extended to vector morphisms while preserving all the old identities.

The main result shows that the vector iterate $g^\dagger$ in $(J, \perp)$ satisfies the equation $g^\dagger = (g_\perp)^\dagger$, where $g_\perp$ is a nonsingular morphism simply related to g (so that $(g_\perp)^\dagger$ is the unique solution of the iteration equation for $g_\perp$).

In the case that $J = \Gamma\mathrm{Tr}$, the iterative theory of Γ-trees, it is shown that the vector iterate $g^\dagger$ in $(J, \perp)$ is a metric limit of "modified powers" of g.

Key words. Algebraic iterative theory, computation semantics iteration, flowcharts

Introduction. This paper is a sequel to [3]. In that paper it was shown that each morphism $\perp : 1 \to 0$ in an iterative theory J determines uniquely an extension of the scalar iterative operation in J to all scalar morphisms in such a way that $I_1^\dagger = \perp$ and all scalar iterative identities remain valid. In the current paper the scalar iteration operation is extended to all vector morphisms while preserving all old identities.

In § 2, this extension is defined in an inductive way. It is shown (Theorem 2.4) that four identities characterize this extension. Our main result (Theorem 2.7) provides an explicit description of extended iteration in any pointed iterative theory $(J, \perp)$. The special case of the iterative theory of labeled trees $(\Gamma\mathrm{Tr}, \perp)$ [6], [1] is considered in § 5. The Γ-trees $n \to p$ form a complete metric space in a natural way and we show that the vector iterate $g^\dagger$ of the Γ-tree $g : n \to p + n$ is a metric limit of the sequence $g \circ (I_p \oplus \perp_n), g^2 \circ (I_p \oplus \perp_n), g^3 \circ (I_p \oplus \perp_n) \cdots$.

Section 3 is devoted to showing that extended iteration satisfies all iterative theory identities. To this end the category of "iteration theories" is introduced and these theories are characterized in five ways (Theorem 3.3).

In section 4, some examples are given relating extended iteration to ordered algebraic theories [1]. Two examples are given of a choice for $\perp : 1 \to 0$ in the theory of Γ-trees so that in the resulting iteration theory $(\Gamma\mathrm{Tr}, \perp)$ there is some partial ordering of the trees such that the value of extended iteration $g^\dagger$ is the least solution of the iteration equation for g. For other choices of $\perp : 1 \to 0$ in $\Gamma\mathrm{Tr}$, no such ordering need exist (see Theorem 4.4).

1. Preliminaries.

While familiarity with [3] would be very helpful, we will not need to depend on it heavily except in the appendix. However the reader should have some knowledge of algebraic and iterative theories as defined in [5]; the elementary properties of these theories are summarized in § 1.6 of [3].

For the reader's convenience we will recall here some facts, definitions and notation from [3]. An *algebraic theory J* is a category whose objects are the nonnegative integers, having for each $n \geqq 0$, n "distinguished morphisms" $i : 1 \to n$, $i \in [n]$ (where $[n] = \{1, 2, \cdots, n\}$; $[0] = \phi$) such that:

* Received by the editors November 28, 1978, and in final revised form June 30, 1979.

† Department of Pure and Applied Mathematics, Stevens Institute of Technology, Hoboken, New Jersey 07030. During the preparation of this paper this author was partially supported by the National Science Foundation under Grant MGS-78-00882.

‡ Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

for each family f_i: $1 \to p$, $i \in [n]$ $n \geqq 0$, of morphisms in J there is a unique morphism f: $n \to p$ such that for each $i \in [n]$, f_i is the composition of i: $1 \to n$ and f: $n \to p$.

In the case $n = 0$, this condition amounts to the requirement that there is a unique morphism O_p: $0 \to p$, for each $p \geqq 0$. The morphism f is the *source tupling* of the morphisms f_i and is denoted $(f_1, \cdots, f_n)$. All morphisms $n \to p$, $n, p \geqq 0$ formed by source tupling the distinguished morphisms are *base*. When the distinguished morphisms are distinct (which is the case in every algebraic theory but the two "trivial theories") the base morphisms are in bijective correspondence with the functions $[n] \to [p]$: the function f: $[n] \to [p]$ corresponds to the source tupling $(f(1), \cdots, f(n))$, where $f(i)$: $1 \to p$ is distinguished. The base morphism corresponding to the identity function on $[n]$ is denoted I_n.

The *composition* of f: $n \to p$ and g: $p \to q$ is denoted either $f \circ g$: $n \to q$ or $n \overset{f}{-} p \overset{g}{\to} q$ (with an arrowhead missing). Source tupling permits defining *source pairing* of two morphisms with a common target. If f_i: $n_i \to p$, $i = 1, 2$, then the *source pairing* (f_1, f_2): $n_1 + n_2 \to p$ satisfies $i \circ (f_1, f_2) = i \circ f_1$, if $i \in [n]$; $i \circ (f_1, f_2) = j \circ f_2$ if $i = n_1 + j$, $j \in [n_2]$. The "circle-sum" $f_1 \oplus f_2$: $n_1 + n_2 \to p_1 + p_2$ of f_i: $n_i \to p_i$, $i \in [2]$, is the source pairing $(f_1 \circ \kappa, f_2 \circ \lambda)$, where $\kappa : p_1 \to p_1 + p_2$ and $\lambda : p_2 \to p_1 + p_2$ are base morphisms corresponding to the inclusion and translated inclusion functions. (See [3, § 1.6] for a list of some elementary properties of these operations.)

A morphism g: $1 \to p$ in an algebraic theory J is *ideal* if for any h: $p \to q$, $g \circ h$: $1 \to q$ is not distinguished; J itself is *ideal* if every nondistinguished morphism $1 \to p$ in J is ideal.

An *iterative theory* is an ideal theory J such that for any ideal morphism g: $1 \to p + 1$, the *iteration equation* for g (in the variable ξ: $1 \to p$)

$$\xi = g \circ (I_p, \xi) \tag{1.1}$$

has a unique solution. In [2] it was shown that in an iterative theory, when $n > 1$ and the morphism g: $n \to p + n$ has the property that $i \circ g$ is ideal for each $i \in [n]$ (we say "g is ideal") then the iteration equation (1.1) for g (now in the variable ξ: $n \to p$) also has a unique solution, denoted $g^\dagger$. Thus

$$g^\dagger = g \circ (I_p, g^\dagger).$$

The *powers* g^k of a morphism g: $n \to p + n$ in an algebraic theory J are defined inductively:

$$g^0 = O_p \oplus I_n : n \to p + n,$$
$$g^{k+1} = g^k \circ (I_p \oplus O_n, g).$$

Two useful facts about g^k are:

$$(I_p \oplus O_n, g)^k = (I_p \oplus O_n, g^k), \text{ all } k \geqq 0;$$
$$\text{if } \quad \xi: n \to p \text{ satisfies } \xi = g \circ (I_p, \xi),$$
$$\text{then } \quad \xi = g^k \circ (I_p, \xi), \text{ all } k \geqq 0.$$

Let g: $n \to p + n$ be a morphism in an algebraic theory. The morphism g is *power ideal* if for some $k \geqq 1$, g^k is ideal; g is *power successful* if g is base and for some $k \geqq 1$, g^k may be written as $n \overset{a}{-} p \xrightarrow{I_p \oplus O_n} p + n$, for some base morphism a: $n \to p$. A number $i \in [n]$ is a *singular position* of g if for each $k \geqq 1$ there is some number u_{k+1} in $[n]$ such that $i \circ g = O_p \oplus u_{k+1}$. The set of all singular positions of g is denoted K_g. If K_g is empty,

g is a *nonsingular* morphism. In [3, **2.17**] it was shown that in an iterative theory, the iteration equation for a nonsingular morphism g has a unique solution.

A morphism $g: n \to p+n$ in an algebraic theory J is a "*mkl*-morphism" [3, **2.7**] if $m+k+l=n$ and there are base morphisms $a: m \to p+m$, $b: k \to k$ and a power ideal morphism $h: l \to p+n$ such that

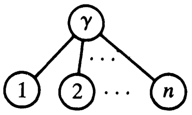
$$g = (a \oplus b \oplus O_l, h), \quad \text{and} \quad a^m = a^\dagger \oplus O_m, \quad \text{where } a^\dagger: m \to p$$

and in fact $a^\dagger$ is the unique solution of the iteration equation for a. The properties of nonsingular, ideal, power ideal and power successful morphisms are discussed in detail in [3].

In §§ 3–5, Γ denotes a ranked alphabet; i.e. Γ is the union $\bigcup_{n=0}^{\infty} \Gamma_n$ of the pairwise disjoint sets Γ_n. The iterative theory of all Γ-trees, mentioned in the introduction, was studied in [6]. (Algebraic theories of trees were studied also in [1].) The subtheory Γtr of ΓTr consists of those Γ-trees having (up to isomorphism) a finite number of descendancy trees. In [6] it was shown that Γtr is the iterative theory freely generated by Γ: i.e. for any iterative theory J and any function $F: \Gamma \to J$ taking $\gamma \in \Gamma_n$ to an ideal morphism $\gamma F: 1 \to n$ in J there is a unique "theory morphism" $\mathbf{F}'$: $\Gamma\text{tr} \to J$ such that the diagram

$$\begin{tikzcd}
\Gamma \arrow[r, "\iota"] \arrow[dr, "\mathbf{F}"'] & \Gamma\text{tr} \arrow[d, "\mathbf{F}'"] \\
 & J
\end{tikzcd}$$

commutes, where if $\gamma \in \Gamma_n$, $\gamma\iota$ is the tree having a root labeled γ and n leaves; the ith leaf of $\gamma\iota$ is labeled i, as indicated by this figure.



If J and J' are algebraic theories, a *theory morphism* $\mathbf{F}: J \to J'$ is a family of functions taking each morphism f: $n \to p$ in J to a morphism $f\mathbf{F}$: $n \to p$ in J' such that $(f \circ g)\mathbf{F} = f\mathbf{F} \circ g\mathbf{F}$ and $i\mathbf{F} = i$ for all composible f, g in J and distinguished morphisms $i: 1 \to n$.

2. The uniqueness and explicit description of extended iteration. The first task of this section is to define an extension of the iteration operation in an iterative theory J in such a way that the "identities" or "equations" which hold for the restricted operation (defined only on those morphisms whose iteration equation has a unique solution) will continue to hold. The definition of this operation is inductive. In Def. 2.7 an explicit description of this operation is given.

Two equations valid in all iterative theories are given by (2.1) and (2.2).

$$[O_m \oplus g]^\dagger = O_m \oplus g^\dagger \tag{2.1}$$

where g: $n \to p+n$ is ideal. This identity may be verified by showing that the righthand side is a solution of the iteration equation for $O_m \oplus g$.

The following equation, established in [2], will be referred to as the "pairing identity".

$$(f_1, f_2)^\dagger = (g^\dagger, f_2^\ddagger \circ (I_p, g^\dagger)) \tag{2.2}$$

where f_1: $n \to p+n+1$ and f_2: $1 \to p+n+1$ are ideal, and where g is defined to be the composition

$$(2.3) \qquad g\colon n \overset{f_1}{-} p+n+1 \xrightarrow{(I_{p+n}, f_2^\dagger)} p+n.$$

We will repeat the argument of [2] below (in 2.5) which shows that the pairing identity holds in all iterative theories.

A special case of the equation (2.1) occurs when $n = 1$; i.e. $g\colon 1 \to p+1$. In this case, if g is base and $g = j\colon 1 \to p+1$, where $j < p+1$, then $g^\dagger$ is meaningful since the unique solution of the iteration equation for g is $j\colon 1 \to p$ (note the change of target). Of course if g is ideal, $g^\dagger$ is meaningful. But if $g = O_p \oplus I_1\colon 1 \to p+1$, every morphism $1 \to p$ is a solution of the iteration equation for g. The requirement that (2.1) remain valid for the extended operation (also denoted †) means that extended † should satisfy

$$(2.1') \qquad [O_m \oplus I_1]^\dagger = O_m \oplus I_1^\dagger.$$

These considerations yield the uniqueness part of the following theorem.

THEOREM 2.4. *Let $\perp\colon 1 \to 0$ be a morphism in the iterative theory J. There is one and only one operation $g \mapsto g^\dagger$, defined for all $g\colon n \to p+n$ (all $n, p \geqq 0$, yielding a morphism $g^\dagger\colon n \to p$) which satisfies* (2.1'), (2.2) *as well as*

$$(2.4.1) \qquad I_1^\dagger = \perp,$$

$$(2.4.2) \qquad g^\dagger = g \circ (I_p, g^\dagger),$$

for all scalar $g\colon 1 \to p+1$, all $p \geqq 0$.

Proof. For scalar g which is ideal or of the form $j \oplus O_1$, where $j\colon 1 \to p$ is base, we define $g^\dagger$ by the requirement that (2.4.2) be satisfied. (Thus, on scalar ideal g the extended and unextended iteration operations agree.) For scalar g of the form $O_p \oplus I_1$, we define $g^\dagger$ by (2.1') and (2.4.1).

Vector iteration in $(J, I_1^\dagger = \perp)$ is defined inductively. Assume that for all p and all $h\colon k \to p+k$, $k \leqq n$, $h^\dagger$ is defined. If (as in (2.2)) $f = (f_1, f_2)\colon n+1 \to p+n+1$, we *define* $f^\dagger$ by (2.2).

It is then obvious that (2.2) *holds for all* $n, p \geqq 0$. Note that when $n = 0$, (2.2) becomes:

$$(O_{p+1}, f_2)^\dagger = (O_p, f_2^\dagger \circ (I_p, O_p))$$

which is trivially true.

From the above theorem, we obtain

COROLLARY 2.5. *In $(J, I_1^\dagger = \perp)$, extended vector iteration satisfies* (*and is determined by*) *the identities* (2.1'), (2.2), (2.4.1) *and* (2.4.2).

We now repeat the argument of [2] to show that the following holds.

THEOREM. *In $(J, I_1^\dagger = \perp)$ the condition* (2.4.2) *is valid for all vectors g.*

Proof. The proof is by induction. Assume that (2.4.2) holds for all $h\colon k \to p+k$, all $k \leqq n$. Let $f = (f_1, f_2)\colon n+1 \to p+n+1$, where $f_1\colon n \to p+n+1$, $f_2\colon 1 \to p+n+1$. Define $g\colon n \to p+n$ by (2.3). We then substitute

$$\xi = (g^\dagger, f_2^\dagger \circ (I_p, g^\dagger))$$

in the right-hand side of the iteration equation for f yielding:

$$\begin{aligned} f\circ(I_p, g^\dagger, f_2^\dagger\circ(I_p, g^\dagger)) &= (f_1, f_2)\circ(I_{p+n}, f_2^\dagger)\circ(I_p, g^\dagger) && \text{by (1.6.4) [3],}\\ &= (f_1\circ(I_{p+n}, f_2^\dagger)\circ(I_p, g^\dagger), f_2^\dagger\circ(I_p, g^\dagger)) && \text{by [3, (1.6.4)]}\\ & && \text{and } f_2^\dagger = f_2\circ(I_{p+n}, f_2^\dagger),\\ &= (g^\dagger, f_2^\dagger\circ(I_p, g^\dagger)) && \text{by (2.3) and } g^\dagger = g\circ(I_p, g^\dagger). \end{aligned}$$

We record here without proof the following miscellaneous fact.

PROPOSITION 2.6. *With $f = (f_1, f_2)$, g as in* (2.2) *and* (2.3): *f is nonsingular iff both g and f_2 are nonsingular.*

We turn now to the second task of this section and one of the main points of the paper—the explicit description of the extended iteration operation. Recall from the introduction the definition of the set K_g of the singular positions of the morphism $g: n \to p+n$.

DEFINITION 2.7. Let $g: n\to p+n$ be a morphism in $(J, I_1^\dagger = \perp)$. The morphism $g_\perp: n\to p+n$ is defined by the following requirements:

$$\text{(2.7.1)}\qquad i\circ g_\perp = \perp\circ O_{p+n},\quad \text{if } i\in K_g;$$

$$\text{(2.7.2)}\qquad i\circ g_\perp = i\circ g,\quad \text{if } i\notin K_g.$$

LEMMA. *If i is a power ideal* (*respectively, power successful*) *position of g, then i is a power ideal* (*respectively, power successful*) *position of $g_\perp$.*

Proof. Suppose that i is a power ideal position of g, so that for some k, $i\circ g^k$ is ideal. Let r be the least integer such that $i\circ g^{r+1}$ is ideal. If $r=0$, then $i\circ g_\perp = i\circ g$ is ideal. If $r>0$, then $i\circ g^r = p+i'$, for some $i'\in[n]$, and $i'\circ g = i\circ g^{r+1}$ is ideal. Then for all $k\leqq r$, $i\circ(g_\perp)^k = i\circ g^k$. In particular, $i\circ(g_\perp)^r = p+i'$ so that $i\circ(g_\perp)^{r+1} = i'\circ g_\perp = i'\circ g$ is ideal. Thus i is a power ideal position of $g_\perp$.

The proof in the case that i is a power successful position of g is similar.

We may now give an explicit description of $g^\dagger$ in $(J, I_1^\dagger = \perp)$.

THEOREM. *In $(J, I_1^\dagger = \perp)$, for any $g: n\to p+n$, $g_\perp$ is nonsingular and*

$$\text{(2.7.3)}\qquad g^\dagger = (g_\perp)^\dagger.$$

Proof. By (2.7.1) and the Lemma, $g_\perp$ is nonsingular. Thus, by [3, **2.17**] the iteration equation for $g_\perp$ has a unique solution, and we need verify only that $g^\dagger$ is one such solution.

By the Theorem in § 2.5,

$$\text{(2.7.5)}\qquad g^\dagger = g\circ(I_p, g^\dagger).$$

Thus from (2.7.2) if $i\notin K_g$,

$$\text{(2.7.6)}\qquad i\circ g^\dagger = i\circ g_\perp\circ(I_p, g^\dagger),$$

while if $i\in K_g$, by (2.7.1) we have

$$\text{(2.7.7)}\qquad i\circ g_\perp\circ(I_p, g^\dagger) = \perp\circ O_p.$$

We now need the following fact, proved in the Appendix.

LEMMA. *If $i\in K_g$, $i\circ g^\dagger = \perp\circ O_p$.*

Applying the Lemma, we have

$$i\circ g^\dagger = i\circ g_\perp\circ(I_p, g^\dagger),\quad \text{for all } i\in[n].$$

Thus $g^\dagger = g_\perp\circ(I_p, g^\dagger)$, and, since $g_\perp$ is nonsingular, $g^\dagger = (g_\perp)^\dagger$, by [3, **2.19**].

COROLLARY 2.8. *If $\bar{g}$ is a "mkl morphism", $\bar{g}=(a\oplus b\oplus O_l, h)$, (defined in* § 1) *then*

$$\bar{g}=(a^{\dagger}\oplus\perp_k, h\circ(I_p, a^{\dagger}\oplus\perp_k)) \tag{2.8.1}$$

where $\perp_k=(\overbrace{\perp,\cdots,\perp}^{k})\colon k\to 0.$

Proof. The proof follows easily from the following fact, used in [3, **2.12**]. If $g\colon n\to p+n$ is writable in the form

$$(a\oplus c\oplus O_l, h) \tag{2.8.2}$$

where $m+k+l=n$, $a\colon m\to p+m$, $c\colon k\to k$ and $h\colon l\to p+n$ and $\xi=(\xi_1,\xi_2,\xi_3)$ is any solution of the iteration equation for g, where $\xi_1\colon m\to p$, $\xi_2\colon k\to p$, $\xi_3\colon l\to p$, then

$$\xi_1=a\circ(I_p,\xi_1) \tag{2.8.3}$$

$$\xi_2=c\circ\xi_2 \tag{2.8.4}$$

$$\xi_3=h\circ(I_p,\xi_1,\xi_2,\xi_3)=h\circ[(I_p,\xi_1,\xi_2)\oplus I_l]\circ(I_p,\xi_3). \tag{2.8.5}$$

Now in the case $\bar{g}$ is a *mkl*-morphism, $\bar{g}_{\perp}$ is writable in the form (2.8.2) where $c\colon k\xrightarrow{\text{base}}1\xrightarrow{\perp}0\xrightarrow{O_k}k$, and a and h are as in g. Then, by (2.8.3) and the hypothesis on a, $\xi_1=a^{\dagger}$; by (2.8.4), $\xi_2=\perp_k\circ O_p$. Lastly, (2.8.5) forces ξ_3 to be the unique solution to the iteration equation of $h\circ(I_p, a^{\dagger}\oplus\perp_k)\oplus I_l$, which is $h^{\dagger}\circ(I_p, a^{\dagger}\oplus\perp_k)$. The corollary now follows from Theorem 2.7.

3. Iteration theories. By a "preiteration theory" we mean an algebraic theory P (not necessarily an ideal theory) equipped with an operation † ("iteration") which for each $n, p\geqq 0$ takes a morphism $g\colon n\to p+n$ in P to a morphism $g^{\dagger}\colon n\to p$ in P (subject to no conditions at all). Of course, strictly speaking † is a family of operations indexed by pairs n, p of nonnegative integers. In this section we will provide five equivalent conditions on a preiteration theory P which ensure that P will satisfy all identities of iterative theories. In this section the "pairing identity" (2.2) plays a major role.

PROPOSITION 3.1. *Let P and Q be preiteration theories which satisfy the pairing identity (i.e. for all f, (2.2) holds). If $\mathbf{G}\colon P\to Q$ is a theory morphism which preserves scalar iteration (i.e. $f^{\dagger}\mathbf{G}=(f\mathbf{G})^{\dagger}$, all $f\colon 1\to p+1$ in P) then $\mathbf{G}$ preserves vector iteration.*

The easy proof is by induction.

Remark 3.2. If J is an iterative theory and $\perp\colon 1\to 0$ is in J, then the preiteration theory $(J, I_1^{\dagger}=\perp)$ satisfies the pairing identity for all $f\colon n+1\to p+n+1$, (all $n, p\geqq 0$) in J (see the proof of Theorem 2.4).

Before stating the main theorem of this section we introduce some notation. Γ will always denote a ranked set, and $\Gamma_{\square}$ is the ranked set obtained from Γ by adjoining a "new" element $\square$ to Γ_0. Thus

$$(\Gamma_{\square})_0=\Gamma_0\cup\{\square\};\qquad(\Gamma_{\square})_n=\Gamma_n,\qquad n>0.$$

Recall that $\Gamma\mathrm{tr}$ is the iterative theory (of trees) freely generated by Γ (see [6]).

THEOREM 3.3. *The following five properties of a preiteration theory P are equivalent.*

(3.3.1) *P satisfies the pairing identity* and *for any Γ, and any function $\mathbf{F}\colon\Gamma\to P$*[1] *there is an extension of $\mathbf{F}$ to a theory morphism*

$$\mathbf{F}'\colon\Gamma\mathrm{tr}\to P$$

[1] A function from Γ to P will be assumed to be "rank preserving"; i.e. $\gamma\in\Gamma_n\mapsto\gamma\mathbf{F}\colon 1\to n$ in P.

which preserves scalar iteration applied to ideal *morphisms* (*i.e. if* $g\colon 1 \to p+1$ *is ideal,* $g^{\dagger}\mathbf{F}' = (g\mathbf{F}')^{\dagger}$).

(3.3.2) *For any* Γ *and any function* $\mathbf{F}\colon \Gamma \to P$ *there is an extension of* $\mathbf{F}$ *to a theory morphism*

$$\mathbf{F}'\colon (\Gamma_{\Box}\mathrm{tr}, I_1^{\dagger} = \Box) \to P$$

which preserves extended vector iteration (*i.e. for all* $g\colon n \to p+n$, $g^{\dagger}\mathbf{F}' = (g\mathbf{F}')^{\dagger}$).

(3.3.3) *For any* Γ *and any function* $\mathbf{F}\colon \Gamma \to P$ *there is an extension of* $\mathbf{F}$ *to a theory morphism*

$$\mathbf{F}'\colon \Gamma\mathrm{tr} \to P$$

which preserves vector iteration applied to ideal morphisms.

(3.3.4) *There is an iterative theory J and a theory morphism* $\mathbf{H}\colon J \to P$ *such that every morphism in P is the image of an ideal morphism in J and such that* $g^{\dagger}\mathbf{H} = (g\mathbf{H})^{\dagger}$ *for all* ideal *morphisms* $g\colon n \to p+n$.

(3.3.4′) *The same as* (3.3.4) *except that each morphism in P is the image of some* (*not necessarily ideal*) *morphism in J.*

Proof. (3.3.1) $\Rightarrow$ (3.3.2). Let P satisfy (3.3.1). We first show that P will satisfy some identities in addition to the pairing identity. Let Γ be a ranked set so large that there is a surjective function $\mathbf{F}\colon \Gamma \to P$. For any morphisms $h\colon 1 \to q+1$, $\beta\colon q \to s$ in P, let $\bar{h}$ and $\bar{\beta}$ be ideal morphisms in Γtr such that $\bar{h}\mathbf{F}' = h$, $\bar{\beta}\mathbf{F}' = \beta$, where $\mathbf{F}'\colon \Gamma\mathrm{tr} \to P$ is the extension of $\mathbf{F}$ guaranteed by (3.3.1). Since it is easy to show that

$$[\bar{h} \circ (\bar{\beta} \oplus I_1)]^{\dagger} = \bar{h}^{\dagger} \circ \bar{\beta}$$

in Γtr, and since $\mathbf{F}'$ is a theory morphism preserving scalar iteration of ideal morphisms,

$$[h \circ (\beta \oplus I_1)]^{\dagger} = h^{\dagger} \circ \beta \tag{3.3.5}$$

in P.

Note that in the case $q = 0$ and $h = I_1$, (3.3.5) specializes to

$$[O_s \oplus I_1]^{\dagger} = I_1^{\dagger} \circ O_s = O_s \oplus I_1^{\dagger}. \tag{3.3.6}$$

In the same way one may verify that for $j \in [p]$ the identity

$$[j \oplus O_1]^{\dagger} = j \tag{3.3.7}$$

is valid in P, where $j\colon 1 \to p$ is base.

We may now prove (3.3.2). Let Γ and $\mathbf{F}\colon \Gamma \to p$ be fixed. We must show there is an extension of $\mathbf{F}$ to a theory morphism

$$\mathbf{F}'\colon (\Gamma_{\Box}\mathrm{tr}, I_1^{\dagger} = \Box) \to P$$

which preserves extended vector iteration.

First we extend $\mathbf{F}$ to $\Gamma_{\Box}$ by defining $\Box\mathbf{F} = I_1^{\dagger}$ in P. Then by assumption, $\mathbf{F}\colon \Gamma_{\Box} \to P$ extends to a theory morphism $\mathbf{F}'\colon \Gamma_{\Box}\mathrm{tr} \to P$ which preserves scalar iteration applied to *ideal* morphisms. We now show that $\mathbf{F}'$ also preserves scalar iteration on *base* morphisms. If $f\colon 1 \to p+1$ is base, either $f = j \oplus O_1$, some $j\colon 1 \to p$ or $f = O_p \oplus I_1$. In the first case $f^{\dagger} = j$ in $\Gamma_{\Box}$tr and thus $f^{\dagger}\mathbf{F}' = j\mathbf{F}' = j = (f\mathbf{F}')^{\dagger}$, by (3.3.7) since $\mathbf{F}'$ is a theory morphism. Similarly, if $f = O_p \oplus I_1$, $f^{\dagger} = O_p \oplus \Box$ in $(\Gamma_{\Box}\mathrm{tr}, I_1^{\dagger} = \Box)$, so $f^{\dagger}\mathbf{F}' = O_p \oplus \Box\mathbf{F}' = O_p \oplus I_1^{\dagger} = (f\mathbf{F})^{\dagger}$, by (3.3.6). Thus $\mathbf{F}'$ preserves scalar iteration. Thus by Proposition 3.1 $\mathbf{F}'$ preserves extended vector iteration, completing the proof.

Proof (3.3.2) $\Rightarrow$ (3.3.3). The proof is obvious.

Proof (3.3.3) $\Rightarrow$ (3.3.1). The fact that the pairing identity is valid in P is proved in the same way it was shown that (3.3.1) implies that the identity (3.3.5) holds in P. The rest of the statement follows trivially.

Proof (3.3.3) $\Rightarrow$ (3.3.4). The proof is obvious.

Proof (3.3.4) $\Rightarrow$ (3.3.3). Let $\mathbf{H}: J \to P$ be as in the statement of (3.3.4) and let $\mathbf{F}: \Gamma \to P$ be an arbitrary function. There is a function $\mathbf{G}: \Gamma \to J$ such that for each $\gamma \in \Gamma_n$, $\gamma\mathbf{G}: 1 \to n$ is an ideal morphism in J and $\gamma\mathbf{GH} = \gamma\mathbf{F}$. By [6, 4.1.2] $\mathbf{G}$ has an extension $\mathbf{G}': \Gamma\mathrm{tr} \to J$ to an ideal theory morphism, which necessarily preserves vector iteration applied to ideal morphisms. We may *define* $\mathbf{F}'$ to be the composition

$$\mathbf{F}': \Gamma\mathrm{tr} \xrightarrow{\mathbf{G}'} J \xrightarrow{\mathbf{H}} P:$$

Since both $\mathbf{G}'$ and $\mathbf{H}$ preserve vector iteration applied to ideal morphisms, so does $\mathbf{F}'$.

Proof (3.3.4) $\Rightarrow$ (3.3.4′). The proof is obvious.

Proof (3.3.4′) $\Rightarrow$ (3.3.4). Assume that $H: J \to P$ is a surjective theory morphism which preserves vector iteration applied to ideal morphisms. Without loss of generality we may assume that J is $\Gamma\mathrm{tr}$, for some ranked set Γ. Let Γ_Δ be the ranked set obtained from Γ by adding a new element Δ to Γ_1, so that $(\Gamma_\Delta)_1 = \Gamma_1 \cup \{\Delta\}$; $(\Gamma_\Delta)_n = \Gamma_n$, $n \neq 1$. Let $\perp: 1 \to 0$ be a morphism such that $\perp H = I_1^\dagger$ in P. We now apply the

Δ-LEMMA [3, **3.6**]. *There is a unique theory morphism* $\Phi: \Gamma_\Delta\mathrm{tr} \to \Gamma\mathrm{tr}$ *such that* (a) $\Delta\Phi = I_1$; (b) $\Delta^\dagger\Phi = \perp$; (c) $\gamma\Phi = \gamma$, *all* $\gamma \in \Gamma$; (d) *if* $f\Phi$ *is base, say* $j: 1 \to p+1$, *then* $f = \Delta^r \circ j$, *for some* $r \geqq 0$.

It is easy to show, using Prop. 3.1 and parts (b) and (d) of the Δ-Lemma that $g^\dagger\Phi = (g\Phi)^\dagger$ for all ideal $g: n \to p+n$ in $\Gamma_\Delta\mathrm{tr}$ (where $I_1^\dagger = \perp$ in $\Gamma\mathrm{tr}$). Let H_Δ be the composition

$$H_\Delta: \Gamma_\Delta\mathrm{tr} \xrightarrow{\Phi} \Gamma\mathrm{tr} \xrightarrow{H} P.$$

Then H_Δ is a theory morphism which preserves vector iteration on ideal morphisms. Lastly, every morphism in P is the H_Δ-image of an ideal morphism in $\Gamma_\Delta\mathrm{tr}$. Indeed, if $f: 1 \to p$ in P is not base, $f = gH$ for some ideal $g: 1 \to p$ in $\Gamma\mathrm{tr}$; hence $f = gH_\Delta$ (by part (c) of the Δ-Lemma). However if $f: 1 \to p$ is base, then $f = I_1 \circ f = (\Delta \circ f)H_\Delta$. This completes the proof of the theorem.

Remark. In (3.3.1), (3.3.2) and (3.3.3) the extension $\mathbf{F}'$ of $\mathbf{F}$ is necessarily unique, by [6, 2.5.1].

DEFINITION 3.4. An *iteration theory* is a preiteration theory which satisfies any (and hence all) of the properties of Theorem 3.3.

PROPOSITION 3.5. *Let* $\perp: 1 \to 0$ *be a morphism in the iterative theory* J. *Then the preiteration theory* $(J, I_1^\dagger = \perp)$ *defined in* 2.5 *is an iteration theory.*

Proof. The Universality Theorem (**3.7** in [3]) implies that for any function $\mathbf{F}: \Gamma \to J$ there is an extension of $\mathbf{F}$ to a theory morphism

$$\mathbf{F}': (\Gamma_\Box\mathrm{tr}, I_1^\dagger = \Box) \to (J, I_1^\dagger = \perp)$$

such that $\Box\mathbf{F}' = \perp$ and such that $\mathbf{F}'$ preserves (extended) scalar iteration. Both preiteration theories $(\Gamma_\Box\mathrm{tr}, I_1^\dagger = \Box)$ and $(J, I_1^\dagger = \perp)$ satisfy the pairing identity, so that by 3.1, $\mathbf{F}'$ preserves extended vector iteration. This shows that $(J, I_1^\dagger = \perp)$ has property (3.3.2), and is thus an iteration theory.

COROLLARY. *The iteration theory* $(\Gamma_\Box\mathrm{tr}, I_1^\dagger = \Box)$ *is freely generated by* Γ *in the class of iteration theories.*

Proof. This is a restatement of property (3.3.2). (Recall the Remark preceding Def. 3.4.)

3.6. As has already been indicated, the intuitive meaning of "iteration theory" is a preiteration theory P which satisfies (according to (3.3.2)) all identities, e.g.

$$g^{\dagger} = g \circ (I_p, g^{\dagger})$$

which are (meaningful and) valid in $(\Gamma_{\square}\mathrm{tr}, I_1^{\dagger} = \square)$. By Theorem 3.3 (3.3.3), a similar statement may be made with "$\Gamma\mathrm{tr}$" in place of "$(\Gamma_{\square}\mathrm{tr}, I_1^{\dagger} = \square)$", but the class of meaningful identities is reduced and "valid" must be properly understood. According to Theorem 3.3 (3.3.1), it is enough to know P satisfies the pairing identity and all scalar iteration identities of $\Gamma\mathrm{tr}$ to conclude that P is an iteration theory.

We state, for emphasis, an immediate corollary of this discussion, Theorem 3.3 and Proposition 3.5:

COROLLARY 3.6.1. *If* $\perp : 1 \to 0$ *is any morphism in the iterative theory* J, *every valid iterative theory identity is true in* $(J, I_1^{\dagger} = \perp)$.

3.7. Although the class of iterative theories is not closed under products (since the product of two ideal theories is not ideal), the class of iteration theories is, i.e.

if P *and* Q *are iteration theories, so is* $P \times Q$.

For example, the pairing identity is valid in $P \times Q$, since otherwise it would fail to hold in either P or Q.

Example 3.8. An important example of an iteration theory which is not even ideal is the theory $[X]$ where X is a nonempty set. (This theory is denoted $[X, Q]$ in [5, p. 189], where Q is a singleton set.) A morphism $f\colon n \to p$ in $[X]$ is a partial function $f\colon X \times [n] \to X \times [p]$. In $[X]$, if $f\colon n \to p + n$, the partial function $f^{\dagger}$ defined below is the least (in the sense of set inclusion of their graphs) solution ξ of the iteration equation for f (viz. $\xi = f \circ (I_p, \xi)$).

3.8.1. $f^{\dagger}\colon X \times [n] \to X \times [p]$ is the partial function defined by: $xif = x'i'$ if there is a sequence $x_0 i_0 x_1 i_1 \cdots x_m i_m$, $m \geqq 0$ such that $x_0 = x$, $i_0 = i$, and for each $j < m$,

$$x_j i_j f = x_{j+1}(p + i_{j+1}); \quad \text{also} \quad x_m i_m f = x' i'.$$

The preiteration theory $[X]$ is an iteration theory since the "forgetful functor" $[[X \circ N, \square]] \to [X]$ is a theory morphism with property (3.3.4). The proof of this statement is in [5]. (The iterative theory $[[X \circ N, \square]]$ of "timed terminal functions" is defined in [5, p. 200].) Hence any equation valid in $[[X \circ N, \square]]$ will hold in $[X]$.

Problem 3.9. Is there any other way of defining $f^{\dagger}$ on $[X]$ so that the resultant preiteration theory is an iteration theory?

Example 3.10. One of the referees asked whether any ideal *iteration* theory is necessarily an *iterative* theory. We answer the question negatively with the following example. Let S be the set consisting of the nonnegative integers N and two "new" points $a \neq b$. Let $g\colon S \to S$ be the function such that $ng = n + 1$, for $n \in N$ and $ag = a$, $bg = b$. Let P be the least subtheory of $S^{\cdot}$ containing $g\colon 1 \to 1$, $a\colon 1 \to 0$ and $b\colon 1 \to 0$. (A morphism $n \to p$ in $S^{\cdot}$ is a function $S^p \to S^n$; composition is function composition and the distinguished morphism $i\colon 1 \to n$ is the ith projection function $S^n \to S$.) P is easily seen to be an ideal theory which is not iterative, since the iteration equation for g has the two solutions a and b. Now define $g^{\dagger} = a = I_1^{\dagger}$ in P, and extend † to all other morphisms using the equations (2.1), (2.4.1), (2.4.2) and the pairing identity. We will use (3.3.4′) to show that P is an iteration theory.

Let $\Gamma_0 = \{\bar{a}, \bar{b}\}$, $\Gamma_1 = \{\bar{g}\}$, $\Gamma_n = \phi$, $n \geqq 1$. Note that there is only one infinite tree $1 \to 0$ in $\Gamma\mathrm{tr}$, namely $\bar{g}^{\dagger}$. We define $H\colon \Gamma\mathrm{tr} \to P$ on the finite trees by the requirements that

$\bar{a}H = a$, $\bar{b}H = b$, $\bar{g}H = g$ and that H be a theory morphism. If we now define $\bar{g}^\dagger H = a$, H preserves scalar iteration on ideal morphisms and remains a theory morphism. Thus by 3.1, H preserves vector iteration on ideal morphisms. Clearly H is surjective, so that by (3.3.4′), P is an iteration theory.

4. Examples involving partial ordering. Let T be an algebraic theory. A *compatible partial ordering* on T is a family of partial orderings $\sqsubseteq$ on the sets $T_{n,p}$ of morphisms $n \to p$ in T which are "compatible" with the theory operations in that

$$\text{(4.1)} \qquad \begin{aligned} &\text{if} \quad f_1 \sqsubseteq f_2 \text{ in } T_{n,p} \text{ and } g_1 \sqsubseteq g_2 \text{ in } T_{p,q} \\ &\text{then} \quad f_1 \circ g_1 \sqsubseteq f_2 \circ g_2 \text{ in } T_{n,q}; \end{aligned}$$

also

$$\text{(4.2)} \qquad \begin{aligned} &\text{if } f_i \sqsubseteq g_i \text{ in } T_{1,p}, \text{ each } i \in [n], \\ &\text{then} \quad (f_1, \cdots, f_n) \sqsubseteq (g_1, \cdots, g_n) \text{ in } T_{n,p}. \end{aligned}$$

In § 2 it was shown how to extend the iteration operation to all morphisms in an iterative theory J, depending on an arbitrary morphism $\perp : 1 \to 0$ in J, in such a way that all iterative theory identities are preserved. In particular the value $g^\dagger$ of the extended iteration operation on g is always a solution of the iteration equation for g; i.e.

$$g^\dagger = g \circ (I_p, g^\dagger).$$

In many treatments of the semantics of programming languages solutions of the iterative equation for g are found as "least" fixed points of the operation

$$\xi \mapsto g \circ (I_p, \xi)$$

(see [1] for one example). We give without proof two examples where the extended iteration operation yields the least fixed point of the iteration equation with respect to some compatible partial ordering.

Example 1. Let J be the iterative theory $\Gamma\mathrm{Tr}$ of all Γ-trees, and suppose $\perp \in \Gamma_0$; i.e. is an "atomic" tree $1 \to 0$. For any f: $n \to p$, let B_f be the set of vertices (leaves in this case) of f labeled $\perp$. *Define* $f \sqsubseteq_\perp g$ if g can be obtained from f by attaching a Γ-tree h_v: $1 \to p$ to each v in B_f. We claim the following

4.3. $\sqsubseteq_\perp$ *is a compatible partial ordering on* $\Gamma\mathrm{Tr}$ *and that in* ($\Gamma\mathrm{Tr}$, $I_1^\dagger = \perp$), $g^\dagger$ *is the* $\sqsubseteq_\perp$*-least solution of the iteration equation for g.*

(The fact that $\sqsubseteq_\perp$ is compatible, for this choice of $\perp$, is proved in [1].)

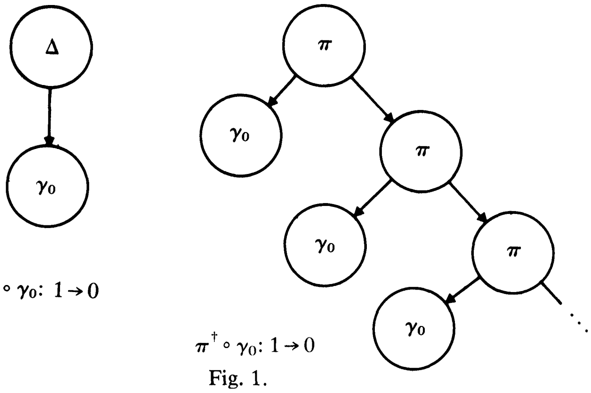
Example 2. Let $J = \Gamma\mathrm{Tr}$ again and this time let "$\perp$" denote the infinite tree $\Delta^\dagger$, where $\Delta \in \Gamma_1$. For f: $n \to p$ in J, let B_f be the set of all vertices v of f such that

(a) the tree of descendants of v in f is isomorphic to $\perp$; and

(b) no predecessor of v has property (a).

If we *define* $f \sqsubseteq_\perp g$ as in Example 1 (with this new definition of B_f), $\sqsubseteq_\perp$ is also a compatible partial ordering on $\Gamma\mathrm{Tr}$ and 4.3 holds on this case also.

Example 3. By way of contrast, if $J = \Gamma\mathrm{Tr}$ and $\perp$ is either the finite tree $\Delta \circ \gamma_0$, where $\Delta \in \Gamma_1$ and $\gamma_0 \in \Gamma_0$, or the infinite tree $\pi^\dagger \circ \gamma_0$, (see Fig. 1) where $\pi \in \Gamma_2$ and $\gamma_0 \in \Gamma_0$, there is no partial ordering $\sqsubseteq_\perp$ on $\Gamma\mathrm{Tr}$ such that 4.3 holds.



$\Delta \circ \gamma_0 \colon 1 \rightarrow 0$

$\pi^{\dagger} \circ \gamma_0 \colon 1 \rightarrow 0$

Fig. 1.

The following theorem, which explains these examples, was obtained in collaboration with Professor Ralph Tindell.

THEOREM 4.4. *Let $\perp \colon 1 \rightarrow 0$ be a tree in* ΓTr. *There is a compatible partial ordering $\sqsubseteq$ on* ΓTr *such that in* $(\Gamma\text{Tr}, I_1^{\dagger} = \perp)$ $g^{\dagger}$ *is the $\sqsubseteq$-least solution of the iteration equation for g iff $\perp$ is homogeneous; i.e. for each vertex v of $\perp$, the tree of descendants of v in $\perp$ is isomorphic to $\perp$.*

We prove the necessity of the condition. Thus suppose $\sqsubseteq$ is a compatible partial ordering on ΓTr such that for each $g \colon n \rightarrow p+n$ in $(\Gamma\text{Tr}, I_1^{\dagger} = \perp)$ $g^{\dagger}$ is the $\sqsubseteq$-least solution of the iteration equation for g. In the case $g = I_1$, it follows that $\perp$ is the $\sqsubseteq$-least morphism $1 \rightarrow 0$ in ΓTr. Let v_0 be any vertex of $\perp$ and let $\tau \colon 1 \rightarrow 0$ be the tree of descendants of v_0. Let $\sigma \colon 1 \rightarrow 1$ be the tree obtained from $\perp$ by deleting all the successors of v_0 (if any) and relabeling v_0 by "1". Then σ has precisely one leaf labeled "1", and clearly

$$\perp = \sigma \circ \tau. \tag{4.4.1}$$

Since $\perp$ is the $\sqsubseteq$-least morphism $1 \rightarrow 0$,

$$\perp \sqsubseteq \tau \tag{4.4.2}$$

and

$$\perp \sqsubseteq \sigma \circ \perp. \tag{4.4.3}$$

By (4.4.2), since $\sqsubseteq$ is compatible,

$$\sigma \circ \perp \sqsubseteq \sigma \circ \tau = \perp. \tag{4.4.4}$$

Thus

$$\sigma \circ \perp = \sigma \circ \tau$$

which shows that $\perp$ is isomorphic to τ (since both are the tree of descendants of v_0 in $\sigma \circ \perp = \sigma \circ \tau = \perp$).

The proof of the sufficiency of the condition will appear in [4].

5. ΓTr as a metric space. The free iterative theory Γtr is a subtheory of the iterative theory ΓTr of *all* labeled Γ-trees (studied in [6]). In this section it is observed that ΓTr is a complete metric space and that for any Γ-tree $\perp \colon 1 \rightarrow 0$ and any morphism $f \colon n \rightarrow p+n$ in the corresponding iteration theory $(\Gamma\text{Tr}, I_1^{\dagger} = \perp)$, the value of the extended iteration operation $f^{\dagger}$ is a metric limit of the trees $f^k \circ (I_p \oplus {}_{\perp n})$.

In [6] the *profile* of an unlabeled tree was defined. The definition is extended to all labeled Γ-trees in the expected way.

DEFINITION 5.1. Let $g: n \to p$ be a tree in $\Gamma\mathrm{Tr}$. For any natural number $d \geqq 0$, the *profile* of g at length (or "depth") d, in symbols $P_d(g)$, is the sequence of elements in $\Gamma \cup [p]$.

$$l(w_1), l(w_2), \cdots, l(w_k)$$

where $w_1, w_2, \cdots, w_k$, $k \geqq 0$, is the sequence (from left to right) of the vertices of g of distance d from a root and where $l(w_i) \in \Gamma \cup [p]$ is the label of w_i, $i \in [k]$.

Two trees have the same profile at every depth iff they are isomorphic. Hereafter we identify isomorphic trees, so that

$$g = g' \quad \text{iff} \quad P_d(g) = P_d(g') \text{ for all } d \geqq 0.$$

More generally, g and g' are identical up to vertices of length $\leqq l$ iff $P_d(g) = P_d(g')$ for $d \leqq l$.

Using the concept of the profile of a tree, we will make the Γ-trees into a metric space.

5.2. Let $(r_n: n = 0, 1, 2, \cdots)$ be a sequence of positive real numbers. Let $g, g': 1 \to p$ be two scalar Γ-trees.

DEFINITION (the function d). If $g \neq g'$, we define

$$d(g, g') = r_{n_0}$$

where n_0 is the least integer k such that $P_k(g) \neq P_k(g')$; if $g = g'$, we let $d(g, g') = 0$. For vector trees $g, g': n \to p$, $n > 1$, we *define*

$$d(g, g') = \max \{d(i \circ g, i \circ g'): i \in [n]\}.$$

We note immediately that

$$(5.2.1) \qquad d(g, g') = 0 \quad \Leftrightarrow \quad g = g' \qquad (\text{since } r_k > 0, \text{ all } k)$$

$$(5.2.2) \qquad d(g, g') = d(g', g).$$

PROPOSITION 5.2.3. *For each $n, p \geqq 0$ the function d is a metric on the set of Γ-trees $n \to p$ in $\Gamma\mathrm{Tr}$.*

Proof. It follows from well-known facts that we need prove the assertion only for the case $n = 1$. (See e.g. Munkres, Theorem 1.3, p. 266, *Topology*, Prentice-Hall, Englewood Cliffs, NJ 1975.) In view of (5.2.1) and (5.2.2), we need verify only the triangle inequality (5.2.4) to show d is a metric. If any two of the three trees g, g' and g'': $1 \to p$ are equal, it is clear that

$$(5.2.4) \qquad d(g, g'') \leqq d(g, g') + d(g', g'').$$

Now suppose that all three trees are pairwise distinct. Let $d(g, g') = r_m$ and $d(g', g'') = r_{m'}$. In the case $m \leqq m'$ we have by Def. 5.1, $d(g, g'') = r_m$. From this fact (5.2.4) follows.

PROPOSITION 5.2.5. *If the sequence $(r_n: n \geqq 0)$ converges monotonically down to zero, the metric d is complete; i.e. every Cauchy sequence converges.*

Proof. Let $(g_k: k \geqq 0)$ be a Cauchy sequence of trees $1 \to p$ in $\Gamma\mathrm{Tr}$. Thus for every real number $\varepsilon > 0$ there is a natural number $n(\varepsilon)$ such that $d(g_k, g'_k) < \varepsilon$ whenever $k, k' > n(\varepsilon)$. If $m(\varepsilon)$ is the least integer such that $r_k < \varepsilon$ for all $k > m(\varepsilon)$, then we have

$$P_d(g_k) = P_d(g_{k'}), \quad \text{all } d \leqq m(\varepsilon)$$

for all $k, k' > n(\varepsilon)$. Thus we may define a tree $g: 1 \to p$ by requiring that for all $i \geqq 1$:

$$P_d(g) = P_d(g_{n(\varepsilon_i)}), \, d \leqq m(\varepsilon_i)$$

where $\varepsilon_i = 1/i$. Then clearly

$$\lim_{k\to\infty} g_k = g.$$

Thus any Cauchy sequence converges.

From now on we will assume the sequence $(r_n: n \geqq 0)$ *converges monotonically to zero.* It is easily shown that any two such sequences yield equivalent metrics, in the sense that the two induced topologies are the same. In the paper [7] the Γ-trees $1 \to 0$ were observed to be a complete metric space with the metric of § 5.2 corresponding to the sequence $r_n = 1/n+1$, $n \geqq 0$.

5.3. The metric behaves nicely with respect to the theory operations.

LEMMA 5.3.1. *Let* $g_i: n \to p$, $h_i: p \to q$ *be* Γ-*trees*, $i = 1, 2$. *Then*

$$d(g_1 \circ h_1, g_2 \circ h_1) \leqq d(g_1, g_2), \tag{5.3.2}$$

and

$$d(g_1 \circ h_1, g_1 \circ h_2) \leqq d(h_1, h_2). \tag{5.3.3}$$

$$d(g^s, h^s) \leqq d(g, h), \quad \textit{any } s \geqq 0, g, h: n \to p+n. \tag{5.3.4}$$

Proof. We prove only (5.3.2) since the proofs of (5.3.3) and (5.3.4) are similar. If $g_1 = g_2$ there is nothing to prove. Otherwise, let $d(g_1, g_2) = r_k$. Since the trees $g_1 \circ h_1$ and $g_2 \circ h_1$ are obtained by "attaching" h_1 to the termini of g_1 and g_2, $P_d(g_1 \circ h_1) = P_d(g_2 \circ h_1)$ all $d \leqq k$. Thus $d(g_1 \circ h_1, g_2 \circ h_1) \leqq d(g_1, g_2)$.

Note that it is possible to have $g_1 \neq g_2$ but $g_1 \circ h_1 = g_2 \circ h_1$; thus in (5.3.2) we cannot in general assert equality. Indeed if $g: 1 \to 1$ is any finite ideal Γ-tree, we can let $g_1 = g$, $g_2 = g \circ g$ and $h_1 = g^\dagger$.

COROLLARY 5.3.5. *The theory operations of composition and source-pairing are continuous.*

Proof. Suppose (g_k) is a sequence of Γ-trees $n \to p$ with limit g, and (h_k) is a sequence of Γ-trees $p \to q$ with limit h. We show that $\lim_{k\to\infty} g_k \circ h_k = g \circ h$. Indeed, from the triangle inequality,

$$d(g_k \circ h_k, g \circ h) \leqq d(g_k \circ h_k, g_k \circ h) + d(g_k \circ h, g \circ h).$$

But by Lemma 5.3.1, the righthand side is less than $d(h_k, h) + d(g_k, g)$, which goes to zero as $k \to \infty$.

The proof that source pairing is continuous is simpler and is omitted.

It will be shown later that iteration and extended iteration are continuous as well.

5.4. The main result of this section is an explicit description of extended iteration in the iteration theory (ΓTr, $I_1^\dagger = \perp$), for any Γ-tree $\perp: 1 \to 0$. We will show that $g^\dagger$ is a metric limit of the trees $g^k \circ (I_p \oplus \perp_n)$, for any Γ-tree $g: n \to p+n$. ($\perp_n$ was defined in Cor. 2.8.)

Call a morphism $g: n \to p+n$ in any iterative theory "quasi-ideal" if for each $i \in [n]$, $i \circ g$ is either ideal or $j \oplus O_n$, for some base $j \in [p]$. In other words, every component position of a quasi-ideal morphism is successful or ideal.

LEMMA. *Suppose* $f: n \to p+n$ *is a quasi-ideal morphism in* ΓTr. *Then for any* $\alpha: n \to p$ *in* ΓTr,

$$\lim_{k\to\infty} f^k \circ (I_p, \alpha) = f^\dagger.$$

Proof. It is easily seen that for any $k \geqq 0$ and $d \leqq k$,

$$P_d(f^{k+1}) = P_d(f^{k+1} \circ (I_p, \alpha)).$$

But since $f^\dagger = f^{k+1} \circ (I_p, f^\dagger)$, we have for $d \leqq k$

$$P_d(f^\dagger) = P_d(f^{k+1} \circ (I_p, \alpha)),$$

from which the result follows.

PROPOSITION. *If $h: n \to p+n$ is a nonsingular morphism in $\Gamma\mathrm{Tr}$, then for any $\alpha: n \to p$, the sequence*

(5.4.1) $$h \circ (I_p, \alpha), h^2 \circ (I_p, \alpha), \cdots$$

is Cauchy and converges to $h^\dagger$.

Proof. Let $h = g_\perp$ (defined in 2.7), so that $h^\dagger = g^\ddagger$, by Theorem 2.7. By the proposition of § 5.4 and the fact that $(I_p, \perp_n \circ O_p) = I_p \oplus \perp_n$, we have

$$P_k(h^s \circ (I_p, \alpha)) = P_k(h^s) = P_k(h^t) = P_k(h^t \circ (I_p, \alpha)).$$

Thus the sequence (5.4.1) is Cauchy and therefore converges. Let $f = h^n$. Then $f^\dagger = h^\dagger$, and $\lim_{k\to\infty} f^k \circ (I_p, \alpha) = f^\dagger$, by the Lemma. But since $f \circ (I_p, \alpha), f^2 \circ (I_p, \alpha), \cdots$ is a subsequence of (5.4.1), it follows that the limit of (5.4.1) is $h^\dagger = f^\dagger$.

We may now prove the following:

THEOREM 5.5. *Let $g: n \to p+n$ be an arbitrary Γ-tree in ($\Gamma\mathrm{Tr}$, $I_1^\dagger = \perp$). Then*

$$\lim_{k\to\infty} g^k \circ (I_p \oplus \perp_n) = g^\dagger.$$

Proof. Let $h = g_\perp$ (defined in 2.7), so that $h^\dagger = g^\dagger$, by Theorem 2.7. By the proposition of § 5.4 and the fact that $(I_p, \perp_n \circ O_p) = I_p \oplus \perp_n$, we have

$$\lim_{k\to\infty} h^k \circ (I_p \oplus \perp_n) = h^\dagger = g^\dagger.$$

In order to prove the theorem, we prove that

(5.5.1) $$h^k \circ (I_p \oplus \perp_n) = g^k \circ (I_p \oplus \perp_n), \quad \text{all } k \geqq 1.$$

We prove (5.5.1) by induction on k.

When $k = 1$, we have to show

(5.5.2) $$g_\perp \circ (I_p \oplus \perp_n) = g \circ (I_p \oplus \perp_n).$$

If $i \notin K_g$, $i \circ g_\perp = i \circ g$, so $i \circ g_\perp \circ (I_p \oplus \perp_n) = i \circ g \circ (I_p \oplus \perp_n)$. If $i \in K_g$, $i \circ g_\perp = \perp \circ O_{n+p}$, so that $i \circ g_\perp \circ (I_p \oplus \perp_n) = \perp_n \circ O_p$. Also $i \circ g = O_p \oplus i'$, some $i' \in [n]$, since $i \in K_g$, and thus $i \circ g \circ (I_p \oplus \perp_n) = \perp_n \circ O_p$. This completes the proof of (5.5.2).

Now assume (5.5.1) holds for k. Then

$$\begin{aligned} h^{k+1} \circ (I_p \oplus \perp_n) &= h \circ (I_p \oplus O_n, h^k) \circ (I_p \oplus \perp_n) \\ &= h \circ (I_p, h^k \circ (I_p \oplus \perp_n)) \\ &= h \circ (I_p, g^k \circ (I_p \oplus \perp_n)), \end{aligned}$$

by the induction hypothesis. We now show

(5.5.3) $$g_\perp \circ (I_p, g^k \circ (I_p \oplus \perp_n)) = g \circ (I_p, g^k \circ (I_p \oplus \perp_n)).$$

Let L and R be the morphisms on the left and right sides of the equation (5.5.3). We show $i \circ L = i \circ R$ for all $i \in [n]$. We clearly need only consider the case $i \in K_g$. In this case $i \circ g_\perp = \perp \circ O_{n+p}$, so that $i \circ L = \perp \circ O_p$. But since $i \in K_g$, $i \circ g^{k+1} = O_p \oplus i'$, for some $i' \in [n]$. Hence $i \circ R = i \circ g^{k+1} \circ (I_p \oplus \perp_n) = \perp \circ O_p$ completing the proof.

5.6. The continuity of extended iteration will be shown to follow from Theorem 5.5 and the Lemma 5.3.1.

PROPOSITION. *Let $g_k\colon n \to p+n$, $k \geqq 0$, be a sequence of Γ-trees converging to the tree $g\colon n \to p+n$ in* (ΓTr, $I_1^\dagger = \perp$). *Then the sequence*

$$g_0^\dagger, g_1^\dagger, \cdots,$$

converges to $g^\dagger$.

Proof. By the triangle inequality, for any $k, s > 0$

$$\begin{aligned} d(g_k^\dagger, g^\dagger) &\leqq d(g_k^\dagger, (g_k)^s \circ (I_p \oplus \perp_n)) + d((g_k)^s \circ (I_p \oplus \perp_n), g^s \circ (I_p \oplus \perp_n)) \\ &\quad + d(g^s \circ (I_p \oplus \perp_n), g^\dagger). \end{aligned} \tag{5.6.1}$$

By (5.3.2) and (5.3.4), the middle term is not greater than $d(g_k, g)$, for any s. Thus, given any real $\varepsilon > 0$, first choose k such that $d(g_k, g) < \varepsilon/3$ and for that k, choose s so large that both the first and third summand in (5.6.1) are less than $\varepsilon/3$, using Theorem 5.5. This completes the proof.

Appendix. In this section we will prove the fact stated in the proof of Theorem 2.7.

LEMMA. *If $f\colon n \to p+n$ is a morphism in $(J, I_1^\dagger = \perp)$ and $i \in K_f$, then $i \circ f^\dagger = \perp \circ O_p$.*

We will use the function f^σ (defined in [3, 2.3.10]) and we prove first the following fact:

PROPOSITION A. *In $(J, I_1^\dagger = \perp)$ if $i \in K_f$, and if $f^\sigma = u_1 u_2 u_3 \cdots$ then $i \circ f^\dagger = u_r \circ f^\dagger$, for any $r \geqq 1$.*

Proof. When $r = 1$, $u_r = i$ and there is nothing to prove. Assume $i \circ f^\dagger = u_r \circ f^\dagger$. From [3, (2.3.11)] and the fact $f^\dagger = f^r \circ (I_p, f^\dagger)$, we obtain

$$\begin{aligned} i \circ f^\dagger = i \circ f^r \circ (I_p, f^\dagger) &= [O_p \oplus u_{r+1}] \circ (I_p, f^\dagger) \\ &= u_{r+1} \circ f^\dagger, \quad \text{by [3, (1.6.2) and (1.6.3)]}, \end{aligned}$$

completing the induction.

We now prove the Lemma by induction on $n \geqq 1$. If $n = 1$ and $i = 1 \in K_f$, then $f = O_p \oplus I_1$, so that $i \circ f^\dagger = \perp \circ O_p$ by (2.1') and (2.4.1). Now assume $f\colon n+1 \to p+n+1$. Write $f = (f_1, f_2)$ where f_1 has source n and f_2 has source 1, and define g by (2.3). Then (2.2) holds. Assume now $i \in K_f$ and assume inductively: $i \in K_g \Rightarrow i \circ g^\dagger = \perp \circ O_p$. We distinguish three cases.

Case 1. *if* $f^\sigma = u_1 u_2 \cdots u_r (n+1)(n+1) \cdots$, $r \geqq 0$, where $u_i \in [n]$ for each $i \in [r]$.

Case 2. *if* $f^\sigma = u_1 u_2 \cdots u_r (n+1) u_{r+2} \cdots$, $r \geqq 0$, where $u_i \in [r]$ and $u_{r+2} \in [n]$.

Case 3. *if* $f^\sigma = u_1 u_2 \cdots u_i \cdots$, where $u_i \in [n]$ for all i.

On Case 1. $(n+1) \circ f = p+n+1 = O_{p+n} \oplus I_1 = f_2$. Hence by (2.1') and (2.4.1): $f_2^\dagger = O_{p+n} \oplus \perp = \perp \circ O_{p+n}$ so that by (2.2) we infer $(n+1) \circ f^\dagger = \perp \circ O_p$. Since $u_{r+1} = (n+1)$, it follows (cf. (2.3.11) of [3]) that

$$i \circ f^{r+1} = (n+1) \circ f. \tag{A.1}$$

From Proposition A we infer

$$i \circ f^\dagger = (n+1) \circ f^\dagger = \perp \circ O_p. \tag{A.2}$$

Thus in this case (without the aid of the inductive assumption) we've shown $i \in K_f \Rightarrow i \circ f^\dagger = \perp \circ O_p$.

In preparation for the remaining cases we make the following *Observation*: where $i, i' \in [n]$,

(A.3) $i \circ f = p + i' \Rightarrow i \circ g = p + i'$; in the notation of [3, 2.3.3], $if^\nu \in [n] \Rightarrow ig^\nu = if^\nu$;

(A.4) $[i \circ f = p + n + 1 \;\&\; (n+1) \circ f = p + i'] \Rightarrow i \circ g = p + i'$;
$[if^\nu = n + 1 \;\&\; (n+1)f^\nu \in [n]] \Rightarrow ig^\nu = (n+1)f^\nu$.

Assertion (A.3) follows from (2.3); (A .4) follows from (2.3) together with the observation that since $f_2 = p + i'$ and $i' \in [n]$, we have $f_2^\dagger = p + i'$. It follows that in Case 2 we have

$$ig^\sigma = u_1 u_2 \cdots u_r u_{r+2} \cdots$$

while in Case 3, we have $ig^\sigma = if^\sigma$. Thus in Cases 2 and 3

(A.5) $$i \in K_f \quad \Rightarrow \quad i \in K_g.$$

With the aid of the inductive assumption we conclude $i \circ g^\dagger = \perp \circ O_p$, and thus from (2.2), $i \circ f^\dagger = \perp \circ O_p$. This shows that for Cases 2 and 3:

(A.6) $$\text{if } i \in K_f \cap [n], \quad \text{then } i \circ f^\dagger = \perp \circ O_p.$$

It remains to show for Case 2 where $i = n + 1$ that $i \circ f^\dagger = \perp \circ O_p$. But

$$(n+1) \circ f^\dagger = (n+1) \circ f \circ (I_p, f^\dagger) = (O_p \oplus u_{r+2}) \circ (I_p, f^\dagger) = u_{r+2} \circ f^\dagger.$$

By (A.6), $u_{r+2} \circ f^\dagger = \perp \circ O_p$.

The proof is complete since the three cases considered exhaust all possibilities. Thus if $if^\sigma = u_1 u_2 \cdots$ and there is a t such that $u_{t+1} = n + 1$, we choose r to be the smallest such t. Case 1 obtains if $(n+1)f^\nu = n + 1$ while Case 2 occurs if $(n+1)f^\nu \neq n + 1$. If there is no such t, then Case 3 obtains.

REFERENCES

[1] J. B. Wright, J. W. Thatcher, E. G. Wagner and J. A. Goguen, *Rational algebraic theories and fixed point solutions*, Proc. IEEE Symp. Foundations of Comp. Sci. (Houston, Texas), Oct. 1976.

[2] S. L. Bloom, S. Ginali and J. Rutledge, *Scalar and vector iteration*, J. Comput. System Sci., 14 (1977), pp. 251–256.

[3] S. L. Bloom, C. C. Elgot and J. B. Wright, *Solutions of the iteration equation and extensions of the scalar iteration operations*, this Journal, 9 (1980), pp. 25–45.

[4] S. L. Bloom and R. Tindell, *Compatible orderings on the metric theory of trees*, this Journal, to appear.

[5] C. C. Elgot, *Monadic computation and iterative algebraic theories*, *Logic Colloq.* '73, 80, Studies in Logic, North Holland, 1975.

[6] C. C. Elgot, S. L. Bloom and R. Tindell, *The algebraic structure of rooted trees*, J. Comput. System Sci., 16 (1978), no. 3, pp. 362–399; extended abstract in Proc. Johns Hopkins 1977 Conf. Inf. Sci. and Systems.

[7] J. Mycielski and W. Taylor, *A compactification of the algebra of terms*, Algebra Universalis, 6 (1976), pp. 159–163.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0007 $01.00/0

BOUNDS ON THE SCHEDULING OF TYPED TASK SYSTEMS*

JEFFREY M. JAFFE†

Abstract. We study the scheduling of different types of tasks on different types of processors. If there are k types of tasks and m_i identical processors for tasks of type i, the finishing time of any list schedule is at most $k+1-(1/\max(m_1, \cdots, m_k))$ times worse than the optimal schedule. This bound is best possible. If the processors execute at different speeds then the performance of any list schedule (relative to the optimal schedule) is bounded by k plus the maximum ratio between the speeds of any two processors of the same type.

Key words. scheduling, list scheduling, typed task systems, data flow computation, worst case performance bounds

1. Introduction. The problem of job scheduling on multiprocessor systems has been extensively studied (for a current survey see [1], [8]). The conventional approach has been to consider a system where each processor may handle any job or task. These systems are referred to as "ordinary" task systems. In some systems certain tasks may be processed only by designated processors for those tasks. Examples of these include data flow models of computation [3], [10] where primitive operations are computed by different processors. Similarly, in machines such as the CDC6600 [17], there are several specialized functional modules. Also, in a system where I/O tasks and arithmetic tasks are handled by different processor units, such an assumption may be relevant. In this paper we analyze some of the properties of schedules for systems with different *types* of tasks. Many of the results for ordinary task systems generalize trivially to the typed case. The *NP* completeness results of [18] clearly carry over directly, as do approximate solutions for certain special cases (for example when the tasks are independent [4]).

The complexity of determining the optimal schedule is *NP*-complete even in very simple cases. It is shown in [6], that the decision problem of determining whether a given typed task system can be scheduled with a finishing time smaller than a given bound is *NP*-complete even if there are only two processors, one of each of two types. Also, if the number of types of processors varies, the decision problem is *NP*-complete even if the precedence constraint is restricted to being a forest. The techniques used in [6] are adaptations of those found in [1], [5].

The focus of this paper is to extend the results of Graham [7], which provide bounds for non-preemptive list schedules. List schedules are a class of schedules that satisfy fundamental "no-waste" requirements. The performance criterion is that we attempt to minimize the finishing time of the system. In ordinary task systems, any list schedule is at most $2-(1/m)$ times worse than optimal where m is the number of processors. For typed task systems as defined in § 2 an analogous bound is obtained. With a similar definition of "unwasteful schedules", it is shown in § 3 that any such schedule is at most $k+1-(1/\max(m_1, \cdots, m_k))$ times worse than optimal, where k is the number of types of tasks and m_i is the number of processors of type i. This bound is best possible for all values of $k, m_1, \cdots, m_k$ as shown in § 4.

The results of [12], [13] which provide bounds for list schedules for task systems that are scheduled on processors of different speeds are also extended. In ordinary task systems with processors of different speeds, any list schedule is at most (approximately) $(f/s)+1$ times worse than optimal where f is the speed of the fastest processor and s the

* Received by the editors August 15, 1978. This report was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation Grant MCS77-19754.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Now at IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

speed of the slowest processor. It is shown (in §§ 6 and 7) that the bound for typed task systems is (approximately) $k+\max(f_1/s_1, \cdots, f_k/s_k)$ where f_i is the speed of the fastest processor of type i and s_i is the speed of the slowest processor of type i. In Section 8, generalizations of [9], [11] are briefly discussed.

2. Typed task systems. An *ordinary task system* $(\mathcal{T}, <, \mu)$ consists of:

(1) A set $\mathcal{T}$ of *n tasks*.

(2) A *partial ordering* $<$ on $\mathcal{T}$.

(3) A *time function* $\mu : \mathcal{T} \rightarrow \mathbb{R}$.

The set $\mathcal{T}$ represents the set of tasks or jobs that need to be executed. The partial ordering specifies which tasks must be executed before other tasks. The value $\mu(T)$ is the *time requirement* of the task T.

A *k type task system* $(\mathcal{T}, <, \mu, \nu)$ is a task system $(\mathcal{T}, <, \mu)$ together with a *type function* $\nu : \mathcal{T} \rightarrow (1, \cdots, k)$. Intuitively, if $\nu(T) = i$ then T must be executed by a processor of type i.

A *schedule* for $(\mathcal{T}, <, \mu, \nu)$ is a total function $S : \mathcal{T} \rightarrow \mathbb{R}$. We refer to $S(T)$ as the *starting time* of the task T and $S(T)+\mu(T)$ as the *finishing time* of the task T. We also say that $T \in \mathcal{T}$ is *being executed at time t* for times t such that $S(T) \leqq t < S(T)+\mu(T)$.

A *valid schedule* for $(\mathcal{T}, <, \mu, \nu)$ *on a set of equally fast processors* $\mathcal{P} = \{P_{ij} : 1 \leqq i \leqq k \text{ and } 1 \leqq j \leqq m_i\}$ is a schedule for $(\mathcal{T}, <, \mu, \nu)$ with the properties:

(1) For all $i = 1, \cdots, k$ and all $t \in \mathbb{R}$ the number of tasks of type i being executed at time t does not exceed m_i.

(2) For $T, T' \in \mathcal{T}$, if $T < T'$, the starting time of T' is at least as large as the finishing time of T.

Condition one asserts that processor capabilities may not be exceeded. Condition two forces the obedience of precedence constraints.

The *finishing time* of a valid schedule is the maximum finishing time of the set of tasks. An *optimal schedule* is any valid schedule that minimizes the finishing time. For two valid schedules S and S', with finishing times w and w' the *performance ratio of S to S'* is w/w'.

There are schedules that may be arbitrarily worse than the optimal schedule. For example, there may be a time before the finishing time at which no task is being executed, but such trivially improvable schedules are not interesting. We restrict attention to *list schedules* which have attracted considerable attention for ordinary task systems [7], [12], [13].

List schedules are designed to avoid the apparently wasteful behavior of letting a processor be idle while there are executable tasks. A list schedule uses a (priority) *list L* which is a permutation of the set $\mathcal{T}$, i.e., $L = (T_1, \cdots, T_n)(T_i \in \mathcal{T}$ and for $i \neq j$, $T_i \neq T_j)$. The *list schedule* for $(\mathcal{T}, <, \mu, \nu)$ with the list L is constructed as follows. At each point in time that at least one processor completes a task, the processors that are not still executing are assigned unexecuted executable tasks of their respective types if such tasks are available. The tasks are chosen by giving higher priority to those tasks with the lower indices in L. Any schedule that is unwasteful in the sense that processors are never permitted to be idle unless no free tasks of the same type are available can be formulated as a list schedule.

The motivation for this heuristic comes from several sources. The primary motivation emanates from the optimality of some list schedule in the unit execution time case. When each task requires an equal amount of time at least one list schedule is an optimal schedule. Other sources of interest include the fact that it is simple to implement, and due to its simplicity, it is a good starting place for building other heuristics.

To analyze list schedules the following definitions are useful. A *chain* C is a sequence of tasks $C=(T_1, \cdots, T_l)$ with $T_i \in \mathcal{T}$ such that for all j, $1 \leqq j < l$, $T_j < T_{j+1}$. C *starts* with task T_1. The *length* of C is $\sum_{j=1}^{l} \mu(T_j)$. The *height of a task* $T \in \mathcal{T}$ equals the length of the longest chain starting at T. The *height of* $(\mathcal{T}, <, \mu, \nu)$ equals the length of the longest chain starting at any task $T \in \mathcal{T}$.

While the notion of the height of a task is a static notion which is a property of $(\mathcal{T}, <, \mu, \nu)$ we also associate a dynamic notion of the height of a task with any schedule for $(\mathcal{T}, <, \mu, \nu)$. Specifically, let S be a schedule for $(\mathcal{T}, <, \mu, \nu)$, and let t be less than the finishing time of S. Then the *height of the task T at time t* is equal to the length of the longest chain starting at T, where the length of the chain considers only the unexecuted time requirements. Similarly, the *height of* $(\mathcal{T}, <, \mu, \nu)$ *at time t* is the length of the longest chain starting at any task not yet completed $T \in \mathcal{T}$. Note that if a portion of a task has been finished at time t, then it contributes to the height only that proportion of the time requirement which has not yet been completed.

It is convenient to analyze schedules based on whether or not the height is decreasing during a given interval of time. One may plot the height of $(\mathcal{T}, <, \mu, \nu)$ as a function of time for a given schedule S and make the following observation. The height is a nonincreasing function which starts at the original height of $(\mathcal{T}, <, \mu, \nu)$ for $t=0$, and ends at height 0 at the finishing time of S. If during an interval of time, the height was a monotonically decreasing function of time then that interval is called a *height reducing interval.* If during an interval of time the height is constant, the interval is called a *constant height interval.*

Notation. The total time requirement of all the type i tasks will be denoted by μ_i and the (original, static) height of $(\mathcal{T}, <, \mu, \nu)$ will be denoted h.

3. Performance bounds for list scheduling. In this section a bound is obtained on the performance ratio of any list schedule to an optimal schedule. It is shown that in a k type task system, the ratio is at most $k+1-(1/\max(m_1, \cdots, m_k))$.[1] A naive performance bound is given by $m_1+\cdots+m_k$. This follows from the fact that an optimal schedule may use at most $m_1+\cdots+m_k$ processors at each point in time, and the fact that any list schedule uses at least one processor at every point in time. The result of this section is that all list schedules are far better than the naive bound. The comparison of list schedules to optimal schedules is applicable even to the situations that no optimal schedule is a list schedule.

THEOREM 1. *Let $(\mathcal{T}, <, \mu, \nu)$ be a k type task system to be scheduled on a set of equally fast processors. The performance ratio of any list schedule for $(\mathcal{T}, <, \mu, \nu)$ to an optimal schedule for $(\mathcal{T}, <, \mu, \nu)$ is at most $k+1-(1/\max(m_1, \cdots, m_k))$.*

The analysis approach that we use is to obtain a number of lower bounds, $LB_1, \cdots, LB_{k+1}$, on the finishing time of an optimal schedule for a given task system. An upper bound, UB, is obtained on the finishing time of any list schedule. The ratio $(UB/\max(LB_1, \cdots, LB_{k+1}))$ is an upper bound on the performance ratio ɔf the schedule to optimal.

LEMMA 1. *Let $(\mathcal{T}, <, \mu, \nu)$ as above. Let w_{opt} be the finishing time, of an optimal schedule for $(\mathcal{T}, <, \mu, \nu)$. Then $w_{\text{opt}} \geqq \max(h, \mu_1/m_1, \cdots, \mu_k/m_k)$.*

Proof. Clearly at most m_i units of time requirement of type i tasks may be executed during each time unit (for every i). Thus at least $\lceil \mu_i/m_i \rceil$ units of time must be spent on the execution of $\mathcal{T}$ for every i. A conservative lower bound is thus $\max(\mu_1/m_1, \cdots, \mu_k/m_k)$.

[1] Subsequent to the submission of this manuscript this result was published independently by C. L. Liu and J. W. S. Liu, *Acta Informatica*, 10 (1978) pp. 95–104, using slightly different techniques.

Also, by the way height is defined, the height of $(\mathcal{T}, <)$ may decrease at a rate of at most one per unit time. Thus h is a lower bound on the finishing time and $w_{\text{opt}} \geqq \max(h, \mu_1/m_1, \cdots, \mu_k/m_k)$. □

LEMMA 2. *Let $(\mathcal{T}, <, \mu, \nu)$ as above. Let w be the finishing time of a list schedule for $(\mathcal{T}, <, \mu, \nu)$. Then*

$$w \leqq (\mu_1/m_1) + (\mu_2/m_2) + \cdots + (\mu_k/m_k) + h(1 - (1/\max(m_1, \cdots, m_k))).$$

Proof. Given a list schedule S with finishing time w, divide the interval $[0, w]$ into constant height intervals and height reducing intervals. Note that during height reducing intervals, the height is decreasing by a rate of exactly one per unit time since the height of the greatest height task is being reduced at that rate. Thus the total length of height reducing intervals is equal to h. The goal is now to show that the total length of the constant height intervals is at most $(\mu_1/m_1) + \cdots + (\mu_k/m_k) - (h/\max(m_1, \cdots, m_k))$.

At each point in a constant height interval, all m_i processors of type i are in use for some value of i. We will prove this by contradiction. Let time t be a time within a constant height interval when for all i fewer than m_i processors are in use. Since there is an idle processor of each type and the schedule is a list schedule, it must be that all executable tasks are being executed. In particular, all of the maximum height tasks are being executed. But then, it follows that the height is being reduced, contradicting the assumption that t is in a constant height interval.

Let h_i denote the total time requirement of type i tasks executed during height reducing intervals. Clearly $\sum_{i=1}^{k} h_i \geqq h$ since during a height reducing interval of length l, the height of $(\mathcal{T}, <)$ is reduced by l, and at least l units of time requirement of tasks are completed. Now an upper bound on the length of constant height intervals is obtained. Note that the total length of constant height intervals during which m_i tasks of type i are executed is at most $\lfloor(\mu_i - h_i)/m_i\rfloor$. Thus in S, m_i tasks of type i for some i may be executed for a total duration of at most $\lfloor(\mu_1 - h_1)/m_1\rfloor + \cdots + \lfloor(\mu_k - h_k)/m_k\rfloor$ units of time. That is, the total length of all constant height intervals is at most

$$(\mu_1/m_1) + \cdots + (\mu_k/m_k) - ((h_1/m_1) + \cdots + (h_k/m_k)) \\ \leqq (\mu_1/m_1) + \cdots + (\mu_k/m_k) - (h/\max(m_1, \cdots, m_k)).$$

A bound on w is thus given by: $w \leqq (\mu_1/m_1) + \cdots + (\mu_k/m_k) + h(1 - (1/\max(m_1, \cdots, m_k)))$. □

We may now put together the upper bound on list schedules and the lower bound on optimal schedules. The two results combine to show that the worst possible performance ratio is $k + 1 - (1/\max(m_1, \cdots, m_k))$.

Proof of Theorem. Fix a k type task system $(\mathcal{T}, <, \mu, \nu)$, and let $p = \max(u_1/m_1, \cdots, \mu_k/m_k, h)$. Then a lower bound on any optimal schedule is p. A conservative upper bound on any list schedule is $(k + 1 - (1/\max(m_1, \cdots, m_k)))p$. Thus the performance ratio of the list schedule to an optimal schedule is bounded by $k + 1 - (1/\max(m_1, \cdots, m_k))$. □

4. Achievability results for list scheduling strategies. In this section it is shown that the bound of Theorem 1 is achievable. Specifically, for any k and any values of $m_1, \cdots, m_k$ there are k type task systems and list schedules for the systems with the property that the performance of the schedules approaches $k + 1 - (1/\max(m_1, \cdots, m_k))$ times worse than optimal.

The set of task systems used for this proof are sketched below (Fig. 1). Each node in the graph represents one task. Arrows specify the partial ordering and the labels of the

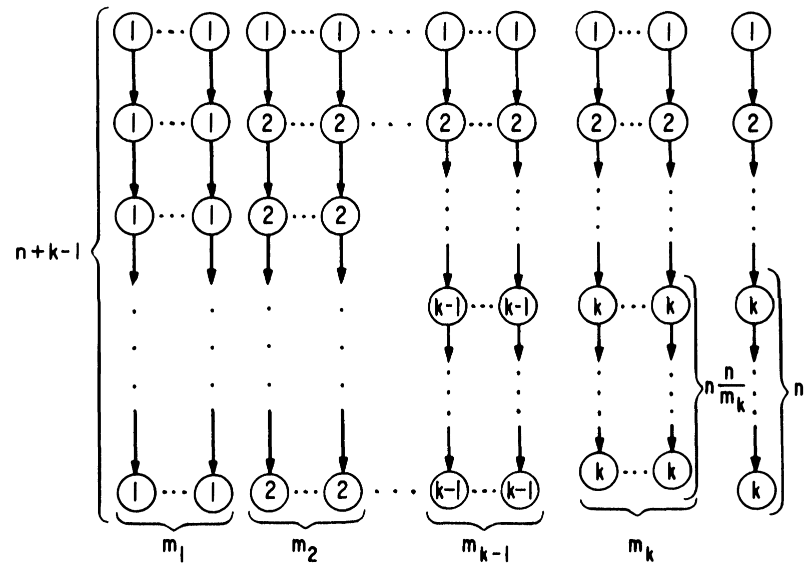


FIG. 1.

nodes represent the type of the tasks. Each task has unit execution time. Assume without loss of generality that $m_k = \max(m_1, \cdots, m_k)$.

In the task systems, there are m_i columns of tasks that consist primarily of type i tasks and are informally referred to as "corresponding to type i" $(1 \leqq i \leqq k-1)$. Each of these m_i columns contains a chain of $n+k-1$ tasks (n arbitrary). The jth task in each of these columns has $\nu(T)=j$ (for $j \leqq i-1$) and $\nu(T)=i$ (for $i \leqq j$).

There are m_k+1 columns that "correspond' to type k". In each of these columns the jth task (for $j \leqq k-1$) has $\nu(T)=j$. For the first m_k of these m_k+1 columns there is a chain of $n-(n/m_k)$ additional tasks, with $\nu(T)=k$ for each task in the chain. For the (m_k+1)st column there is a chain of n additional tasks, with $\nu(T)=k$ for each task in the chain.

The following is an asymptotically optimal strategy. The first $k-1$ tasks of each column are executed using an arbitrary list schedule. For fixed values of k and the m_i's this may be done in constant time. Now, only n units of time are required to complete the entire system. It is clear that only n units of time are required to finish the columns corresponding to each of the first $k-1$ types of processors. During these same n units of time the columns corresponding to the kth type of processor may be completed as follows: One of the m_k processors of type k is used continuously on the (m_k+1)st of these columns finishing this column in n units of time. The other m_k-1 processors are used on the other m_k columns in rotation. Thus during the first unit of time, no task is executed from the first column, during the second unit of time, no task is executed from the second column, etc. Thus, the total amount of time for this procedure is $n+O(1)$ for fixed values of $k, m_1, \cdots, m_k$.

An inefficient list schedule is now presented. The schedule first handles all type 1 tasks, then all type 2 tasks, etc. For the first $n+k-1$ units of time only tasks from columns that correspond to type 1 are executed. At the next $n+k-1$ units of time all tasks from columns that correspond to type 2 are executed, stripping off type 1 tasks from the tops of the rest of the columns in the process. In this manner, $(k-1)n+O(1)$ units of time are required to finish all of the columns that correspond to the first $k-1$ types of processors.

Now the last m_k+1 columns of the system are executed. Using a list schedule, only the first m_k of them are processed for the next $n-(n/m_k)$ units of time, completing these columns in their entirety. Another n units of time are required just to process the last of these m_k+1 columns. The total amount of time used by this schedule is thus $n(k+1-(1/m_k))+O(1)$ and the performance ratio between this and the optimal schedule is $(n(k+1-(1/m_k))+O(1))/(n+O(1))$. As n goes to infinity, the ratio approaches $k+1-(1/m_k)$. □

A few remarks may be made about the nature of the construction. First, all tasks take unit time in the example. Thus, although Theorem 1 applies to any typed task system, it is achievable even in the special case where each task requires only unit time. This is particularly significant in light of the fact that for this special case some list schedule is guaranteed to be optimal. Another feature of interest is that the system used is a disjoint union of chains. Each chain may be viewed as one large task, and each task within the chain may be viewed as a subtask of the larger task. We thus overcome the objection that the example is a contrived, complicated system which is unlikely to occur in practice. Finally, the "bad" schedule was an uncontrived type of schedule. The schedule executes those tasks which most recently became executable.

5. Uniform nonidentical processors. We now analyze the situation that each processor runs at a different rate. In the models of high speed computation that partially motivate this research [3], [10], the idea is to use many processors of potentially different speeds. This generalization is also the natural extension of the work of [12], [13] which considered processors of different speeds for ordinary task systems.

The processors of each type are assumed to be *uniform.* That is, the relative speeds of the processors are the same for every task. The more complicated situation in which certain processors handle some tasks relatively quickly, but others relatively slowly is not even very well understood for ordinary task systems. Also, this situation is less likely to occur in a system where the tasks have already been subdivided into different types. In this regard typed task systems may be viewed as a special case of nonuniform ordinary task systems. If a processor is of a different type than a task, then the speed for the processor on the task is infinity.

When the processors $\mathcal{P}=\{P_{ij}: 1\leqq i\leqq k \text{ and } 1\leqq j\leqq m_i\}$ are not equally fast, there is an associated *rate function* $r:\mathcal{P}\to\mathbb{R}$. Informally, the rate function specifies the speed of a processor. If a task T is assigned to a processor P then $\mu(T)/r(P)$ time units are required for the processing of T on P. Thus the actual time that a task T requires on P equals the time requirement of T (i.e. $\mu(T)$) only if $r(P)=1$.

Since the speeds of the processors are not the same, a schedule must specify which task is assigned to which processor. A *valid schedule* for $(\mathcal{T}, <, \mu, \nu)$ *on a set of uniform nonidentical processors* $\mathcal{P}$ *with rate function r* is a total function $S:\mathcal{T}\to\mathbb{R}\times\mathcal{P}$ satisfying conditions (a), (b) and (c) below. If $S(T)=(t, P)$ then the *starting time* of T is t, the *finishing time* of T is $t+(\mu(T)/r(P))$ and T is *being executed on P* for times x such that $t\leqq x<t+(\mu(T)/r(P))$. The function S satisfies:

(a) If $S(T)=(t, P_{ij})$ then $\nu(T)=i$.

(b) For all $T, T'\in\mathcal{T}$, and all $t\in\mathbb{R}$, T and T' are not both being executed on the same processor at time t.

(c) For $T, T'\in\mathcal{T}$, if $T<T'$ the starting time of T' is no less than the finishing time of T.

The definitions of *finishing time* of a schedule, *optimal schedule*, and *performance ratio* generalize in a straightforward manner and are omitted.

A *list schedule* may be generalized in two ways. One way is to only insist that at no point in time may a task of a certain type be executable while a processor of the same type is idle. A second potential generalization is to further insist that when tasks are assigned, the highest priority executable tasks are assigned to the fastest available processors. The bounds obtained are applicable to either generalization.

The total processing power of processors of type i, denoted r_i is defined by:

$$r_i = \sum_{j=1}^{m_i} r(P_{ij}).$$

This represents the total amount of the time requirement of type i tasks that may be processed in unit time. Note that if the processors are equally fast, then $r_i = m_i$.

Let f_i denote the rate of the fastest processor of type i. That is, $f_i = \max\{r(P_{ij}) : 1 \leqq j \leqq m_i\}$. Similarly, s_i denotes the rate of the slowest processor of type i.

In order to analyze list schedules on processors of different speeds the concept of height of a task must be modified to take into account the speeds of the processors. In a manner analogous to § 3, we would like to be able to say that the total amount of time spent on height reducing intervals is at most the height of the graph. Thus it is convenient to have the height reduced at least one unit of height per unit time during height reducing intervals.

The *height length* of a task T (of type i) is $\mu(T)/s_i$. Thus even if P_{im_i} (the slowest processor of type i) processes T, it executes one unit of the height length of T per unit time. The length of a chain C, the height of a task T, and the height of $(\mathcal{T}, <, \mu, \nu)$ are defined as usual, except that summations are taken of height lengths of the tasks instead of the time requirements of tasks.

As above, μ_i denotes the time requirement of all type i tasks, and h denotes the height of $(\mathcal{T}, <, \mu, \nu)$. There is some chain of tasks whose length equals h. Let c_i denote the sum of the time requirements of type i tasks along this chain. Then $h = (c_1/s_1) + \cdots + (c_k/s_k)$.

6. Performance bounds for list schedules on uniform nonidentical processors. Following the general outline of § 3, we obtain lower bounds on the performance of an optimal schedule and an upper bound on the performance of any list schedule. The main result of this section is that the worst case performance of list schedules is approximately $k + \max_i (f_i/s_i)$.

THEOREM 2. *Let $(\mathcal{T}, <, \mu, \nu)$ be a k type task system to be scheduled on a set of uniform nonidentical processors. Then the performance ratio of any list schedule for $(\mathcal{T}, <, \mu, \nu)$ to an optimal schedule for $(\mathcal{T}, <, \mu, \nu)$ is at most $k + (\max_i (f_i/s_i))(1 - \min_i (s_i/r_i))$.*

Proof. To obtain a lower bound on the finishing time of any optimal schedule, note that at most r_i units of the time requirement of type i tasks may be completed in one time unit. Thus a lower bound is given by $\max(\mu_1/r_1, \cdots, \mu_k/r_k)$. Also, consider a chain of tasks of length h. Let c_i be the sum of the time requirements of type i tasks along this chain as above. At any point in time at most one task on this chain is being executed (possibly by a fast processor). Thus, a lower bound on w_{opt} is given by $(c_1/f_1) + \cdots + (c_k/f_k)$.

To obtain an upper bound on list schedules, fix a list schedule S, and let p denote the total duration of height reducing intervals. Note $p \leqq h$ since the height is reduced at least at a rate of one unit of height per unit time (during height reducing intervals).

As in § 3, to determine the total duration of constant height intervals, we first count the amount of time requirement finished during height reducing intervals. Specifically, let h_i denote the total time requirement of type i tasks that is completed during height reducing intervals. Note that $(h_1/s_1) + \cdots + (h_k/s_k) \geqq h$. For consider a height reducing interval during which one task (of type i) was at the greatest height throughout the interval (all height reducing intervals may be partitioned into such intervals). Assume that the height is reduced by a total of l in that interval. Then, ls_i units of the time requirement of that task are completed during that interval. Using this argument for all intervals, the above inequality follows.

At a constant height interval, r_i units of the time requirement of type i tasks are completed, for some i. The reasoning is similar to that of § 3, if there are free processors of every type then the height is being reduced. Using arguments similar to those of § 3, the total time spent on constant height intervals is bounded by $\sum_{i=1}^{k} (\mu_i - h_i)/r_i$.

Let w be the finishing time of an arbitrary list schedule and let w_{opt} be the finishing time of an optimal schedule. A bound on the performance ratio of any list schedule to an optimal schedule is given by:

$$(1)\qquad \frac{w}{w_{\text{opt}}} \leqq \frac{h + (\sum_{i=1}^{k} (\mu_i - h_i)/r_i)}{\max((\mu_1/r_1), \cdots, (\mu_k/r_k), ((c_1/f_1) + \cdots + (c_k/f_k)))}.$$

Separating out the h_i terms from the summation, and using the first k lower bounds on w_{opt} yields:

$$(2)\qquad \frac{w}{w_{\text{opt}}} \leqq k + \frac{h - ((h_1/r_1) + \cdots + (h_k/r_k))}{((c_1/f_1) + \cdots + (c_k/f_k))}.$$

Now let $d = \min_i s_i/r_i$. Then for every value of i, $r_i \leqq (s_i/d)$. By increasing the value of the numerator of the right hand side of (2), one thus obtains:

$$(3)\qquad \frac{w}{w_{\text{opt}}} \leqq k + \frac{h - (d((h_1/s_1) + \cdots + (h_k/s_k)))}{((c_1/f_1) + \cdots + (c_k/f_k))}.$$

Now use $h \leqq ((h_1/s_1) + \cdots + (h_k/s_k))$ together with $h = ((c_1/s_1) + \cdots + (c_k/s_k))$ to obtain:

$$(4)\qquad \frac{w}{w_{\text{opt}}} \leqq k + \frac{(1-d)((c_1/s_1) + \cdots + (c_k/s_k))}{((c_1/f_1) + \cdots + (c_k/f_k))}.$$

Let $q = \max(f_1/s_1, \cdots, f_k/s_k)$, be the greatest quotient between fastest and slowest rates for processors of the same type. Using $s_i \geqq (f_i/q)$ for every i (in the numerator of the right hand side of (4)) yields:

$$(5)\qquad \frac{w}{w_{\text{opt}}} \leqq k + \frac{q(1-d)((c_1/f_1) + \cdots + (c_k/f_k))}{((c_1/f_1) + \cdots + (c_k/f_k))}.$$

From equation (5), it follows immediately that $w/w_{\text{opt}} \leqq k + q(1-d)$, i.e., $w/w_{\text{opt}} \leqq k + (\max_i (f_i/s_i))(1 - \min_i (s_i/r_i))$. □

Note that when the processors of each type are equally fast then $q = 1$ and $d = 1/\max(m_1, \cdots, m_k)$ and the bound matches that of Theorem 1. Also, if $k = 1$, then the bound of $1 + (f_1/s_1)(1 - (s_1/r_1)) = 1 + (f_1/s_1) - (f_1/r_1)$ matches the bound of [12], [13].

7. Achievability results for list scheduling on uniform nonidentical processors. To show that Theorem 2 is almost achievable we combine the construction of § 4 with a construction used in [12], [13]. The result used from [12], [13] is as follows. Fix a set of uniform nonidentical processors $\mathcal{P}$, of one type. Then there are ordinary task systems for $\mathcal{P}$ (with empty precedence relation) that achieve the list schedule upper bound. Specifically, the performance ratio of list schedules to optimal schedules over this set of task systems is arbitrarily close to $1 + (f/s) - (f/r)$ where f is the speed of the fastest processor, s the speed of the slowest and r the total processing power of all of the processors.

Consider the task system of Fig. 2. Diagramming conventions are as in Fig. 1. The notation $\mu = r(P_{ij})$ means that the time required for the task equals the rate of the processor P_{ij}. A node labeled with B denotes a copy of one of the task systems used to obtain the lower bound in [12], [13] with the type of each task in this system being type k. We do not elaborate on the time requirements of tasks in B, but remind the reader that in B there are no precedence constraints. The interpretation of an arrow between two nodes labeled with B indicates a precedence dependence of each task at the destination of the arrow on each task at the source of the arrow. The class of task systems described in the figure is parameterized by the variable n and the class of task systems described in [12], [13]. Let n' denote the time required to execute B using an optimal schedule. Assume without loss of generality that $\max(\{(f_i/s_i)-(f_i/r_i): i = 1, \cdots, k\})$ is achieved by processors of type k.

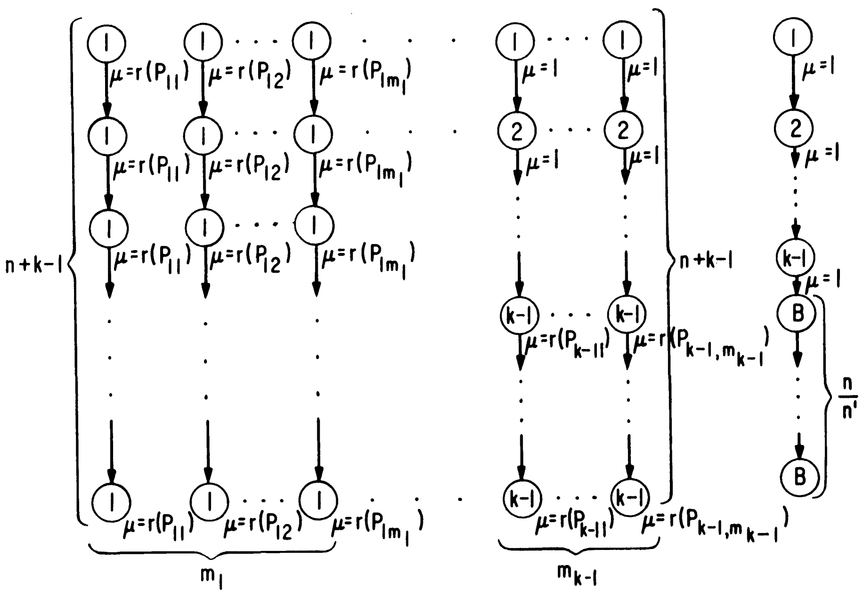


FIG. 2.

An asymptotically optimal schedule first executes the first $k-1$ tasks of each column using an arbitrary list schedule. Then only n more units of time are required. It is clear how to finish the columns that correspond to the first $k-1$ types of tasks in n units of time. By using the optimal schedule for each occurrence of B each occurrence of B requires only n' units of time. Since there are n/n' copies of B only n units of time are required.

A bad list schedule spends $(k-1)n$ units of time completing the tasks that correspond to the first $k-1$ types of processors. It then spends arbitrarily close to $(n/n')(n')(1+(f_k/s_k)-(f_k/r_k))$ units of time to complete the column that corresponds to the kth type of processor using the bad list schedule from [12], [13]. The exact number of steps depends on which task system is used for the nodes labeled with B. The ratio thus approaches $k+(f_k/s_k)-(f_k/r_k)$ for large n and B's for which the performance of list schedules approaches a ratio of $1+(f_k/s_k)-(f_k/r_k)$. □

The gap between our upper and lower bounds on performance ratios is not very large. The gap is between $k+(\max(\{f_i/s_i: i=1,\cdots,k\}))(1-\min(\{s_i/r_i: i=1,\cdots,k\}))$ and $k+\max(\{(f_i/s_i)-(f_i/r_i): i=1,\cdots,k\})$. Since both are between $k+(\max_i(\{f_i/s_i\}))$ and $k+(\max_i(\{f_i/s_i\}))-1$, for all practical purposes the result is tight. Also, if $k=1$ or if processors of each type run at the same rate then the bound of Theorem 2 is achievable.

Also note that the notion of list schedule considered in [12], [13] is the version where whenever tasks become executable, they are assigned to the fastest available processors. Thus, our results are applicable even to the more restrictive notion of list schedule.

8. Summary and other results. We have presented a generalization of the ordinary task systems that are used to model scheduling problems. This generalization is a more effective model of the scheduling problem found on certain types of machines. We have presented tight bounds for list schedules on equally fast processors. Almost tight results have been obtained for typed task systems executed on a set of processors of different speeds.

There are a number of other results for schedules on equally fast typed processors which directly generalize existing results for identical processors. In [2], Coffman and Graham define a label algorithm which always produces an optimal schedule for scheduling a partially ordered set of unit execution time jobs on two identical processors. This algorithm is generalized in [11] for m identical processors, and the generalized algorithm is at most $2-(2/m)$ times worse than optimal. If $\max(m_1,\cdots,m_k)\geqq 2$, then the natural generalization of this algorithm to typed task systems is at most $k+1-(2/\max(m_1,\cdots,m_k))$ times worse than optimal. The proof of this result generalizes the proof of [11], paying special care to types with only one processor. This bound is achievable [14].

The second generalization is related to the level algorithm of Muntz and Coffman [15], [16] for preemptive scheduling of partially ordered tasks. In [11], Lam and Sethi analyze this algorithm for m identical processors, and obtain a performance bound of $2-(2/m)$. Using similar techniques one may generalize this to typed task systems. If $\max(m_1,\cdots,m_k)\geqq 2$, then the level algorithm is at most $k+1-(2/\max(m_1,\cdots,m_k))$ times worse than the optimal preemptive schedule.

In [9], a nonlist scheduling algorithm is given to schedule partially ordered tasks on processors of uniformly different speeds. The algorithm is asymptotic to $\sqrt{m}$ times worse than optimal, independent of the speeds of the processors. Similar speed independent bounds are obtained for typed task systems on processors of uniformly different speeds. In the common case that the processors of a machine are roughly of the same speed, the almost tight bound of this paper is the important bound. However, if there are extremely slow processors, the results of [9] prevent behavior which is as bad as the ratio between the speeds of two different processors of the same type.

Acknowledgment. The author would like to express his thanks to Alan Baratz, Errol Lloyd, Michael Loui, and Albert Meyer for helpful readings of this paper.

REFERENCES

[1] E. G. Coffman, *Computer and Job Shop Scheduling Theory*, John Wiley, New York, 1976.

[2] E. G. Coffman and R. L. Graham, *Optimal scheduling for two-processor systems*, Acta Informatica, 1 (1972), pp. 200–213.

[3] J. B. Dennis, *First Version of a Data Flow Procedure Language*, Lecture Notes in Computer Science 19, G. Goos and J. Hartmanis eds., pp. 362–376; *Symposium on Programming*, Institut de Programmation, Univ. of Paris, Paris, France, April 1974, pp. 241–271; Also MIT LCS TM61, May 1975.

[4] M. R. GAREY AND R. L. GRAHAM, *Bounds for Multiprocessing Scheduling with Resource Constraints*, this Journal, 4, 2 (1975), pp. 187–200.

[5] M. R. GAREY AND D. S. JOHNSON, *Complexity results for multiprocessor scheduling under resource constraints*, Proceedings of the Eighth Annual Princeton Conference on Information Sciences and Systems, 1974.

[6] D. K. GOYAL, *Scheduling processor bound systems*, Proceedings of the Sixth Texas Conference on Computing Systems, 1977.

[7] R. L. GRAHAM, *Bounds on Multiprocessing Timing Anomalies*, SIAM J. Appl. Math. 17 (1969), pp. 263–269.

[8] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Discrete Optimization (1977).

[9] J. M. JAFFE, *Efficient Scheduling of Tasks Without Full Use of Processor Resources*, MIT Laboratory for Computer Science Technical Memo 122, January 1979; Theor. Comput. Sci., to appear.

[10] R. M. KARP AND R. E. MILLER, *Properties of a model for parallel computations: determinacy, termination, queueing*, SIAM J. Appl. Math., 14 (1966), pp. 1390–1411.

[11] S. LAM AND R. SETHI, *Worst case analysis of two scheduling algorithms*, this Journal, 6 (1977), pp. 518–536.

[12] J. W. S. LIU AND C. L. LIU, *Bounds on Scheduling Algorithms for Heterogeneous Computing Systems*, TR No. UIUCDCS-R-74-632 Dept. of Comp. Sci., Univ. of Illinois, June 1974.

[13] ———, *Bounds on Scheduling Algorithms for Heterogeneous Computing Systems*, IFIP74, North Holland, Amsterdam, pp. 349–353.

[14] E. L. LLOYD, private communication.

[15] R. R. MUNTZ AND E. G. COFFMAN JR., *Optimal preemptive scheduling on two-processor systems*, IEEE Trans. Comptrs., C-18, 11 (1969), pp. 1014–1020.

[16] R. R. MUNTZ AND E. G. COFFMAN JR., *Preemptive scheduling of real time tasks on multiprocessor systems*. J. Assoc. Comput. Mach., 17 (1970) 324–338.

[17] J. E. THORNTON, *Design of a Computer—The Control Data 6600*, Scott, Foresman College Division, 1971.

[18] J. D. ULLMAN, *NP-complete scheduling problems*, J. Comput. Systems. Sci., 3 (1975), pp. 384–393.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0008 $01.00/0

RANDOM GRAPHS AND GRAPH OPTIMIZATION PROBLEMS*

BRUCE W. WEIDE†

Abstract. One major difficulty in analyzing algorithms for graph optimization problems is that the probabilistic behavior of the optimum solutions to most of the important problems is generally unknown. We present a general method for relating some well-known results regarding the probability of existence of certain subgraphs in random graphs to the probabilistic behavior of solutions to graph optimization problems, where the problem graphs have edge weights independently chosen from an arbitrary distribution. Application of the technique to well-studied problems such as the traveling salesman problem shows that stronger statements can be made about the optimum solutions than have previously been proved, and that the analysis is straightforward.

Key words. graph theory, random graphs, algorithms, optimization, traveling salesman problem, *NP*-complete problems

1. Introduction. All known deterministic algorithms for solving *NP*-hard problems require computing times not bounded by any polynomial in the size of the problem description, and it seems fairly safe to say that the prospects for showing the existence of polynomial-time algorithms for such problems are dim. Primarily for this reason, recent efforts have been directed toward polynomial-time approximation algorithms for problems which seem computationally intractable with the requirement of exact solution.

Unfortunately, it has recently been shown that even producing a good approximate answer to certain problems is *NP*-hard. For instance, approximating the chromatic number of a graph to within a factor of two is essentially as difficult as finding the exact chromatic number [6]. This fact, combined with the apparently high success rate of heuristic approaches to solving certain difficult problems in practice, led Karp [9] to undertake a more serious investigation of probabilistic approximation algorithms. We will call a *probabilistic approximation algorithm* a procedure that with high probability, but not necessarily always, produces a solution which is very close to the exact answer.

Karp [9] has suggested measuring the success of such an algorithm by the *stochastic convergence* (to zero) of the sequence of errors produced by the algorithm under some probabilistic model. In our case, the problems (rather than the steps taken by the algorithm) will be randomly chosen.

Specifically, we say that a predicate Q is true *almost surely* if, for a sequence of problems $P_1, P_2, \cdots$, one of each size (i.e., problem P_n is of size n) chosen independently and at random according to some distribution, the predicate is false only for finitely many with probability one. We say that Q is true *in probability* if the probability that Q is true tends to one as $n \to \infty$.

A particularly important case is where the predicate depends on the definition of a random variable $X_n = f(P_n)$ (here f is some real-valued function of a problem instance) in the following way: $Q(P_n)$ is "$|X_n| < \varepsilon$." If Q is true almost surely for every fixed $\varepsilon > 0$ then we say that the sequence of random variables $\{X_n\}$ *converges almost surely* to 0, written $X_n \to 0$ (a.s.). Similarly, if Q is true in probability for every fixed $\varepsilon > 0$, then we say $\{X_n\}$ *converges in probability* to 0, and write $X_n \to 0$ (pr.). An algorithm is said to *succeed strongly* (respectively, *weakly*) if the sequence of random variables given by the relative errors of the answers produced by the algorithm converges almost surely (respectively, in probability) to zero. Stochastic convergence of a sequence of random

* Received by the editors March 19, 1979.

† Department of Computer and Information Science, Ohio State University, Columbus, Ohio 43210.
This work was supported in part by the National Science Foundation under Grant MCS-79-12688.

variables to a constant other than zero is defined analogously. It is easy to show that almost sure convergence implies convergence in probability, but not vice-versa; see Chung [3], Weide [14].

Proving strong or weak success of a probabilistic approximation algorithm is no trivial task, but it is considerably facilitated by making use of the following lemma, which we will call the *relative error lemma.*

LEMMA (Weide [14]). *Let X_n be a random variable equal to the value of the exact solution to problem P_n, and let Y_n be a random variable equal to the value of the solution produced by some probabilistic approximation algorithm for problem P_n. Suppose there is a function $g(n)$ and a constant $c > 0$ such that the following two conditions are satisfied*:

(1) $X_n g(n) \to c$ (*a.s.*) (*respectively, pr.*);

(2) $Y_n g(n) \to c$ (*a.s.*) (*respectively, pr.*).

Then the probabilistic approximation algorithm succeeds strongly (*respectively, weakly*); *that is,* $(X_n - Y_n)/X_n \to 0$ (*a.s.*) (*respectively, pr.*).

Using the relative error lemma, the problem of determining the distribution of the relative error of the approximation is overcome, since proving stochastic convergence of suitably normalized forms of the actual answer and the approximation to the same nonzero constant leads immediately to a proof that the algorithm succeeds strongly or weakly. We still face three difficulties: finding a realistic probabilistic model, determining the stochastic behavior of the approximate answers Y_n, and that of the true answers X_n. In practice, the probabilistic model is dictated not so much by what is realistic but by what assumptions result in tractable mathematics in the analysis. Determining the stochastic behavior of the approximate answers produced by an algorithm depends heavily on how complicated the algorithm is. It is the problem of establishing the stochastic behavior of the true solution values that is addressed here.

2. Graph optimization problems and random weighted graphs. Given a graph on n nodes (vertices) with weighted edges, we can characterize a *graph optimization problem* by two features. The first is a set of *feasible solutions*, which consists of all subgraphs of the problem graph that satisfy a set of structural constraints. The second is an *objective function* defined by the weights of edges in each feasible solution, and which is to be minimized over all feasible solutions.

Many classical problems from graph theory and operations research are graph optimization problems. The *traveling salesman problem* (TSP), for instance, has for feasible solutions the set of Hamiltonian circuits of the problem graph, and an objective function which is given by the sum of the edge weights of a feasible solution. The *minimum spanning tree problem* has the set of spanning trees of the problem graph as its feasible solutions, and the same objective function. The *minimum weighted k-clique problem* has a set of feasible solutions consisting of all k-cliques of the problem graph, and again the same objective function. For the *bottleneck traveling salesman problem,* the feasible solutions are once again the Hamiltonian circuits, but the objective function is the maximum edge weight contained in a circuit.

A large number of graph optimization problems are known to be *NP*-complete, including the traveling salesman problem and the bottleneck TSP described above. The maximum weighted k-clique problem, when k is not fixed but is part of the problem description, is also *NP*-complete, as are a whole host of other interesting and useful problems from operations research. For this reason, they are often solved (approximately) by what we have termed probabilistic approximation algorithms, since guaranteed optimum solutions apparently cannot be produced for these problems in a reasonable length of time.

In order to analyze the effectiveness of such algorithms, we must first develop a probabilistic model to describe the origins of problem instances. Since each instance is a graph with weighted edges, a likely candidate for this model is a complete labeled graph with edge weights chosen from some distribution F. It is even possible to account for the absence of a number of edges by allowing F to vary with the number of nodes of the graph, so that a problem instance of size n is the complete graph on n nodes with edge weights chosen independently from the distribution F_n. (For instance, F_n may assign nonzero probability to the weight $+\infty$, which for purposes of minimization problems makes an edge essentially nonexistent.) We call such a graph a *random weighted graph*. While this model might not seem very general, it includes as special cases the models employed by Borovkov [2], Lueker [10], Garfinkel and Gilbert [7], and others who have addressed the problem. They allow only one edge-weight distribution (such as uniform or normal) and/or do not permit it to change as a function of the number of nodes. The model is inadequate for description of Euclidean versions of the problems, since the edge weights must be independent, but the stochastic behavior of many similar problems in that class is essentially already known (see Beardwood, Halton, and Hammersley [1], Papadimitriou [12]).

As mentioned previously, we leave it to the algorithm designer to analyze the stochastic behavior of the solution values produced by his algorithm on such random weighted graphs. Techniques suggested by Borovkov [2], Lueker [10], Garfinkel and Gilbert [7], and Weide [14] could be useful in this regard. Here, we address the problem of determining the stochastic behavior of the true optimum solution values for graph optimization problems when problem instances are the random weighted graphs described above.

3. Relating graph optimization problems to random graphs. Random weighted graphs are basically an unexplored area. However, *random graphs* (where each potential edge of the complete graph on n nodes is present with probability p_n) have been a topic of considerable interest ever since their introduction by Erdös and Rényi [4]. Investigations of random graphs have centered on the probability of the existence of subgraphs satisfying certain properties as a function of p_n. One of the earliest results, due to Erdös and Rényi [4], is that the probability that a random graph with edge probability $p_n = (c + \ln n)/n$ is connected is asymptotic to $e^{-e^{-c}}$. One can easily prove that if $p_n \leqq a(\ln n/n)$ for some $a < 1$ and for all sufficiently large n, then a random graph with edge probability p_n *is not* connected with probability tending to one. On the other hand, if $p_n \geqq a(\ln n/n)$ for some $a > 1$ and for all sufficiently large n, then a random graph with edge probability p_n *is* connected with probability tending to one. (That is, in probability, it has a spanning tree.) Other results include a theorem of Pósa [13] showing that there is a constant C such that if p_n exceeds $C(\ln n/n)$ for all sufficiently large n, then the random graph almost surely has a Hamiltonian circuit. Grimmett and McDiarmid [8] give similar results for independent vertex sets and cliques.

The challenge is to relate these known results about the existence of subgraphs satisfying certain structural conditions to the values of optimum solutions to graph optimization problems. For most cases where the objective function is the sum of the edge weights of a feasible solution or the maximum edge weight, the answer is given by Theorem 1, which has been on the verge of discovery for a couple of years (see § 4 below).

THEOREM 1. *Let S_n be the set of feasible solutions to a graph optimization problem for a complete graph with n nodes. Suppose that there exist q_n and p_n such that the following conditions are true:*

(1) *A random labeled graph with edge probability at most q_n almost surely (respectively, in probability) does not contain any member of S_n as a subgraph.*

(2) *A random labeled graph with edge probability at least p_n almost surely (respectively, in probability) contains a member of S_n as a subgraph.*

Let instances of the graph optimization problem be random weighted graphs with edge weights from the distribution F_n. Then:

(1) *If X_n is the value of the optimum solution when the objective function is the maximum edge weight in a feasible solution, then*

$$F_n^{-1}(q_n)^- \leqq X_n \leqq F_n^{-1}(p_n)$$

almost surely (respectively, in probability).

(2) *If all feasible solutions have k_n edges and X_n is the value of the optimum solution when the objective function is the sum of the edge weights in a feasible solution, then*

$$X_n \leqq k_n F_n^{-1}(p_n)$$

almost surely (respectively, in probability).

Proof. For part (1) with "almost surely", consider the subgraph of the problem instance (a complete weighted graph with edge weights chosen from the distribution F_n) consisting of the n nodes along with those edges of weight at most $F_n^{-1}(q_n)^-$. This is just a random labeled graph with edge probability $F_n(F_n^{-1}(q_n)^-) \leqq q_n$, and therefore almost surely does not contain a member of S_n as a subgraph. Hence, there is almost surely no feasible solution to the graph optimization problem having all edges with weight less than $F_n^{-1}(q_n)^-$, which proves that $F_n^{-1}(q_n)^- \leqq X_n$ almost surely. Similarly, the subgraph of the problem instance consisting of the n nodes and those edges with weight at most $F_n^{-1}(p_n)$ is a random labeled graph with edge probability $F_n(F_n^{-1}(p_n)) \geqq p_n$, and therefore almost surely has a member of S_n as a subgraph. This shows that there is almost surely a feasible solution with maximum edge weight at most $F_n^{-1}(p_n)$, so that $X_n \leqq F_n^{-1}(p_n)$ almost surely. The same argument holds with "in probability" substituted for "almost surely".

Part (2) is proved in analogous fashion, the only differences being that there is no similar lower bound for X_n, and that the upper bound is $k_n F_n^{-1}(p_n)$. □

A remark about the simplicity of the proof is in order. The difficulty in characterizing the stochastic behavior of optimum solution values has been dramatically eased by making use of the assumed availability of q_n and p_n. Actual determination of these parameters is, in general, quite difficult. Only the fact that others have already found q_n and p_n for a wide variety of important problems saves us the trouble of having to try to bound X_n by a more direct method. Examples in the next section illustrate the value of this approach.

4. Examples. Our first application of Theorem 1 is to the bottleneck TSP, where the objective is to minimize the maximum edge weight in a Hamiltonian circuit.

THEOREM 2. *Let X_n be the value of the optimum solution to the bottleneck TSP for a random weighted graph with edge weights chosen from the distribution F_n. Then there is a constant $C \geqq 1$ such that*

$$F_n^{-1}(\ln n/n)^- \leqq X_n \leqq F_n^{-1}(C \ln n/n)$$

in probability. The upper bound holds almost surely.

Proof. The theorem follows immediately from Pósa's characterization of when a random undirected graph has a Hamiltonian circuit, from the fact that a graph must be connected in order to have a Hamiltonian circuit, and from Theorem 1. The only difference for the case of directed graphs is a different value of C. □

Garfinkel and Gilbert [7] have recently reported a lower bound on the *expected* value of the optimum solution to a bottleneck TSP for a complete directed graph of $1-\beta(n+1, 1/(n-1))/(n-1)$, which is asymptotic to $\ln n/n$. Furthermore, they show that, with probability tending to one, the optimum value lies between c/n and $((1+\varepsilon)(2/n)^{1/2} \ln n)^{1/2}$. Their argument for the upper bound is actually quite similar to that used in Theorem 1 above, although it uses a weaker result concerning the threshold for existence of a Hamiltonian circuit and therefore fails to provide as good a bound as Theorem 2, which makes use of Pósa's result. The lower bound in Theorem 2 is also much stronger than that presented by Garfinkel and Gilbert, who obtained theirs by a direct probabilistic argument and not by a similar technique.

Applying Theorem 2 to the specific case investigated by Garfinkel and Gilbert where edge weights are uniformly distributed, we find that the ratio of the optimum solution value to $(\ln n/n)$ is between one and some constant $C \geqq 1$ with probability tending to one. For normally distributed edge weights, the ratio of the optimal solution value to $(-\sqrt{2 \ln n})$ converges in probability to one.

The case of the usual TSP is equally interesting. Here, since the objective function is the sum of the edge weights, Theorem 1 does not permit proof of a lower bound on the optimal solution value. However, an upper bound is easily demonstrated.

THEOREM 3. *Let X_n be the value of the optimum solution to the TSP for a random weighted graph with edge weights chosen from the distribution F_n. Then there is a constant $C \geqq 1$ such that*

$$X_n \leqq nF_n^{-1}(C \ln n/n)$$

almost surely.

Proof. Using Pósa's result and Theorem 1, the conclusion follows immediately. □

Again, we have immediately improved upon Lueker's [10] results which showed that for normally distributed edge weights, the ratio of the expected value of the optimal solution value to $(-n\sqrt{2 \ln n})$ converges to one. It is also true that the ratio of the optimal solution value to $(-n\sqrt{2 \ln n})$ is almost surely at most one. Furthermore, Theorem 3 allows similar conclusions to be proved for other distributions. In a subsequent version of his 1978 paper (submitted for publication to this journal), Lueker has extended his results from the normal distribution and expected values to arbitrary distributions and stochastic convergence, using a relationship between subgraph existence and optimization problems very much like Theorem 1. This new paper includes an almost sure lower bound on X_n that matches the upper bound of Theorem 3 for normally distributed edge weights.

How can these theorems be related to probabilistic approximation algorithms? Consider random weighted graphs with edge weights drawn from a special type of distribution called a *fixed-cost distribution* having the following properties:

(1) There exists some $A = \sup\{x : F(x) = 0\} > 0$.

(2) F can be expanded as $F(x) = F(A) + c(x-A)^\delta(1+o(1))$, for some $\delta > 0$, as $x \to A^+$.

(3) F is a distribution with finite mean and variance.

Intuitively, every edge in such a graph has a weight consisting of two components. One component is a positive fixed cost A for traversing that edge in a traveling salesman tour, and the other is a random non-negative variable cost. Weide [14] has shown that for such graphs, the standard greedy algorithm for the TSP produces a tour of length T_n for which T_n/n converges almost surely to A. It is easy to show from Theorem 3 that the lemma, then, we can prove that the greedy algorithm succeeds strongly for graphs with edge weights chosen from a fixed-cost distribution.

5. Conclusions. Many other similar results can be proved for graph optimization problems having feasible solutions which can be characterized by the parameters q_n and p_n of Theorem 1. These include the minimum spanning tree problem, for which the feasible solutions are all spanning trees and the appropriate graph property is connectedness; the minimum weighted k-clique problem, for which the corresponding graph property is the existence of a k-clique; minimum weighted matching, for which the property is the existence of a matching; and so forth. Relatively simple probabilistic approximation algorithms for such problems come to mind immediately, and some can be analyzed using methods similar to those employed by Borovkov [2], Karp [9], Lueker [10], Garfinkel and Gilbert [7], and Weide [14]. Theorem 1 here may facilitate proofs that such simple algorithms succeed stochastically under certain not-too-restrictive conditions.

It is, of course, possible to prove theorems similar to Theorem 1 for random weighted graphs having weights associated with their vertices rather than, or in addition to, their edges. An interesting problem is to extend part (2) of Theorem 1 to include an almost sure lower bound on the optimal solution value when the objective function is the sum of the edge weights. As mentioned before, Lueker has already made considerable progress in this direction.

Acknowledgments. T. Nishizeki first made a crucial observation leading to Theorem 1. Many discussions with Michael Shamos, Jon Bentley and particularly Bill Eddy are also gratefully acknowledged. One of the anonymous referees made several very insightful suggestions which significantly improved my own understanding of some earlier work and, I hope, the presentation of this work.

REFERENCES

[1] J. BEARDWOOD, J. H. HALTON AND J. M. HAMMERSLEY, *The shortest path through many points*, Proc. Camb. Philos. Soc., 55 (1959), 4 pp. 299–327.

[2] A. A. BOROVKOV, *A probabilistic formulation of two economic problems*, Dokl. Akad. Nauk SSSR, 146 (1962), 5 pp. 983–986 = Sov. Math. Dokl., 3 (1962), 5 pp. 1403–1406.

[3] K. L. CHUNG, *A Course in Probability Theory*, 2nd Ed., Academic Press, New York, 1974.

[4] P. ERDÖS AND A. RÉNYI, *On random graphs I.* Publ. Math., 6 (1959), pp. 290–297.

[5] P. ERDÖS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[6] M. R. GAREY AND D. S. JOHNSON, *The complexity of near-optimal graph coloring*, J. Assoc. Comput. Mach., 23 (1976), pp. 43–49.

[7] R. S. GARFINKEL AND K. C. GILBERT, *The bottleneck traveling salesman problem: algorithms and probabilitistic analysis*, Ibid., 25 (1978), pp. 435–448.

[8] G. R. GRIMMETT AND C. J. H. MCDIARMID, *On colouring random graphs*, Math. Proc. Cambridge Philos. Soc., 77 (1975), pp. 313–324.

[9] R. M. KARP, *The probabilistic analysis of some combinatorial search algorithms*, Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 1–19.

[10] G. S. LUEKER, *Maximization problems on graphs with edge weights chosen from a normal distribution*, Proc. 10th Annual Symp. on Theory of Comp., ACM, New York, 1978, pp. 13–18.

[11] J. W. MOON, *Almost all graphs have a spanning cycle*, Canad. Math. Bull., 15 (1972), pp. 39–41.

[12] C. PAPADIMITRIOU, *The probabilistic analysis of matching heuristics*, Proc. 15th Annual Allerton Conf. on Comm., Control, and Comp., Univ. of Ill., Urbana–Champaign, Sept. 28–30, 1977, pp. 368–378.

[13] L. PÓSA, *Hamiltonian circuits in random graphs*, Discrete Math., 14 (1976), pp. 359–364.

[14] B. W. WEIDE, *Statistical Methods in Algorithm Design and Analysis*, Ph.D. thesis, Department of Computer Science, Carnegie–Mellon Univ., Pittsburgh, PA, 1978, CMU-CS-78-142.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0009 $01.00/0

GENERATING ALL MAXIMAL INDEPENDENT SETS: NP-HARDNESS AND POLYNOMIAL-TIME ALGORITHMS*

E. L. LAWLER†, J. K. LENSTRA‡ AND A. H. G. RINNOOY KAN¶

Abstract. Suppose that an independence system $(E, \mathcal{I})$ is characterized by a subroutine which indicates in unit time whether or not a given subset of E is independent. It is shown that there is no algorithm for generating all the K maximal independent sets of such an independence system in time polynomial in $|E|$ and K, unless $\mathcal{P} = \mathcal{NP}$. However, it is possible to apply ideas of Paull and Unger and of Tsukiyama et al. to obtain polynomial-time algorithms for a number of special cases, e.g. the efficient generation of all maximal feasible solutions to a knapsack problem. The algorithmic techniques bear an interesting relationship with those of Read for the enumeration of graphs and other combinatorial configurations.

Key words. independence system, satisfiability, maximality test, lexicography test, set packing, clique, complete k-partite subgraph, knapsack problem, on-time set of jobs, inequality system, facet generation, matroid intersection

1. Introduction. Let E be a finite set of elements and let $\mathcal{I}$ be a nonempty family of subsets of E satisfying a single axiom: if $I \in \mathcal{I}$ and $I' \subseteq I$, then $I' \in \mathcal{I}$. Under these conditions, $(E, \mathcal{I})$ is said to be an *independence system* and $\mathcal{I}$ is its family of *independent sets*. An independent set I is said to be *maximal* if there is no $I' \in \mathcal{I}$ such that $I' \supset I$. The subsets of E that are not contained in $\mathcal{I}$ are *dependent sets*. A dependent set J is called *minimal* if $J' \in \mathcal{I}$ for each $J' \subset J$.

Suppose that $|E| = n$ and that $(E, \mathcal{I})$ is characterized by a computer subroutine which indicates in unit time whether or not a given subset of E is an independent set. All independent sets can be generated in $O(n|\mathcal{I}|)$ time: given an independent set, $O(n)$ applications of the subroutine suffice to determine the next independent set in a lexicographic listing. But suppose that one is interested only in all the maximal independent sets, of which there are K, $K \leqq |\mathcal{I}|$. These can be found in time polynomial in n and K only in the unlikely event that $\mathcal{P} = \mathcal{NP}$, as we show in § 2.

There are, however, a number of special types of independence systems for which it is possible to generate all the maximal independent sets efficiently. In § 3, an analysis of a procedure due to Paull and Unger [5] reveals that there is a polynomial-time algorithm for this purpose, provided that a certain subproblem can be solved in polynomial time. Improvements in running time and storage requirements suggested by Tsukiyama et al. [8] are discussed as well. In § 4, we investigate some of these independence systems. Typical of these special cases is the problem of generating all the maximal feasible solutions to a knapsack problem. In § 5, we examine the relationship between our approach and a technique for the enumeration of graphs and other combinatorial configurations, recently proposed by Read [6].

2. Complexity. We shall show that the problem of generating all the K maximal independent sets of an arbitrary independence system is NP-hard, i.e., if there is an algorithm for the problem which runs in time polynomial in n and K, then there is a polynomial-time algorithm for solving the satisfiability problem [2].

* Received by the editors May 16, 1978. This research was partially supported by the National Science Foundation under Grant MC.S 76-17605, and by NATO under Special Research Grant 9.2.02 (SRG. 7).

† Computer Science Division, University of California, Berkeley, California 94720.

‡ Mathematisch Centrum, Amsterdam, The Netherlands.

¶ Erasmus University, Rotterdam, The Netherlands.

Let $F(X_1, \cdots, X_N)$ be a Boolean expression in conjunctive normal form. Let $E = \{T_1, F_1, \cdots, T_N, F_N\}$, and for any $j \in \{1, \cdots, N\}$ and any $J \subseteq E$, define

$$x_j(J) = \begin{cases} \textit{true} & \text{if } T_j \in J, F_j \notin J, \\ \textit{false} & \text{if } F_j \in J, T_j \notin J, \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

Let $I \in \mathscr{I}$ if either

(i) there exists a $j \in \{1, \cdots, N\}$ such that both $T_j \notin I$, $F_j \notin I$, or

(ii) each clause of F contains a letter X_j or $\bar{X}_j$ whose defined value is *true*, i.e., $F(x_1(I), \cdots, x_N(I)) = \textit{true}$.

It is easily seen that $(E, \mathscr{I})$ is an independence system. Moreover, F is not satisfiable if and only if the only maximal independent sets are $E - \{T_j, F_j\}$ for $j = 1, \cdots, N$.

Assume there exists a general procedure for generating all the maximal independent sets of an arbitrary independence system with running time $\phi(n, K)$, where ϕ is a polynomial function of n and K. Apply this procedure to the independence system defined above and allow it to run for time $\phi(2N, N)$. Then F is satisfiable if and only if either

(i) $F(x_1(I), \cdots, x_N(I)) = \textit{true}$ for some generated I, or

(ii) the procedure fails to halt within the allotted time, establishing that there are more than N maximal independent sets.

For any given $J \subseteq E$, the conjunctive normal form can be evaluated in time proportional to its length. Appropriate modification of the unit-time assumption for independence testing thus establishes that the procedure solves the satisfiability problem in polynomial time. Since the latter problem is *NP*-complete, it can be solved in polynomial time if and only if $\mathscr{P} = \mathscr{NP}$ [2]. Hence, we have the following theorem.

THEOREM 1. *If there exists an algorithm for generating all the maximal independent sets of an arbitrary independence system in time polynomial in n and K, then* $\mathscr{P} = \mathscr{NP}$.

To obtain a reduction to, rather than from, the satisfiability problem, we now consider the problem of generating all maximal independent sets and all minimal dependent sets of an independence system. Let there be L such sets. We shall show that if there is a polynomial-time algorithm for the satisfiability problem, then there is an algorithm for generating all these sets in time polynomial in n and L. Each step of the latter algorithm yields a new set on the list.

Suppose then, that at a certain point sets $I_1, \cdots, I_l$ have been generated. Let $\mathscr{L} \subseteq \{1, \cdots, l\}$ indicate the generated sets which are maximal independent and $\bar{\mathscr{L}} = \{1, \cdots, l\}$-$\mathscr{L}$ those which are minimal dependent. Any new set I must satisfy $I \not\subseteq I_i$ for all $i \in \mathscr{L}$ and $I_i \not\subseteq I$ for all $i \in \bar{\mathscr{L}}$. Form the Boolean expression

$$(\bigwedge_{i \in \mathscr{L}} \bigvee_{j \notin I_i} X_j) \wedge (\bigwedge_{i \in \bar{\mathscr{L}}} \bigvee_{j \in I_i} \bar{X}_j).$$

The length of this expression is $O(nl)$ and by our assumption one can determine if it is satisfiable in $\psi(nl)$ time, for some polynomial function ψ. If the expression is not satisfiable, then $l = L$ and the algorithm terminates. Otherwise, construct a truth assignment in polynomial time, by successively fixing the value of each variable and determining if the reduced expression is satisfiable. Next define $I = \{j | X_j = \textit{true}\}$ and test I for independence in unit time. If I is independent, augment it until a maximal independent set results; if I is dependent, remove elements until a minimal dependent set is found. Either procedure requires $O(n)$ time. Since clearly $I \neq I_i$ for $i = 1, \cdots, l$, I is the new set on the list. We thus have the following theorem.

THEOREM 2. *If $\mathcal{P} = \mathcal{NP}$, then there exists an algorithm for generating all the maximal independent sets and all the minimal dependent sets of an arbitrary independence system in time polynomial in n and L.*

3. An algorithm.

3.1. A generalized Paull–Unger procedure. We now assume that $E = \{1, \cdots, n\}$ and that independence testing requires time c. Let $\mathcal{I}_j$ be the family of all independent sets that are maximal within $\{1, \cdots, j\}$. By definition, $\mathcal{I}_o = \{\emptyset\}$. We seek to construct $\mathcal{I}_j$ from $\mathcal{I}_{j-1}$ in order to obtain $\mathcal{I}_n$, the family of all K independent sets that are maximal within E.

Suppose that $I \in \mathcal{I}_{j-1}$. If $I \cup \{j\} \in \mathcal{I}$, then clearly $I \cup \{j\} \in \mathcal{I}_j$. If $I \cup \{j\} \notin \mathcal{I}$, then $I \in \mathcal{I}_j$. It follows that

$$|\mathcal{I}_0| \leqq |\mathcal{I}_1| \leqq \cdots \leqq |\mathcal{I}_n| = K.$$

Observing that the elbments of E can be numbered arbitrarily, we obtain the following result.

THEOREM 3. *For any $J \subseteq E$, the number of independent sets maximal within J does not exceed K.*

Suppose that $I' \in \mathcal{I}_j$ and $j \in I'$. Since $I' - \{j\}$ is independent and included in $\{1, \cdots, j-1\}$, there must be some $I \in \mathcal{I}_{j-1}$ such that $I' - \{j\} \subseteq I$. Moreover, I' is an independent set that is maximal within $I \cup \{j\}$. This observation suggests the following procedure to obtain $\mathcal{I}_j$ from $\mathcal{I}_{j-1}$, which is a generalization of an algorithm due to Paull and Unger [5].

Step 1. For each $I \in \mathcal{I}_{j-1}$, find all independent sets I' that are maximal within $I \cup \{j\}$.

Step 2. For each such I', test I' for maximality within $\{1, \cdots, j\}$. Each set I' that is maximal within $\{1, \cdots, j\}$ is a member of $\mathcal{I}_j$, and we have seen that each member of $\mathcal{I}_j$ can be found in this way. However, a given $I' \in \mathcal{I}_j$ may be obtained from more than one $I \in \mathcal{I}_{j-1}$. In order to eliminate duplications, we need one further step.

Step 3. Reject each I' that passes the maximality test if it appears among the sets already found to be in $\mathcal{I}_j$. Suppose that in Step 1, for each $I \in \mathcal{I}_{j-1}$, at most K' sets I' are found in time c'; by Theorem 3, we have $K' \leqq K$. For each I', the maximality test in Step 2 requires $O(nc)$ time, and the duplication test in Step 3 can be accomplished with $O(K)$ pairwise set comparisons, each of which requires $O(n)$ time. It follows that, for fixed j, $O(c'K)$ time suffices for the first step, $O(ncKK')$ time for the second step, and $O(nK^2K')$ time for the third step. Thus, the overall running time to obtain $\mathcal{I}_n$ is $O(nc'K + n^2cKK' + n^2K^2K')$. This yields the following theorem.

THEOREM 4. *All the maximal independent sets of an independence system can be generated in time polynomial in n, c and K, if it is possible to list in polynomial time all independent sets that are maximal within $I \cup \{j\}$, for arbitrary $I \in \mathcal{I}_{j-1}$, $j = 1, \cdots, n$.*

In § 4, we investigate several cases in which the subproblem referred to in Theorem 4 (the "$I \cup \{j\}$ problem") can be solved in polynomial time.

3.2. Improvements of Tsukiyama et al. A technique suggested by Tsukiyama et al. [8] enables one to eliminate duplications more efficiently. It yields significant improvements in both running time and storage requirements of the Paull–Unger procedure.

Instead of comparing a set I' with all members of $\mathcal{I}_j$ found previously, one retains I' only if it is obtained from the *lexicographically smallest* $I \in \mathcal{I}_{j-1}$ from which it can be produced. Hence Step 3 is modified in the following way.

Step 3′. For each I' obtained from $I \in \mathcal{I}_{j-1}$ that is maximal within $\{1, \cdots, j\}$, test for each $i<j$, $i \notin I$, the set $(I' - \{j\}) \cup (I \cap \{1, \cdots, i-1\}) \cup \{i\}$ for independence. Reject I' if any of these tests yields an affirmative answer.

If, indeed, any affirmative answer is obtained, then $I' - \{j\}$ is included in an independent set that is lexicographically smaller than I, and hence in a lexicographically smaller maximal independent set from $\mathcal{I}_{j-1}$.

For each I', the lexicography test in Step 3′ requires $O(nc)$ time, which is the same as required by the maximality test in Step 2. Hence, the overall running time of the revised procedure is $O(nc'K + n^2cKK')$.

Possibly of even greater interest for some applications is the fact that storage requirements can be greatly reduced by organizing the computation as a depth-first search of a tree. Nodes at level j correspond to members of $\mathcal{I}_j$, with the tree rooted at $\emptyset$, the unique member of $\mathcal{I}_0$. Since for each $I \in \mathcal{I}_{j-1}$, either $I \cup \{j\} \in \mathcal{I}_j$ or $I \in \mathcal{I}_j$, each node has at least one and at most K' children. Whenever in the depth-first search a member of $\mathcal{I}_n$ is encountered, it is outputted. The maximum number of subproblems that must be maintained in stack to allow backtracking is $O(nK')$. A further decrease in storage requirements can be obtained at the expense of an increase in running time.

4. Applications. In this section we investigate various independence systems for which all maximal independent sets can be generated in polynomial time.

4.1. Set packing. Let S be a finite set with $|S| = m$ and let $\mathcal{S} = \{S_1, \cdots, S_n\}$ be a family of (not necessarily distinct) subsets of S. A subfamily $I \subseteq \mathcal{S}$ is a *packing* in S if the sets in I are pairwise disjoint. The packings correspond to the independent sets of an independence system with $E = \mathcal{S}$. All maximal packings can be generated in polynomial time, as shown below.

First consider the "$I \cup \{j\}$ problem". Let $A_j \subseteq \mathcal{S}$ consists of the sets S_i for which $S_i \cap S_j \neq \emptyset$. Given $I \in \mathcal{I}_{j-1}$, the only sets which can possibly be maximal within $I \cup \{S_j\}$ are I itself and $(I - A_j) \cup \{S_j\}$. Thus $K' \leqq 2$. It follows that, given A_j, the $I \cup \{j\}$ problem can be solved in $O(n)$ time.

Assuming the sets S_i are specified by ordered lists of indices, one can find the sets $A_1, \cdots, A_n$ in $O(mn^2)$ time. It follows that Step 1 requires $O(mn^2 + n^2K)$ time.

The maximality test for I' is equivalent to verifying that $I' \cap A_i \neq \emptyset$ for all $i<j$, $S_i \notin I$. Since each such test can be carried out in $O(n^2)$ time, Step 2 requires $O(n^3K)$ time.

The lexicography test is easily seen to be equivalent to verifying that $[I - (A_j \cap \{S_{i+1}, \cdots, S_{j-1}\})] \cap A_i \neq \emptyset$ for all $i<j$, $S_i \notin I$. Thus, Step 3′ requires $O(n^3K)$ time as well.

It follows that the overall running time of the procedure is $O(mn^2 + n^3K)$. Since it is possible to implement the search tree in $O(n)$ space, $O(mn)$ space is sufficient overall.

Suppose $\mathcal{S}$ is induced by an undirected m-edge n-vertex graph G with edge set S. S_j denotes the set of edges incident to vertex j and A_j denotes the set of vertices adjacent to vertex j. Then each packing $I \subseteq \mathcal{S}$ is an *independent* or *stable set* of vertices of G, or, equivalently, a *clique* of the complementary graph $\bar{G}$. It was in this context that the Paull–Unger procedure and the improvements of Tsukiyama et al. were originally proposed.

For the graph problem, it is natural for the sets A_j to be given as input in the form of ordered lists. Under this assumption, and noting that $\sum_{j=1}^{n} |A_j| = 2m$, one can reduce the time bound to $O(mnK)$ and the space bound to $O(m+n)$, as shown in [8].

4.2. Complete k-partite subgraphs. Let G be an undirected graph with vertex set $V = \{v_1, \cdots, v_n\}$ and edge set S with $|S| = m$. A *complete k-partite subgraph* of G is defined by a collection $\{V_1, \cdots, V_k\}$ of pairwise disjoint subsets of V such that $\{v_i, v_j\} \in S$ for $v_i \in V_g$, $v_j \in V_h$, if and only if $g \neq h$. Note that an independent set of vertices defines a complete 1-partite subgraph and that a complete k'-partite subgraph is also a complete k-partite subgraph for $k = k'+1, \cdots, n$.

The complete k-partite subgraphs of G correspond to the independent sets of the following independence system. Let $E = V$ and let $I \in \mathcal{I}$ if there exists a partition $P(I) = \{V_1, \cdots, V_k\}$ of I (i.e., $\bigcup_{h=1}^{k} V_h = I$ and $V_g \cap V_h = \emptyset$ for $1 \leqq g < h \leqq k$) that defines a complete k-partite graph on I. We will show how to generate all maximal complete k-partite subgraphs of G in polynomial time.

Again consider the "$I \cup \{j\}$ problem". Let $P(I) = \{V_1, \cdots, V_{k'}\}$ with $V_h \neq \emptyset$ $(h = 1, \cdots, k')$ and $k' \leqq k$.

First, suppose that $\{v_i, v_j\} \in S$ for all $v_i \in I$. If $k' < k$, then the single independent set I' that is maximal within $I \cup \{v_j\}$ is $I \cup \{v_j\}$ itself, with $P(I \cup \{v_j\}) = P(I) \cup \{v_j\}$. If $k' = k$, then there are $k+1$ sets I', for which $P(I')$ is obtained by deleting any one of the members of $P(I) \cup \{v_j\}$.

Suppose now that $\{v_i, v_j\} \in S$ only for all $v_i \in V'_h \subseteq V_h$ $(h = 1, \cdots, k')$, where $V'_h = \emptyset$ for $h = 1, \cdots, a$, $\emptyset \subset V'_h \subset V_h$ for $h = a+1, \cdots, b$ and $V'_h = V_h$ for $h = b+1, \cdots, k'$, with $0 \leqq a \leqq b \leqq k'$ and $b > 0$. In this case, $b+1$ independent sets I' that are maximal within $I \cup \{v_j\}$ are defined by $P(I') = P(I)$ and $P(I') = \{V'_1, \cdots, V'_{h-1}, (V_h - V'_h) \cup \{v_j\}, V'_{h+1}, \cdots, V'_b, V_{b+1}, \cdots, V_{k'}\}$ for $h = 1, \cdots, b$. In the special case that $a = 0$, even more sets I' may exist. If $k' < k$, then the single additional set I' is defined by $P(I') = \{V'_1, \cdots, V'_b V_{b+1}, \cdots, V_{k'}, \{v_j\}\}$. If $k' = k$, then there are $k - b$ additional set I', for which $P(I')$ is obtained by deleting any one of the sets $V_{b+1}, \cdots, V_{k'}$ from $\{V'_1, \cdots, V'_b, V_{b+1}, \cdots, V_{k'}, \{v_j\}\}$. (Note that these sets are not maximal in the case that $a > 0$.)

Since $K' = O(k)$ and independence testing requires $O(m)$ time, the overall running time of the procedure is $O(n^2 mkK)$.

4.3. Knapsack problems. Next consider the *knapsack inequality* $\sum_{j=1}^{n} a_j x_j \leqq b$, $x_j \in \{0, 1\}$ $(j = 1, \cdots, n)$, where $a_1 \geqq a_2 \geqq \cdots \geqq a_n > 0$. The feasible solutions to this inequality correspond in a natural way to the independent sets of an independence system with $E = \{1, \cdots, n\}$ and $I \in \mathcal{I}$ if $\sum_{j \in I} a_j \leqq b$. We are interested in generating all maximal feasible solutions.

Consider the $I \cup \{j\}$ problem and assume that $I \cup \{j\} \notin \mathcal{I}_j$. Feasibility is restored by removing any element h from $I \cup \{j\}$. Thus $K' \leqq j$, and the $I \cup \{j\}$ problem can be solved in $O(n)$ time.

For a given $I \in \mathcal{I}_{j-1}$, define $m(h) = \max \{i \mid i < h, i \notin I\}$; let $a_{\max\emptyset} = \infty$. A set $I' = (I - \{h\}) \cup \{j\}$ $(h \in I)$ passes the maximality test if and only if $\sum_{i \in I'} a_i + a_{m(j)} > b$, and it passes the lexicography test if and only if $\sum_{i \in I} a_i - a_h + a_{m(h)} > b$. Moreover, for *all* I' arising from $I \cup \{j\}$, these tests can be carried out in $O(n)$ time altogether. It follows that the overall running time of the procedure is $O(n^2 K)$.

The *unbounded* knapsack inequality, in which the x_j are allowed to take on any nonnegative integer value, is reducible to the 0-1 case by introducing $2a_j, 4a_j, \cdots, 2^k a_j$ into the problem in addition to a_j, where k is the smallest integer such that $2^{k+1} a_j > b$. Then E contains $O(n \log b)$ elements, and the algorithm is still strictly polynomial.

4.4. On-time sets of jobs. Suppose there are *n jobs* to be processed, one at a time, by a single *machine* starting at time 0. Job j requires an uninterrupted *processing time* of p_j units and has a *deadline* d_j. Let $E = \{1, \cdots, n\}$ and let $I \in \mathcal{I}$ if all the jobs in I can be scheduled for completion by their deadlines. It is well known that such a schedule exists if and only if the jobs in I are all completed on time when sequenced in order of nondecreasing deadlines. Hereafter, assume $d_1 \leqq d_2 \leqq \cdots \leqq d_n$.

Again consider the $I\cup\{j\}$ problem and assume that $I\cup\{j\}\notin\mathcal{I}_j$. In this case, we have $\sum_{i\in I}p_i+p_j>d_j$. Independence is restored by removing job j from $I\cup\{j\}$ or by removing some jobs from I such that job j, which can be assumed to remain in the last position, is completed on time. It follows that solving the $I\cup\{j\}$ problem is equivalent to finding all maximal subsets $H\subseteq I$ such that $\sum_{i\in H}p_i\leqq d_j-p_j$, which can be accomplished by applying the knapsack procedure of § 4.3. By Theorem 3, the number of maximal subsets H does not exceed $K-1$. Hence the $I\cup\{j\}$ problem can be solved in $O(n^2K)$ time.

Since maximality and lexicography tests require $O(n)$ time, it follows that the overall running time of the procedure is $O(n^3K^2)$.

4.5. Inequality systems. The problems considered in §§ 4.1, 4.3 and 4.4 can all be viewed as special instances of the general problem of finding all maximal feasible solutions to an *inequality system* of the form $Ax\leqq b$, $x_j\in\{0,1\}(j=1,\cdots,n)$, where the $m\times n$-matrix $A=(a_{ij})$ and the m-vector $b=(b_i)$ have nonnegative components.

For example, given a set $S=\{1,\cdots,m\}$ and a family $\mathcal{S}=\{S_1,\cdots,S_n\}$ of subsets of S, define $a_{ij}=1$ if $i\in S_j$, $a_{ij}=0$ otherwise. In the case that $b_i=1$ $(i=1,\cdots,m)$, the maximal feasible solutions correspond to the maximal packings in S; they can be generated in polynomial time, as has been shown in § 4.1. In the case that $b_i=\sum_{j=1}^n a_{ij}-1$ $(i=1,\cdots,m)$, the maximal feasible solutions correspond to the complements of the minimal coverings of S. We have not been able to devise a polynomial-time algorithm for this problem. Nor have we been able to obtain an NP-hardness result similar to Theorem 1 for this case or even for a general inequality system, although we conjecture that no polynomial-time algorithm exists unless $\mathcal{P}=\mathcal{NP}$.

For the scheduling problem discussed in § 4.4, we have $m=n$, $a_{ij}=p_j$ if $i\geqq j$, $a_{ij}=0$ otherwise, and $b_i=d_i$ (cf. [4]). The same technique as above can be applied to a slightly wider class of inequality systems, where b is an m-vector with nondecreasing components and A is a nonnegative $m\times n$-matrix such that

(i) $a_{ij}>0$ implies $a_{ij'}>0$ for all $j'<j$, and

(ii) the strictly positive entries in each column are nonincreasing.

In this case, the $I\cup\{j\}$ problem with $I\cup\{j\}\notin\mathcal{I}_j$ can be solved by applying the knapsack procedure of § 4.3 to the constraint of smallest index h such that $a_{hj}>0$. Any maximal subset of $I\cup\{j\}$ that satisfies constraint h will then satisfy the remaining constraints as well.

The reader may be able to construct other examples in which a certain property of A permits one to restrict attention to a single constraint when independence has to be restored. In each such case, the knapsack procedure can be applied to solve the $I\cup\{j\}$ problem in polynomial time.

4.6. Facet generation. Consider the convex hull P of all 0-1 vectors x satisfying the general inequality system $Ax\leqq b$, where $A\geqq 0$. Balas and Zemel [1] have established a correspondence between the *facets* of P and the *minimal covers* of A, i.e. the minimal feasible solutions to $Ax\nleqq b$. Such covers are in one-one correspondence to the maximal feasible solutions to $Ax'\ngtr b'$, where $b_i'=\sum_{j=1}^n a_{ij}-b_i-1$ $(i=1,\cdots,m)$, under the assumption that all data are integers.

Thus, in order to generate the facets of P, it suffices to generate the K maximal feasible solutions to $Ax'\ngtr b'$. This inequality system can be considered as the *disjunction of m* knapsack inequalities $\sum_{j=1}^n a_{ij}x_j'\leqq b_i'$ $(i=1,\cdots,m)$, the ith such inequality having K_i maximal feasible solutions. In the case that $m=1$, the procedure of § 4.3 can be applied to yield all minimal covers in polynomial time. In the general case, the following procedure may have some practical value, even though it is not polynomial in K.

A maximal feasible solution to the entire system has to be feasible and maximal with respect to at least one of the separate inequalities. The procedure of § 4.3 is now applied to each of these inequalities in turn. However, a maximal feasible solution to inequality i is accepted as a maximal feasible solution to $Ax' \not> b'$ only if it is

(i) infeasible for each of the inequalities $1, \cdots, i-1$, and

(ii) infeasible or maximal feasible for each of the inequalities $i+1, \cdots, m$.

It is not hard to see that this procedure generates all minimal covers without duplication.

For inequality i, application of the knapsack procedure requires $O(n^2K_i)$ time, and conditions (i) and (ii) can be checked in $O(mn)$ time for any candidate solution, or in $O(mnK_i)$ time altogether. It follows that the overall running time of the procedure is $O((mn+n^2)\sum K_i)$. Unfortunately, there exist inequality systems for which $\sum K_i$ is exponentially related to K. For example, in the simple case that $m = n-1$, $a_{ij} = 1$, $b'_i = i$ $(i = 1, \cdots, m, j = 1, \cdots, n)$, we have $K_i = \binom{n}{i}$ $(i = 1, \cdots, m)$, $\sum K_i = 2^n - 1$, and $K = n$.

For some special cases, truly polynomial-time algorithms can still be obtained. For example, suppose A is such that the entries in each row are monotone nonincreasing. If $I \cup \{j\} \not\in \mathcal{I}$, then removal of any element from $I \cup \{j\}$ restores feasibility, so that $K' \leqq n$.

In analogy to the above approach, one might view a general inequality system $Ax \leqq b$ as the *conjunction* of m knapsack inequalities. In this case, however, a maximal feasible solution to the entire system can be feasible but nonmaximal with respect to each of the separate inequalities. It seems hard to make any significant progress beyond the special cases discussed in § 4.5.

4.7. Matroid intersections. A matroid $M = (E, \mathcal{I})$ is an independence system such that for all $J \subseteq E$, all independent sets maximal within J have the same cardinality [3]. Given m matroids $M_i = (E, \mathcal{I}_i)$ $(i = 1, \cdots, m)$ with $E = \{1, \cdots, n\}$, their *intersection* $(E, \mathcal{I})$ is an independence system defined by $\mathcal{I} = \bigcap_{i=1}^m \mathcal{I}_i$. We are interested in generating all maximal independent sets in $(E, \mathcal{I})$, assuming that independence testing in M_i requires time c_i $(i = 1, \cdots, m)$.

Consider the $I \cup \{j\}$ problem. If $I \cup \{j\} \not\in \mathcal{I}_i$, then addition of j must have destroyed independence in some of the m matroids, say, in $M_1, \cdots, M_l$. Each of these matroids M_i contains a unique minimal dependent set or *circuit* C_i, and independence in M_i is restored by removing any one element from C_i.

It follows that, in order to solve the $I \cup \{j\}$ problem, it is necessary to find all minimal subsets of $\bigcup_{i=1}^l C_i$ that contain at least one element from each circuit, i.e., all minimal coverings of $(C_1, \cdots, C_l)$. In view of our remark in § 4.5, we settle for a brute force approach: consider all n^m possible solutions. This yields an overall running time of $O(n^{m+2}K \sum c_i)$, which is, at least, polynomial for fixed m.

For certain special cases, e.g. the generation of all spanning trees [7], the special structure of the system can be exploited and significant improvements made.

5. An enumeration procedure of Read. We conclude by noting a relationship between our techniques and those proposed by Read [6] for the enumeration of graphs, digraphs, and other combinatorial configurations. We restate the essential features of Read's procedure in our terms, as follows.

The family $\mathcal{I}_j$ is to be obtained from the family $\mathcal{I}_{j-1}$ by applying an *augmentation operation* to each set in $\mathcal{I}_{j-1}$. These sets are processed in a *canonical linear order* "<" and the augmentation routine produces sets I' from each $I \in \mathcal{I}_{j-1}$ in this same order. For each I′ ∈ $\mathscr{I}_j$, let $f(I')$ denote the first set in $\mathscr{I}_{j-1}$ which produces I' when subjected to the augmentation operation. Suppose that the canonical order is *weakly monotonic* in the sense that for all I', $I'' \in \mathscr{I}_j$, $I' < I''$ implies $f(I') \leqq f(I'')$. Then it is simple to avoid duplications: when applying the augmentation operation, retain the next set produced only if it follows the member of $\mathscr{I}_j$ that has been obtained lastly.

Consider, for example, how this procedure is applied by Read to generate all the nonisomorphic digraphs on five vertices. The nondiagonal elements of the adjacency matrix are written as a string of 20 bits, which can be interpreted as a binary integer. A *canonical* digraph is one which has the largest such integer of all digraphs in its isomorphism class, and this integer is its *code.* Let $\mathscr{I}_{j-1}$ be the family of all canonical digraphs with $j-1$ arcs; their codes specify the canonical linear order. For each $I \in \mathscr{I}_{j-1}$, the augmentation operation produces digraphs I' with j arcs by systematically changing a 0 to a 1 in the 20-bit representation of I. Each such I' is tested for canonicity. Each I' that passes the canonicity test is added to the list $\mathscr{I}_j$ if and only if its code is strictly greater than that of the most recently obtained member of $\mathscr{I}_j$. It can be shown that the property of weak monotonicity is satisfied. Thus, all canonical digraphs with j arcs are generated in this way, without duplication.

We have been unable to devise a weakly monotonic ordering for the problems considered in this paper. The lexicography test of Tsukiyama et al. is, in effect, an alternative to Read's technique for eliminating duplications and amounts to an analysis of the *inverse* of the augmentation operation. That is, when I' is obtained from $I \in \mathscr{I}_{j-1}$, I' is retained only if $f(I') = I$, where $f(I')$ denotes the lexicographically smallest set in $\mathscr{I}_{j-1}$ which produces I' when subjected to the augmentation operation.

REFERENCES

[1] E. Balas and E. Zemel, *All the facets of zero-one programming polytopes with positive coefficients*, Management Sciences Research Report 374, Carnegie–Mellon University, Pittsburgh, 1975.

[2] S. A. Cook, *The complexity of theorem-proving procedures*, Proc. 3rd Annual ACM Symp. Theory Comput., (1971), pp. 151–158.

[3] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[4] E. L. Lawler and J. M. Moore, *A functional equation and its application to resource allocation and sequencing problems*, Management Sci., 16 (1969), pp. 77–84.

[5] M. C. Paull and S. H. Unger, *Minimizing the number of states in incompletely specified sequential switching functions*, IRE Trans. Electron. Comput., EC-8 (1959), 356–367.

[6] R. C. Read, *Every one a winner, or how to avoid isomorphism search when cataloguing combinatorial configurations*, Ann. Discrete Math, 2 (1978), pp. 107–120.

[7] R. C. Read and R. E. Tarjan, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237–252.

[8] S. Tsukiyama, M. Ide, M. Ariyoshi and I. Shirawaka, *A new algorithm for generating all the maximal independent sets*, this Journal, 6 (1977), pp. 505–517.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0010 $01.00/0

BOUNDS ON SELECTION NETWORKS*

ANDREW CHI-CHIH YAO†

Abstract. We investigate the complexity of network selection by measuring it in terms of $U(t, N)$, the minimum number of comparators needed, and $T(t, N)$, the minimum delay time possible, for networks selecting the smallest t elements from a set of N inputs. New bounds on $U(t, N)$ and $T(t, N)$ are presented. In particular, the asymptotic forms of $U(t, N)$ and $T(t, N)$ are determined for any fixed t.

Key words. comparator, complexity, delay time, network, selection, sorting network

1. Introduction. A *(t, N)-selector* where $1 \leqq t < N$ is a network with N inputs $(x_1, x_2, \cdots, x_N)$ and N outputs $(x'_1, x'_2, \cdots, x'_N)$ such that the set $\{x'_1, x'_2, \cdots, x'_t\}$ consists of the smallest t elements of $x_1, x_2, \cdots, x_N$. We consider (t, N)-selectors that are built of basic modules called comparators which are themselves $(1, 2)$-selectors. We shall draw comparators as in Knuth [6] (see Fig. 1). A (2, 5)-selector is shown in Fig. 2.

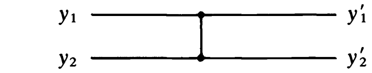


FIG. 1. *A comparator (with* $y'_1 = \min\{y_1, y_2\}$, $y'_2 = \max\{y_1, y_2\}$).

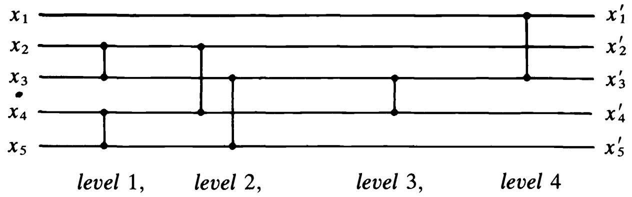


FIG. 2. *A* (2, 5)*-selector.*

Note that the "sorting networks" (for N elements), which have been extensively studied [1], [2], [3], [6], are networks that are (t, N)-selectors for all t $(1 \leqq t < N)$.

In a network the comparators may be grouped into *levels*, where within each level are some comparisons that can be made simultaneously (see Fig. 2). The *delay time* of a network is the minimum number of levels that its comparators can be grouped into.

In this paper, we investigate the complexity of network selection by measuring it in terms of $U(t, N)$, the minimum number of comparators needed in any (t, N)-selector, and $T(t, N)$, the minimum delay time possible for any (t, N)-selector. New bounds on $U(t, N)$ and $T(t, N)$ are presented. In particular, $U(3, N)$ is determined to within a constant of 2, and asymptotic formulae for $U(t, N)$ and $T(t, N)$ are given for fixed t. A new lower bound on the delay time for sorting networks is also obtained. Main results are contained in Theorems 3.1, 3.3, 3.7, 4.3, 4.5.

2. Previous results. The following bounds are known in the literature.

(a) [6]

$$U(1, N) = N - 1, \qquad U(2, N) = 2N - 4. \tag{1}$$

(b) (Due to V. E. Alekseyev, see [6, p. 234].)

$$(N - t)\lceil \log (t + 1)\rceil \leqq U(t, N) \leqq (N - t)\left(1 + \frac{2\hat{S}(t)}{t}\right), \tag{2}$$

* Received by the editors May 3, 1979.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by the National Science Foundation under Grants GJ 41538 and MCS-77-05313.

where $\hat{S}(t)$ is the minimum number of comparators needed in any sorting network with t inputs.[1]

(c) (Due to F. F. Yao, [7].)

$$T(t, N) \geqq \frac{1}{\log 3 - 1}\left(\log t + \log\left(1 - \frac{t}{N}\right)\right). \tag{3}$$

3. New results concerning $U(t, N)$. In this section, we shall show that $U(t, N)$ is well approximated by $\lceil \log (t+1) \rceil N$ when t is small compared to $\sqrt{N}$. A new lower bound for $U(t, N)$, which improves on Alekseyev's bound (2) in most cases, is also derived. As a consequence, $U(3, N)$ is determined to within a constant.

3.1. Asymptotic behavior of $U(t, N)$ for fixed t. Since the best upper bound known for $\hat{S}(t)$ is of the order $t(\log t)^2$ for general t, the inequality that can be deduced from (2) is

$$\lceil \log (t+1) \rceil (N - t) \leqq U(t, N) \leqq C(\log t)^2 N, \tag{4}$$

where C is a constant. Thus, the asymptotic behavior of $U(t, N)$ for fixed t is not well determined by (2). We shall construct (t, N)-selectors that yield a better upper bound for $U(t, N)$. This enables us to identify the leading term of $U(t, N)$. For example, when $t = 11$, we can obtain

$$4(N - 11) \leqq U(11, N) \leqq 4N + C((\log N)^2) \tag{5}$$

for some constant C. In general, we have:

THEOREM 3.1.

$$U(t, N) = \lceil \log (t+1) \rceil N + O\left((\log N)^{\lceil \log((t+1)/3) \rceil}\right) \quad \textit{for fixed } t. \tag{6}$$

To prove Theorem 3.1 we need the following lemma.

LEMMA 3.2.

$$U(t, N) \leqq U(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor) + U(t, \lceil N/2 \rceil + \lfloor t/2 \rfloor) + \lfloor N/2 \rfloor. \tag{7}$$

Proof of Lemma 3.2. We need only show that a (t, N)-selector can be constructed from one $(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor)$-selector, one $(t, \lceil N/2 \rceil + \lfloor t/2 \rfloor)$-selector, and $\lfloor N/2 \rfloor$ additional comparators. Figure 3 shows such a construction. The reason that it works as a (t, N)-selector is that, after the initial $\lfloor N/2 \rfloor$ comparisons in A, at most $\lfloor t/2 \rfloor$ of the smallest t elements can come out on the $\lfloor N/2 \rfloor$ lines of the lower half. These possible

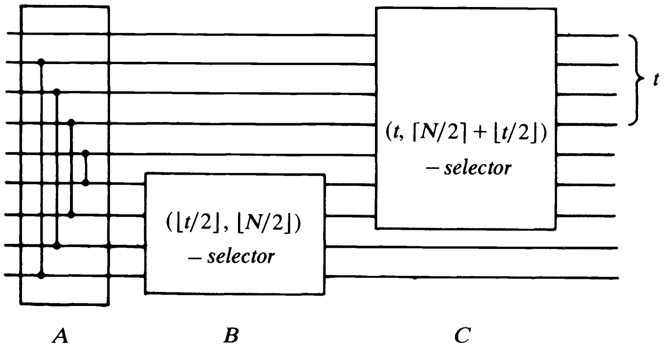

t
$(t, \lceil N/2 \rceil + \lfloor t/2 \rfloor)$
– selector
$(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor)$
– selector
A
B
C

FIG. 3. *A (t, N)-selector.*

[1] Throughout this paper, logarithms are taken with respect to base 2.

"small" elements are selected by B and input into C. Therefore all of the smallest t elements are fed into the $(t, \lceil N/2\rceil + \lfloor t/2\rfloor)$-selector C which finally outputs those elements on the top t output lines. This proves Lemma 3.2. □

Proof of Theorem 3.1. We need the following useful identities.[2] For any integer $t \geqq 3$,

$$\lceil \log (t+1)\rceil = 1 + \lceil \log (\lfloor t/2\rfloor + 1)\rceil, \tag{8}$$

and

$$\left\lceil \log \frac{t+1}{3} \right\rceil = 1 + \left\lceil \log \frac{\lfloor t/2\rfloor + 1}{3} \right\rceil. \tag{9}$$

We shall prove by induction that, for any $t' \geqq 1$,

$$U(t', N) \leqq \lceil \log (t'+1)\rceil N + c_{t'}(\log N)^{\lceil \log ((t'+1)/3)\rceil} \quad \text{for some constant } c_{t'} \text{ and all } N. \tag{10}$$

By (1), statement (10) is true for $t' = 1, 2$. Now, assume that it is true for all $t' < t$ (where $t \geqq 3$); we shall establish (10) for $t' = t$.

$$\text{Let } c' = c_{\lfloor t/2\rfloor},\ \lambda' = \lceil \log ((\lfloor t/2\rfloor + 1)/3)\rceil, \text{ and } \lambda = \lceil \log ((t+1)/3)\rceil.$$

By elementary calculus, $(1-x)^\lambda \leqq 1 - (\lambda x)/2$ for all small enough nonnegative x. Thus there exists a number $N_t \geqq 2(t+2)$ so that $(1 - \log \frac{4}{3}/\log N)^\lambda \leqq 1 - \frac{1}{2}\lambda (\log \frac{4}{3})/\log N$ for all $N \geqq N_t$. Choose c_t such that

$$\lambda c_t > \max \{4c', 4t\lceil \log (t+1)\rceil\}/\log \tfrac{4}{3}, \tag{11}$$

and

$$U(t, N') \leqq \lceil \log (t+1)\rceil N' + c_t(\log N')^\lambda \quad \text{for all } N' \leqq N_t.$$

We wish to show, by induction on N', that

$$U(t, N') \leqq \lceil \log (t+1)\rceil N' + c_t(\log N')^\lambda \quad \text{for all } N'. \tag{12}$$

Let $N > N_t$. Assume that (12) is true for all $N' < N$. Then, by (7) and the induction hypotheses,

$$\begin{aligned} U(t, N) \leqq{}& \lceil \log (\lfloor t/2\rfloor + 1)\rceil \lfloor N/2\rfloor + c'(\log N)^{\lambda'} \\ &+ \lceil \log (t+1)\rceil (\lceil N/2\rceil + \lfloor t/2\rfloor) + c_t\left(\log\left(\frac{N+t}{2} + 1\right)\right)^\lambda + \lfloor N/2\rfloor. \end{aligned}$$

Using (8), (9), (11), and the fact $N \geqq N_t$, we obtain

$$\begin{aligned} U(t, N) &\leqq \lceil \log (t+1)\rceil N + t\lceil \log (t+1)\rceil + c'(\log N)^{\lambda-1} + c_t(\log \tfrac{3}{4}N)^\lambda \\ &= \lceil \log (t+1)\rceil N + c_t(\log N - \log \tfrac{4}{3})^\lambda + c'(\log N)^{\lambda-1} + t\lceil \log (t+1)\rceil \\ &\leqq \lceil \log (t+1)\rceil N + c_t(\log N)^\lambda\left(1 - \frac{\lambda}{2}(\log \tfrac{4}{3})/\log N\right) + c'(\log N)^{\lambda-1} + t\lceil \log (t+1)\rceil \\ &\leqq \lceil \log (t+1)\rceil N + c_t(\log N)^\lambda. \end{aligned}$$

This completes the induction proof of (12).

[2] We give a proof of (8); all the identities involving ceiling and floor functions in this paper can be similarly proved. Let $2^k \leqq t < 2^{k+1}$, then $2^{k-1} + 1 \leqq \lfloor t/2\rfloor + 1 \leqq 2^k$. Both sides of (8) are thus equal to $k+1$. See Knuth [5, § 1.2.4] for more discussions of such identities.

We have completed the induction step in the proof of (10). Theorem 3.1 follows from (10) and Alekseyev's bound (2). □

It is clear that (t, N)-selectors which satisfy the bound of Theorem 3.1 can be explicitly constructed by following the inductive scheme illustrated in Fig. 3, using as bases $(1, N)$-selectors and $(2, N)$-selectors that achieve (1). As will be seen in § 3.3, the $(3, N)$-selectors thus constructed are optimal within a constant number of comparators.

3.2. Other sufficient conditions for $U(t, N) \approx \lceil \log (t+1) \rceil N$. According to Theorem 3.1, $\lceil \log (t+1) \rceil N$ is the dominant term of $U(t, N)$ for fixed t as $N \to \infty$. Actually, this is true under more general situations. We have the following theorem:

THEOREM 3.3. *If $f(N) = O(N^{1/2-\varepsilon})$ for some fixed $0<\varepsilon<\frac{1}{2}$, then*

$$\lim_{N\to\infty} \frac{U(f(N), N)}{N\lceil \log (f(N)+1)\rceil} = 1.$$

COROLLARY 3.4. *If $0<\alpha<\frac{1}{2}$, then*

$$\lim_{N\to\infty} \frac{U(N^{\alpha}, N)}{N \log N} = \alpha.$$

Thus, for any t satisfying $t = O(N^{1/2-\varepsilon})$, $U(t, N)$ is well approximated by $\lceil \log (t+1) \rceil N$. We shall see that Theorem 3.3 (and hence Corollary 3.4) is a consequence of the following theorem and Alekseyev's lower bound $\lceil \log (t+1) \rceil (N-t)$ for $U(t, N)$.

THEOREM 3.5. *For $t<\sqrt{N}$,*

$$U(t, N) \leqq \lceil \log (t+1) \rceil N + 12 \lceil \log (t+2) \rceil^3 \binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil}. \tag{13}$$

The proof of Theorem 3.5 will be given in Appendix A. Obviously, (13) provides an alternative proof of Theorem 3.1. We have proved Theorem 3.1 separately because it has a more elegant proof of its own, also the construction described there often yields better networks than those obtained by the construction used in Appendix A.

Proof of Theorem 3.3. To prove that Theorem 3.3 follows from Theorem 3.5 and Alekseyev's lower bound, it is sufficient to show that, for $t = O(N^{1/2-\varepsilon}) = o(N^{(1-\varepsilon)/2})$,

$$\binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil} = O(N^{1-\delta}) \quad \text{for some constant } \delta > 0.$$

Write $n = \lceil \log N \rceil$, $k = \lceil ((1-\varepsilon)/2) \log N \rceil$, and $\alpha = k/n$. Let $H(x)$ by the *binary entropy function* $(x \log (1/x) + (1-x) \log (1/(1-x)))$. For large N,

$$\begin{aligned}\binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil} &\leqq \binom{n}{k} \\ &= \frac{n!}{k!(n-k)!} \\ &= O\left(\frac{\sqrt{2\pi n}(n/e)^n}{\sqrt{2\pi \alpha n}(\alpha n/e)^{\alpha n}\sqrt{2\pi(1-\alpha)n}((1-\alpha)n/e)^{(1-\alpha)n}}\right) \\ &= O(2^{H(\alpha)n}),\end{aligned}$$

where we have used the monotonicity of $\binom{n}{j}$ for $0 \leqq j \leqq \lfloor n/2 \rfloor$ and Stirling's approximation (see, e.g. [5]). As $H(x)$ increases strictly on $[0, \frac{1}{2}]$, we have for large N,

$H(\alpha)\leqq H(\frac{1}{2}-\frac{1}{4}\varepsilon)<H(\frac{1}{2})=1$ and thus,

$$\binom{\lceil \log N\rceil}{\lceil \log ((t+1)/3)\rceil}=O(2^{(1-\delta)n})$$
$$=O(N^{1-\delta}),$$

where $\delta=1-H(\frac{1}{2}-\frac{1}{4}\varepsilon)$. This proves Theorem 3.3. □

3.3. Lower bounds for $U(t, N)$. Since $U(1,N)=N-1$ and $U(2,N)=2(N-2)$, it is an interesting question whether Alekseyev's lower bound $\lceil\log (t+1)\rceil(N-t)$ is in general achievable for $U(t, N)$. We shall give a new lower bound which shows that, for most values of t, Alekseyev's lower bound cannot be achieved.

THEOREM 3.6. *If t is not a power of 2, we have*

$$U(t,N)\geqq \lceil\log (t+1)\rceil N+(t-2^{\lfloor\log t\rfloor})\log N+C(t)$$

for some function $C(t)$.

Proof. We first prove the theorem for the special case $t=3$, i.e., we will show

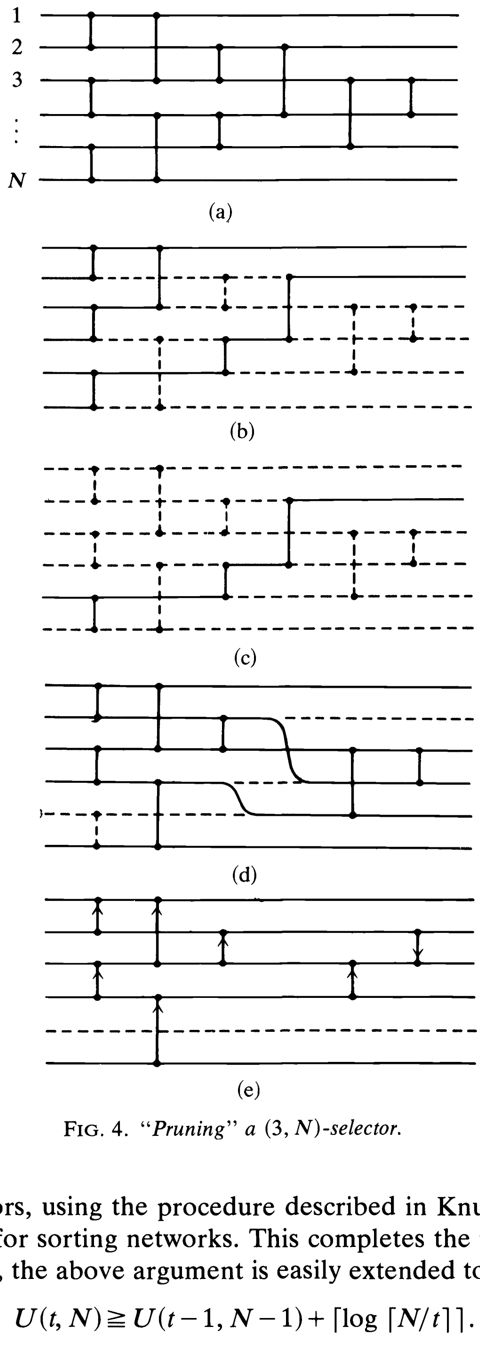
$$U(3,N)\geqq 2N-6+\lceil\log\lceil N/3\rceil\rceil. \tag{14}$$

The basic idea of "pruning" a network used in this proof was originally used for sorting networks (Green [4]). Let A be any $(3, N)$-selector such as the one shown in Fig. 4a. We will show that, by removing at least $\lceil\log\lceil N/3\rceil\rceil$ comparators from A and reconnecting some of the lines, we shall be left with a $(2, N-1)$-selector. This leads to inequality (14) since

$$U(3,N)\geqq U(2,N-1)+\lceil\log\lceil N/3\rceil\rceil=2N-6+\lceil\log\lceil N/3\rceil\rceil.$$

We begin by numbering the lines of the network from the top as in Fig. 4a. If the smallest element is input to the jth line, for any $1\leqq j\leqq N$, this element will always move "upward" across any comparators encountered, and wind up on one of the top three lines at the output end (see Fig. 4b). As j runs from 1 to N, these N paths can be divided into three groups according to the output line they lead to. One group will contain at least $\lceil N/3\rceil$ paths. We can look at this group of paths as a binary tree, regarding the comparators contained in the paths as branch nodes (internal nodes), and the input terminals as leaves. Since there are at least $\lceil N/3\rceil$ leaves, there is a path with at least $\lceil\log\lceil N/3\rceil\rceil$ comparators connected to it (Fig. 4c). We can remove this path and all the comparators incident with it (Fig. 4d). The resulting network looks different from a standard network, as it may contain "twisted" lines. In particular, the $N-1$ input lines consecutively numbered from top down can appear in a permuted order at the output end (the order is 1, 3, 2, 4, 5 in Fig. 4d). Also, a comparator across two lines i and j with $i<j$ may send the smaller element carried to line j (instead of i); for example, the last comparator in Fig. 4d compares elements carried on lines 2 and 3 and sends the smaller to line 3. Nevertheless, the resulting network (Fig. 4d) is functionally identical to a $(2, N-1)$-selector, in that the two smallest input elements always appear in two specified output lines. We now show how a $(2, N-1)$-selector can be obtained from it.

By straightening out the lines while keeping the comparators attached to the same set of lines, we obtain a network with two types of comparators: those that move a smaller input to the upper line and those that move it to the lower line; we distinguish the two types by associating the former with an "up" arrow and the latter with a "down" arrow (see Fig. 4e). It is not hard to transform this network with "arrowed" comparators (see Knuth [6, p. 236]) into a $(2, N-1)$-selector in the standard form with the same



(a)

(b)

(c)

(d)

(e)

FIG. 4. *"Pruning" a $(3, N)$-selector.*

number of comparators, using the procedure described in Knuth [6, p. 239, Ex. 16] originally developed for sorting networks. This completes the proof of formula (14).

For general $t \geqq 2$, the above argument is easily extended to give

$$U(t, N) \geqq U(t-1, N-1) + \lceil \log \lceil N/t \rceil \rceil.$$

Theorem 3.6 follows by repeatedly applying the above inequality and using Alekseyev's bound (2),

$$U(2^{\lfloor \log t \rfloor}, N) \geqq \lceil \log (t+1) \rceil (N - 2^{\lfloor \log t \rfloor}). \qquad \square$$

We conclude this section with the following theorem.

THEOREM 3.7. *For $N \geqq 5$,*

$$2N - 6 + \lceil \log \lceil N/3 \rceil \rceil \leqq U(3, N) \leqq 2N - 5 + \lfloor \log (N-3) \rfloor.$$

Proof. We need only prove the upper bound. From Lemma 3.2, we have

$$\begin{aligned} U(3,N) &\leqq U(3,\lceil N/2\rceil+1)+U(1,\lfloor N/2\rfloor)+\lfloor N/2\rfloor \\ &= U(3,\lceil N/2\rceil+1)+2\lfloor N/2\rfloor-1. \end{aligned} \tag{15}$$

Using $U(3,5)=6$ and $U(3,6)=8$ (see Knuth [6, p. 235]) as the basis of induction, and making use of the identity

$$1+\lfloor\log(\lceil N/2\rceil-2)\rfloor=\lfloor\log(N-3)\rfloor,$$

one can prove by induction from (15) that

$$U(3,N)\leqq 2N-5+\lfloor\log(N-3)\rfloor. \qquad \square$$

Theorem 3.7 determines $U(3,N)$ to within a constant of 2; as it is not hard to check through case analysis (let $N=3l$, $3l-1$, or $3l-2$ with $2^{k-1}<l\leqq 2^k$) that, for $N\geqq 5$,

$$\lfloor\log(N-3)\rfloor\leqq 1+\lceil\log\lceil N/3\rceil\rceil.$$

4. New results concerning $T(t,N)$. In this section a new inequality involving $T(t,N)$ is derived. With the help of this inequality, we can determine the asymptotic value of $T(t,N)$ for fixed t and large N to within a term of the order log log log N. For general values of t and N, this inequality also provides a new method for deriving lower bounds on $T(t,N)$. As an interesting application, it is shown that the minimal delay time for a sorting network with N inputs is at least $2.4\log N$ for large N.

4.1. Main theorem. This subsection is devoted to a proof of the following theorem which forms the basis of all later discussions. Throughout this section we adopt the convention that the binomial coefficient $\binom{k}{j}$ is zero if $k<j$.

THEOREM 4.1. *$T(t,N)$ satisfies the following inequality*:

$$t\lceil\log(t+1)\rceil\geqq\frac{N}{2^{T(t,N)}}\sum_{i=0}^{\lfloor\log t\rfloor}\left[(\lceil\log(t+1)\rceil-i)\binom{T(t,N)}{i}\right]. \tag{16}$$

COROLLARY 4.2.

$$T(t,N)\geqq\log N+\log\left[\frac{1}{t\lceil\log(t+1)\rceil}\binom{T(t,N)}{\lfloor\log t\rfloor}\right]. \tag{17}$$

Proof of Theorem 4.1. In a network that is divided into s levels, each line can be viewed as being partitioned into $s+1$ segments as shown in Fig. 5. Let us associate with each line segment a *weight* as in Knuth [6, p. 234]:

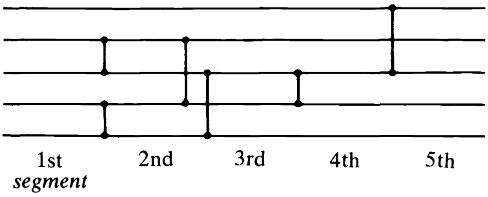


FIG. 5. *Dividing a network into segments.*

(1) The first (i.e., the leftmost) segment of each line is assigned weight 0.

(2) If a line is not connected to any comparator at the lth level, then its weights on the lth and the $(l+1)$st segments are the same.

(3) Let m_i and m_j be the weights on the lth segments of line i and j respectively where $i<j$ and $l \geqq 1$. If there is a comparator at the lth level between line i and line j, then on their $(l+1)$st segments line i has weight $\min(m_i, m_j)$ and line j has weight $\max(m_i, m_j)+1$.

An example of the weight assignment of a network is shown in Fig. 6.

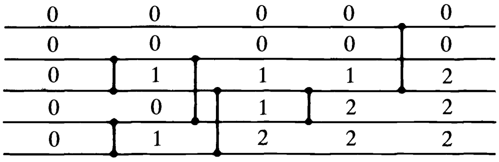


FIG. 6. *Weight assignment of a network.*

For a (t, N)-selector with s levels, we define two $s+1$ by $\lfloor \log t \rfloor + 1$ matrices $X=(x_{lk})$ and $Y=(y_{lk})$ by letting

$x_{lk}=$ the number of lines whose lth segments have weight equal to k,

and

$$(18) \qquad y_{lk}=\sum_{u=0}^{k}(k+1-u)x_{lu}, \qquad l=1,2,\cdots,s+1,$$

$$k=0,1,2,\cdots,\lfloor \log t \rfloor.$$

In particular, $y_{10}=N$.

Of course the difference $y_{l+1,k}-y_{l,k}$ depends on the locations of comparators at the lth level. A comparator at the lth level will not affect $y_{l+1,k}-y_{l,k}$ if at least one of its input segments has weight exceeding k, because this type of comparator does not change the number of lines with weights u for $u \leqq k$; only those comparators at the lth level where the maximum weight of two input segments does not exceed k can affect $y_{l+1,k}-y_{l,k}$. In fact, it is easily seen that every such comparator contributes exactly -1 to the difference $y_{l+1,k}-y_{l,k}$.

LEMMA A.

$$0 \leqq y_{l,k}-y_{l+1,k} \leqq \tfrac{1}{2}(x_{l0}+x_{l1}+\cdots+x_{lk}).$$

Proof of Lemma A. As mentioned above, $y_{l,k}-y_{l+1,k}$ is equal to the number of comparators at the lth level for which both input line segments have weights not exceeding k. Clearly the number of such comparators is bounded by $\frac{1}{2}(x_{l0}+x_{l1}+\cdots+x_{lk})$. □

LEMMA B.

$$(19) \qquad \begin{aligned} &y_{l+1,0} \geqq \tfrac{1}{2}y_{l,0}, \\ &y_{l+1,k} \geqq \tfrac{1}{2}(y_{l,k}+y_{l,k-1}), \qquad k \geqq 1. \end{aligned}$$

Proof of Lemma B. By definition of y_{lk},

$$x_{l0}=y_{l0}, \qquad x_{l0}+x_{l1}+\cdots+x_{lk}=y_{lk}-y_{l,k-1}, \qquad k \geqq 1.$$

According to Lemma A, we have

$$y_{l,0}-y_{l+1,0} \leqq \tfrac{1}{2}y_{l,0}$$

and

$$y_{l,k}-y_{l+1,k} \leqq \tfrac{1}{2}(y_{lk}-y_{l,k-1}), \qquad k \geqq 1.$$

This then leads to (19). □

LEMMA C.

$$y_{lj} \geqq \frac{N}{2^{l-1}} \sum_{i=0}^{j} \binom{l-1}{i}(j+1-i), \qquad l=1,2,\cdots,s+1, \quad j=0,1,2,\cdots,\lfloor \log t \rfloor.$$

Proof of Lemma C. Use Lemma B and prove by induction. □

We are now ready to prove Theorem 4.1. As was shown in Knuth [6, p. 235], when the weight function is so defined, there are in a (t, N)-selector at most t output lines which have weight less than or equal to $\lfloor \log t \rfloor$. Therefore,

$$\begin{aligned} y_{s+1, \lfloor \log t \rfloor} &= \sum_{i=0}^{\lfloor \log t \rfloor} (\lfloor \log t \rfloor + 1 - i) x_{s+1,i} \\ &\leqq (\lfloor \log t \rfloor + 1)t = \lceil \log (t+1) \rceil t. \end{aligned} \tag{20}$$

Note that we have used the identity $\lceil \log (t+1) \rceil = \lfloor \log t \rfloor + 1$ for all integers $t \geqq 1$. On the other hand, according to Lemma C

$$y_{s+1, \lfloor \log t \rfloor} \geqq \frac{N}{2^s} \sum_{i=0}^{\lfloor \log t \rfloor} (\lfloor \log t \rfloor + 1 - i)\binom{s}{i}. \tag{21}$$

Comparing (20) and (21), we obtain

$$t \lceil \log (t+1) \rceil \geqq \frac{N}{2^s} \sum_{i=0}^{\lfloor \log t \rfloor} (\lceil \log (t+1) \rceil - i)\binom{s}{i}. \tag{22}$$

This completes the proof of Theorem 4.1. □

4.2. Value of $T(t, N)$ for fixed t and large N. It is easy to transform a (t, N)-selector into a $(t-t_0, N-t_0)$-selector by deleting the top t_0 lines together with all the comparators that are connected to them. This leads to,

$$T(t, N) \geqq T(t-t_0, N-t_0). \tag{23}$$

In particular, when $t \leqq N/2$, we have

$$T(t, N) \geqq T(1, N-t+1) \geqq \lceil \log (N-t+1) \rceil \geqq \lfloor \log N \rfloor - 1. \tag{24}$$

Formula (24) is true for all t as $T(t, N) = T(N-t, N)$.

To derive a better bound than (24) we use (17),

$$T(t, N) \geqq \log N + \log \left[\frac{1}{t \lceil \log (t+1) \rceil} \binom{T(t, N)}{\lfloor \log t \rfloor} \right]. \tag{25}$$

Now, according to (24), $T(t, N) \geqq \lfloor \log N \rfloor - 1$. Thus, (25) implies

$$T(t, N) \geqq \log N + \log \left[\frac{1}{t \lceil \log (t+1) \rceil} \binom{\lfloor \log N \rfloor - 1}{\lfloor \log t \rfloor} \right]. \tag{26}$$

For fixed t and large N,

$$\binom{\lfloor \log N \rfloor - 1}{\lfloor \log t \rfloor} = \frac{(\lfloor \log N \rfloor)^{\lfloor \log t \rfloor}}{(\lfloor \log t \rfloor)!}\left(1 + O\left(\frac{1}{\log N}\right)\right).$$

Therefore, (26) becomes

$$T(t, N) \geqq \log N + \lfloor \log t \rfloor \log \log N + C(t) \quad \text{for some function } C(t). \tag{27}$$

The next theorem states that the lower bound in (27) is actually a good approximation to $T(t, N)$ in the asymptotic sense. (We remark that it is easy to see that $T(1, N) = \lceil \log N \rceil$).

THEOREM 4.3. *For any fixed $t \geqq 2$ and large N,*

$$T(2, N) = \log N + \log\log N + O(1),$$

$$T(t, N) = \log N + \lfloor \log t \rfloor \log\log N + O(\log\log\log N).$$

Proof. Because of (27), the theorem can be proved by exhibiting (t, N)-selectors that have the desired number of levels. The construction is quite similar to that used in the proof of Theorem 3.5, and will be left to Appendix B. □

4.3. More applications on Theorem 4.1. We shall develop a technique which generates lower bounds for $T(t, N)$ from formula (16). Let us state Theorem 4.1 in the following form: For given t, N, let s_0 be the smallest integer s satisfying

$$(28) \qquad t\lceil \log (t+1)\rceil \geqq \frac{N}{2^s} \sum_{i=0}^{\lfloor \log t \rfloor} (\lceil \log (t+1)\rceil - i)\binom{s}{i},$$

then $T(t, N) \geqq s_0$.

Let $A = \lfloor \log t \rfloor + 1$. The right-hand side of (28) can be written as

$$\frac{N}{2^s}\sum_{i=0}^{A-1} (A-i)\left[\binom{s-1}{i} + \binom{s-1}{i-1}\right]$$

$$= \frac{N}{2^s}\left[\sum_{i=0}^{A-1} (A-i)\binom{s-1}{i} + \sum_{i=0}^{A-2} (A-1-i)\binom{s-1}{i}\right]$$

$$< \frac{N}{2^{s-1}} \sum_{i=0}^{A-1} (A-i)\binom{s-1}{i}.$$

Thus, the right-hand side of formula (28) is a deçreasing function of s for fixed t and N. Therefore, any s that violates (28) will satisfy $s_0 > s$, and, as a consequence, $T(t, N) > s$. This leads to the following procedure:

A TECHNIQUE FOR GENERATING LOWER BOUNDS. *For any given t, N, find an s violating* (28). *We then have $T(t, N) > s$.*

To see how this technique works, we prove the following theorem.

THEOREM 4.4. *For $N \geqq 2$ and $N/(2 \log N) \geqq t \geqq 2$,*

$$(29) \qquad T(t, N) \geqq 2\lfloor \log t \rfloor.$$

Proof. Let $s = 2\lfloor \log t \rfloor$. The right-hand side of (28) is equal to

$$\frac{N}{2^s} \sum_{i=0}^{\lfloor \log t \rfloor} (\lceil \log (t+1)\rceil - i)\binom{s}{i} > \frac{N}{2^s} \sum_{i=0}^{\lfloor \log t \rfloor} \binom{s}{i}$$

$$\geqq \frac{N}{2^s}\frac{1}{2}\sum_{i=0}^{s}\binom{s}{i} = \frac{N}{2}.$$

The left-hand side of (28) is equal to

$$t\lceil \log (t+1)\rceil = t(\lfloor \log t \rfloor + 1) \leqq \frac{N}{2\log N}(\log N - \log\log N) \leqq \frac{N}{2}.$$

Thus (28) is not true for $s = 2\lfloor \log t \rfloor$. This proves $T(t, N) \geqq 2\lfloor \log t \rfloor$. □

It should be pointed out that a slightly different form of (29) can be derived in another way. Noting that each level can contain at most $N/2$ comparators, we can combine the inequality $T(t, N) \geqq U(t, N)/(N/2)$ with Alekseyev's lower bound (2) to obtain a result similar to (29).

The following theorem is a more interesting application of our technique.

THEOREM 4.5. *Let $\alpha_0 = 1/(3(2-\log 3)) \approx 0.8$. For any fixed $\varepsilon > 0$, there exists a number $f(\varepsilon)$ such that*

$$T(\lceil N^{\alpha_0} \rceil, N) \geqq \left(\frac{1}{2-\log 3} - \varepsilon\right) \log N \approx (2.41 - \varepsilon) \log N \tag{30}$$

for all $N \geqq f(\varepsilon)$.

COROLLARY 4.6. *There exists a function $g(\varepsilon)$ such that,*

$$T(t, N) \geqq \left(\frac{1}{2-\log 3} - \varepsilon\right) \log N$$

for all $N \geqq g(\varepsilon)$ and $N/2 \geqq t \geqq N^{\alpha_0}$.

Proof of Theorem 4.5. Let $\varepsilon_0 > 0$ be any fixed number. Without loss of generality we can assume ε satisfies $\varepsilon_0 > \varepsilon > 0$. We shall choose ε_0 to be small enough such that $\varepsilon_0 < \alpha_0$ and that

$$h(\varepsilon) > 0 \quad \text{for all } \varepsilon_0 > \varepsilon > 0,$$

where

$$h(\varepsilon) = \varepsilon\left[1 - \log\left(3 - \frac{\varepsilon}{\alpha_0}\right) + \log\left(2 - \frac{\varepsilon}{\alpha_0}\right)\right] + \alpha_0\left(3 \log\left(1 - \frac{\varepsilon}{3\alpha_0}\right) - 2 \log\left(1 - \frac{\varepsilon}{2\alpha_0}\right)\right).$$

The existence of ε_0 is guaranteed by the facts $h(0) = 0$ and $h'(0) = 2 - \log 3 > 0$.

Now, let $t = \lceil N^{\alpha_0} \rceil$, and $s = \lceil (1/(2-\log 3) - \varepsilon) \log N \rceil = \lceil (3\alpha_0 - \varepsilon) \log N \rceil$. We shall prove that (28) is not satisfied when N is sufficiently large. This then implies the theorem.

To prove our assertion, we observe that for large N, Stirling's approximation yields

$$\begin{aligned}
&\binom{\lceil (3\alpha_0 - \varepsilon) \log N \rceil}{\lfloor \alpha_0 \log N \rfloor} \\
&\approx \frac{((3\alpha_0 - \varepsilon) \log N)!}{(\alpha_0 \log N)!((2\alpha_0 - \varepsilon) \log N)!} \\
&\approx \frac{\sqrt{2\pi(3\alpha_0 - \varepsilon) \log N}\left(\frac{(3\alpha_0 - \varepsilon) \log N}{e}\right)^{(3\alpha_0 - \varepsilon) \log N}}{\sqrt{2\pi\alpha_0 \log N}\left(\frac{\alpha_0 \log N}{e}\right)^{\alpha_0 \log N} \sqrt{2\pi(2\alpha_0 - \varepsilon) \log N}\left(\frac{(2\alpha_0 - \varepsilon) \log N}{e}\right)^{(2\alpha_0 - \varepsilon) \log N}} \\
&\approx \text{constant} \cdot \frac{1}{\sqrt{\log N}} N^{(3\alpha_0 - \varepsilon) \log (3\alpha_0 - \varepsilon) - \alpha_0 \log \alpha_0 - (2\alpha_0 - \varepsilon) \log (2\alpha_0 - \varepsilon)}.
\end{aligned}$$

Therefore, in formula (28)

$$\begin{aligned}
\text{right-hand side} &\geqq \frac{N}{2^s}\binom{\lceil (3\alpha_0 - \varepsilon) \log N \rceil}{\lfloor \alpha_0 \log N \rfloor} \\
&\geqq \text{constant} \cdot \frac{1}{\sqrt{\log N}} N^{1-(3\alpha_0 - \varepsilon)} \\
&\qquad \cdot N^{(3\alpha_0 - \varepsilon) \log (3\alpha_0 - \varepsilon) - \alpha_0 \log \alpha_0 - (2\alpha_0 - \varepsilon) \log (2\alpha_0 - \varepsilon)}.
\end{aligned}$$

After some algebraic manipulation, we have,

$$\text{(31)} \qquad \text{right-hand side} \geqq \text{constant} \cdot \frac{1}{\sqrt{\log N}} N^{\alpha_0} N^{h(\varepsilon)}.$$

However, the left-hand side of (28) is equal to

$$\text{(32)} \qquad \text{left-hand side} = t\lceil \log (t+1) \rceil \leqq \text{constant} \cdot N^{\alpha_0}(\alpha_0 \log N).$$

Since $h(\varepsilon) > 0$, a comparison of (31) and (32) shows that, for sufficiently large N, the right-hand side of (28) is greater than the left-hand side. This is a violation of (28). We have proved our assertion. □

Corollary 4.6 can be obtained in the following way: First we use (23) to get,

$$T(t, N) \geqq T(N^{\alpha_0}, N + N^{\alpha_0} - t) \geqq T\left(N^{\alpha_0}, \frac{N}{2}\right).$$

A lower bound for $T(N^{\alpha_0}, N/2)$ can be obtained in exactly the same way as that for $T(N^{\alpha_0}, N)$. This leads to the corollary.

An interesting consequence of Theorem 4.5 is that, for a sorting network with N inputs, the delay time is at least 2.4 log N for sufficiently large N. This result seems to be new.

Remarks. For simplicity, we have treated $(3\alpha_0 - \varepsilon) \log N$, $\alpha_0 \log N$ as if they were integers in the proof of Theorem 4.5. It is straightforward to modify the proof (bound $\lceil x \rceil!$ by $(x+1)!$ or $(x-1)!$) to make it completely rigorous.

5. Conclusions. In this paper, bounds have been given on the minimal "cost" and "delay time" of selection networks built of comparators. In particular, we have identified the leading asymptotic terms of $U(t, N)$ and $T(t, N)$ for fixed t. Many questions still remain open. For example,

(1) What is the exact value of $U(3, N)$?

(2) What is the order of magnitude of $U(N/2, N)$ as $N \to \infty$? Does

$$\lim_{N \to \infty} \frac{U(N/2, N)}{N \log N} \text{ exist?}$$

(3) What is $T(N/2, N)$?

(4) When the inputs are restricted to be Boolean variables, a comparator can be replaced by a pair of "AND" "OR" logic gates. In this case, networks of comparators are logic circuits of a special type. By adding $t-1$ comparators between the top t output lines of a (t, N)-selector, one can obtain a network whose tth output line contains the tth smallest of the inputs. Replacing the comparators by "AND" "OR" pairs of gates yields a monotone circuit (consisting only of "AND" "OR" gates) that computes the tth order symmetric function of n input Boolean variables. Our results (Theorem 3.1) show that $\approx 2\lceil \log (t+1) \rceil n$ gates are sufficient as $n \to \infty$. Question: Do there exist much better (in terms of the number of gates) general monotone circuits for the symmetric functions?

Appendix A. Proof of Theorem 3.5. Let $f(t, N)$ be a function (to be defined later) that satisfies $f(t, N) \geqq t$. We shall construct a family of networks $E(t, N)$ called (t, N)-*eliminators* with the following property: Of the N output lines of $E(t, N)$ there are $f(t, N)$ designated lines among which the smallest t elements are found for any permutation of the inputs.

According to Alekseyev's upper bound (2), there exists a $(t, f(t, N))$-selector F (dependent on t and N) that contains $(f(t, N)-t)(1+2\hat{S}(t)/t) \leqq 3\lceil\log(t+2)\rceil^2 f(t, N)$ comparators.[3] We can append this network F to the (t, N)-eliminator $E(t, N)$ by making the $f(t, N)$ designated output lines of $E(t, N)$ the inputs to the network F. Figure A.1 shows such an arrangement. Clearly this gives us a (t, N)-selector.

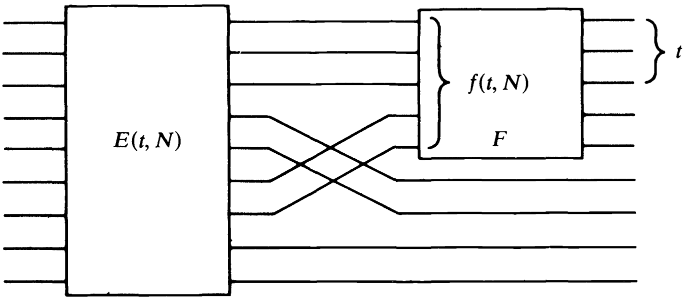


FIG. A.1. *Construction of a (t, N)-selector.*

If $g(t, N)$ is the number of comparators contained in $E(t, N)$, then the total number of comparators in the (t, N)-selector of Figure A.1 is bounded from above by

$$g(t, N)+3\lceil\log(t+2)\rceil^2 f(t, N).$$

We have proved

LEMMA A.1.

$$U(t, N) \leqq g(t, N)+3\lceil\log(t+2)\rceil^2 f(t, N). \tag{A.1}$$

We shall now define $f(t, N)$, $E(t, N)$, and derive upper bounds for $g(t, N)$. They can then be substituted into (A.1) to prove Theorem 3.5.

Network $E(t, N)$. $E(t, N)$ and $f(t, N)$ are derived inductively as follows:

(a) $f(1, N)=1$, $f(2, N)=2$. Networks $E(1, N)$ and $E(2, N)$ are shown in Figs. A.2 and A.3, respectively.

(b) $t \geqq 3$:

(i) If $t \geqq N$, then $f(t, N)=N$ and $E(t, N)$ contains no comparators.

(ii) If $t<N$, then $f(t, N)$ is given by

$$f(t, N)=f(t, \lceil N/2\rceil)+f(\lfloor t/2\rfloor, \lfloor N/2\rfloor), \tag{A.2}$$

and $E(t, N)$ is defined as in Figure A.4. (Note that of the t smallest input elements, at most $\lfloor t/2\rfloor$ will be input to $E(\lfloor t/2\rfloor, \lfloor N/2\rfloor)$. This insures that the construction gives a $(t, f(t, N))$ eliminator.)

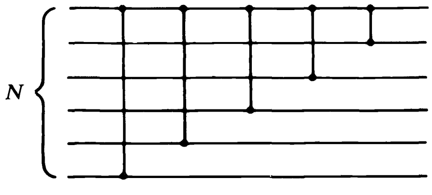


FIG. A.2. *The network $E(1, N)$.*

[3] Batcher's sorting network [1] gives $\hat{S}(t) \leqq (1+\lceil\log t\rceil^2/4)2^{\lceil\log t\rceil} \leqq \lceil\log(t+2)\rceil^2 t$. Hence $1+2\hat{S}(t)/t \leqq 3\lceil\log(t+2)\rceil^2$.

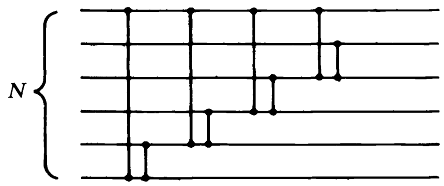


FIG. A.3. *The network* $E(2, N)$.

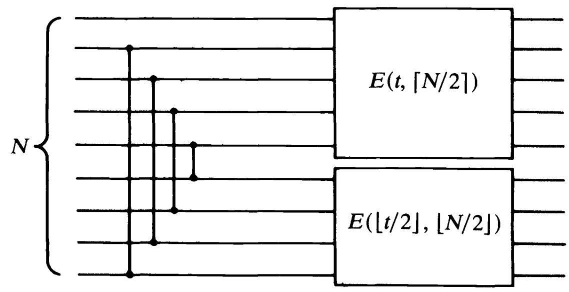


FIG. A.4. *The network* $E(t, N)$ *when* $3 \leqq t < N$; *the* $f(t, N)$ *designated output lines are the union of the designated lines of* $E(t, \lceil N/2 \rceil)$ *and* $E(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor)$.

Clearly, the networks $E(t, N)$ as defined are $(t, f(t, N))$-eliminators.

From this construction, we have

$$\text{(A.3)} \qquad g(t, N) = \begin{cases} 0 & \text{if } t \geqq N, \\ N-1 & \text{if } t = 1 < N, \\ 2N-4 & \text{if } t = 2 < N, \\ g(t, \lceil N/2 \rceil) + g(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor) + \lfloor N/2 \rfloor & \text{if } 3 \leqq t < N. \end{cases}$$

LEMMA A.2. *For* $t, N \geqq 1$,

$$\text{(A.4)} \qquad \begin{aligned} &g(t, N) \leqq \lceil \log (t+1) \rceil N, \\ &f(t, N) \leqq 2\left[\binom{\lceil \log N \rceil}{0} + \binom{\lceil \log N \rceil}{1} + \cdots + \binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil}\right]. \end{aligned}$$

Proof. We prove the lemma by induction in the lexicographic order of (t, N). We will only show the inductive step. Let $t \geqq 3$. One can verify it directly for $N \leqq t$. For $N > t$, we have

$$\begin{aligned} g(t, N) &= g(t, \lceil N/2 \rceil) + g(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor) + \lfloor N/2 \rfloor \\ &\leqq \lceil \log (t+1) \rceil \lceil N/2 \rceil + \lceil \log (\lfloor t/2 \rfloor + 1) \rceil \lfloor N/2 \rfloor + \lfloor N/2 \rfloor \\ &= \lceil \log (t+1) \rceil \lceil N/2 \rceil + (\lceil \log (t+1) \rceil - 1) \lfloor N/2 \rfloor + \lfloor N/2 \rfloor \\ &= \lceil \log (t+1) \rceil N, \end{aligned}$$

and, using the notations λ, λ' (with $\lambda = 1+\lambda'$) as in the proof of Theorem 3.1,

$$
\begin{aligned}
f(t, N) &= f(t, \lceil N/2 \rceil) + f(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor) \\
&\leqq 2\left(\sum_{i=0}^{\lambda} \binom{\lceil \log \lceil N/2 \rceil \rceil}{i} + \sum_{i=0}^{\lambda'} \binom{\lceil \log \lfloor N/2 \rfloor \rceil}{i} \right) \\
&\leqq 2 \sum_{i=0}^{\lambda} \left[\binom{\lceil \log \lceil N/2 \rceil \rceil}{i} + \binom{\lceil \log \lceil N/2 \rceil \rceil}{i-1} \right] \\
&= 2 \sum_{i=0}^{\lambda} \binom{\lceil \log \lceil N/2 \rceil \rceil + 1}{i} \\
&= 2 \sum_{i=0}^{\lambda} \binom{\lceil \log N \rceil}{i}.
\end{aligned}
$$

We have used the monotonicity of $\binom{n}{j}$ in n and the identity $\lceil \log N \rceil = \lceil \log \lceil N/2 \rceil \rceil + 1$ for $N \geqq 2$ in the last derivation. □

Substituting (A.4) into (A.1), we obtain

$$
U(t, N) \leqq \lceil \log (t+1) \rceil N + 6 \lceil \log (t+2) \rceil^2 \sum_{k=0}^{\lceil \log ((t+1)/3) \rceil} \binom{\lceil \log N \rceil}{k}. \tag{A.5}
$$

Now, if $t < \sqrt{N}$,

$$
\binom{\lceil \log N \rceil}{k} \leqq \binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil}
$$

for all $k = 0, 1, 2, \cdots, \lceil \log ((t+1)/3) \rceil$. Therefore (A.5) implies

$$
U(t, N) \leqq \lceil \log (t+1) \rceil N + 12 \lceil \log (t+2) \rceil^3 \binom{\lceil \log N \rceil}{\lceil \log ((t+1)/3) \rceil},
$$

for $t < \sqrt{N}$. This completes the proof of Theorem 3.5. □

Appendix B. Proof of Theorem 4.3. It suffices to exhibit (t, N)-selectors $(t \geqq 2)$ with the desired delay time. As the construction is very similar to that in Appendix A, we shall only sketch the proof.

Define networks $E'(t, N)$ recursively as in Fig. B.1. It is easy to see that $E'(t, N)$ is a $(t, f'(t, N))$-eliminator, where $f'(t, N)$ is given by

$$
f'(1, N) = 1,
$$

$$
f'(t, N) = N, \quad \text{if } t \geqq N,
$$

and

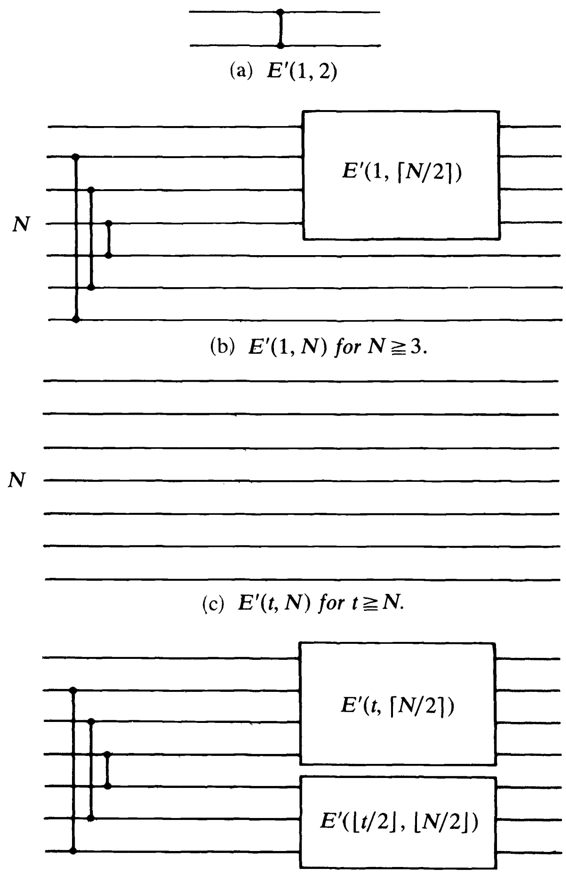
$$
f'(t, N) = f'(t, \lceil N/2 \rceil) + f'(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor) \quad \text{if } N > t \geqq 2.
$$

By induction, one can prove (cf. the proof of Theorem 3.1) that

$$
f'(t, N) = O((\log N)^{\lfloor \log t \rfloor}) \quad \text{for fixed } t \geqq 2. \tag{B.1}
$$

It is also easy to check that the delay time of $E'(t, N)$ is at most $\lceil \log N \rceil$.

We now define a (t, N)-selector for fixed $t \geqq 2$ and large N. We let the N input lines first go through the network $E'(t, N)$, and then feed the $f'(t, N)$ output lines that are known to contain the t smallest elements into a network $E'(t, f'(t, N))$. The



(a) $E'(1,2)$

(b) $E'(1,N)$ *for* $N \geqq 3$.

(c) $E'(t,N)$ *for* $t \geqq N$.

(d) $E'(t,N)$ *for* $N > t \geqq 2$.

FIG. B.1. *The definition of* $E'(t, N)$; *the designated output lines are the union of the designated lines of* $E'(t, \lceil N/2 \rceil)$ *and* $E'(\lfloor t/2 \rfloor, \lfloor N/2 \rfloor)$.

$N_1(=f'(t, f'(t, N)))$ output lines that may contain the t smallest elements are then fed into an $E'(t, N_1)$, whose $N_2(=f'(t, N_1))$ output lines containing the t smallest elements are then fed into a Batcher's sorting network [1] to complete the (t, N)-selector (see Fig. B.2). As the Batcher's network with n inputs has delay time $O((\log n)^2)$, the (t, N)-selector we constructed above has delay time

$$s = \lceil \log N \rceil + \lceil \log f'(t, N) \rceil + \lceil \log N_1 \rceil + O((\log N_2)^2).$$

Using the bounds given by (B.1), we obtain that, for any fixed $t \geqq 2$,

$$T(t, N) \leqq \log N + \lfloor \log t \rfloor \log \log N + O(\log \log \log N). \tag{B.2}$$

To complete the proof of Theorem 4.3, we need to show that the $O(\log \log \log N)$ term can be replaced by $O(1)$ in the case $t = 2$.

Consider the network $E'(2, N)$. It is easy to see from the construction that in the output end, the minimum element always appears in either of the top two lines, and the 2nd smallest element may appear in one of some $f'(2, N)$ lines (which include always the top two lines). A $(2, N)$-selector with delay time no greater than $\lceil \log N \rceil + 1 +$

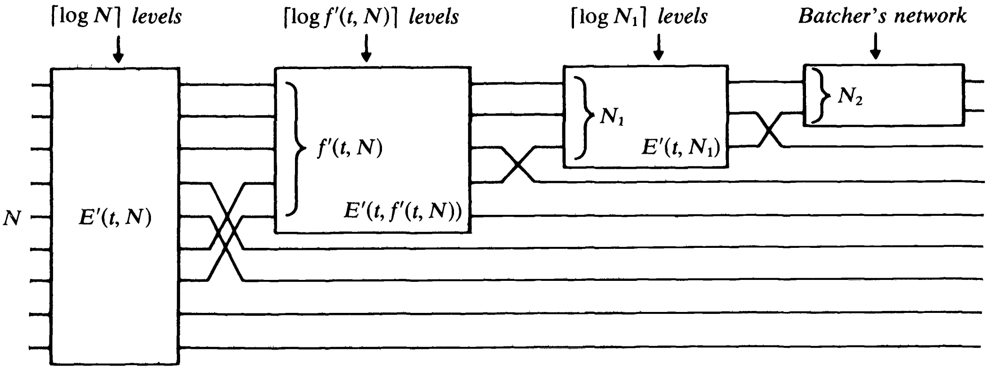

$\lceil \log N \rceil$ *levels*
$\lceil \log f'(t,N) \rceil$ *levels*
$\lceil \log N_1 \rceil$ *levels*
Batcher's network
N
$E'(t,N)$
$f'(t,N)$
$E'(t,f'(t,N))$
N_1
$E'(t,N_1)$
N_2

FIG. B.2. *The construction of a* (t,N)*-selector with delay time* $\log N + \lfloor \log t \rfloor \log\log N + O(\log\log\log N)$; $N_1 = f'(t, f'(t,N))$, $N_2 = f'(t,N_1)$.

$\lceil \log (f'(2,N) - 1) \rceil$ can be constructed by cascading an $E'(2,N)$ with a single comparator between the top two lines and an $E'(1, f'(2,N)-1)$ (see Fig. B.3). This proves

$$T(2,N) \leqq \log N + \log\log N + O(1).$$

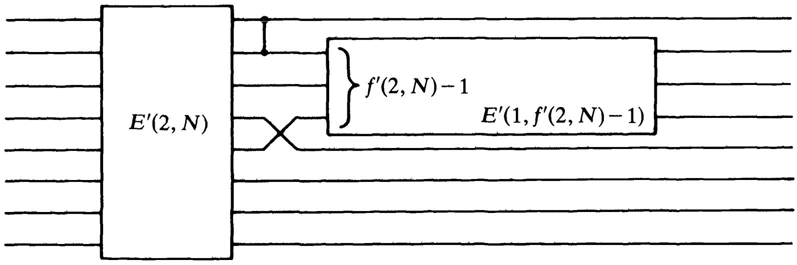


FIG. B.3. *A* $(2,N)$*-selector with delay time* $\log N + \log\log N + O(1)$.

The proof of Theorem 4.3 is completed. □

Acknowledgment. A preliminary version of this paper was presented at the *15th IEEE Symposium on Switching and Automata Theory*, October 1974.

REFERENCES

[1] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Computer Conference 32 (1968), AFIPS Press, Montvale, NJ, 1968, pp. 307–314.

[2] R. L. DRYSDALE III AND F. H. YOUNG, *Improved divide/sort/merge sorting networks*, this Journal, 4 (1975), pp. 264–270.

[3] M. W. GREEN, *Some improvements in non-adaptive sorting algorithms*, Proc. 6th Annual Princeton Conference on Information, Sciences and System, 1972, pp. 387–391.

[4] ———, *Some observations on sorting*, Cellular Logic in Memory Arrays, Final Report, Part 1, SRI Project 5509, Stanford Research Institute, Menlo Park, CA, 1970, pp. 49–71.

[5] D. E. KNUTH, *The Art of Computer Programming*, vol. 1, 2nd edition, Addison-Wesley, Reading, MA, 1973.

[6] ———, *The Art of Computer Programming*, vol. 3, 2nd printing, Addison-Wesley, Reading, MA, 1975.

[7] F. F. YAO, unpublished manuscript, 1974.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

AN OPTIMAL ALGORITHM FOR SYMBOLIC FACTORIZATION OF SYMMETRIC MATRICES*

ALAN GEORGE† AND JOSEPH W. H. LIU‡

Abstract. A fundamental problem in the computer solution of a sparse, N by N, positive definite system of equations $Ax = b$ is, given the structure of A, to determine the structure of its Cholesky factor L, where $A = LL^T$. This problem arises because it is often desirable to set up a data structure for L before the numerical computation is performed, and in order to do this we must know the positions of the nonzeros of L. We describe a *representation* $\mathcal{R}_L$ for L which typically requires far fewer data items than the number of nonzeros in L, and an algorithm is then described which generates $\mathcal{R}_L$. The time and space complexity of the algorithm is shown to be $O(|A|, |\mathcal{R}_L|)$, and can never be worse than $O(|L|)$. Here $|\mathcal{R}_L|$ denotes the number of items in the data structure for L, and $|A|$ and $|L|$ denote the number of nonzeros in A and L respectively. For a certain class of problems, we show that the execution time of the algorithm is $O(N)$, even though $|L|$ is $O(N \log N)$. We also provide some numerical results showing that the algorithm can be implemented so that the program performance reflects its theoretically predicted behavior.

Key words. Symbolic factorization, sparse matrices, linear equations

1. Introduction.

Consider the symmetric positive definite system of linear equations

$$Ax = b, \tag{1.1}$$

where A is N by N sparse. If we solve (1.1) using Cholesky's method, the system is usually first reordered by an algorithm such as the minimum degree algorithm, so that its Cholesky factor suffers low fill-in. We do not deal with this reordering problem here, and simply assume that (1.1) has already been reordered appropriately.

Since L is sparse, the next step in the solution process is to construct a data structure for L, so that only its nonzeros are stored. In order to do this we must know the positions of the nonzeros of L, leading to the problem which we deal with in this paper. That is, given the structure of A, we want to determine the corresponding structure of its Cholesky factorization. Since the process is entirely logical, involving no numerical computation, it is often referred to as "symbolic factorization."

At least two such algorithms for computing the fill-in (or the structure of L) have been described in the literature [8], [9]. The algorithms are quite similar, and their time complexity has been shown to be $O(|L|)$, where $|L|$ denotes the number of nonzeros in L. If the desired output of the algorithms is the positions of the nonzeros in L, then the immediate implication is that these algorithms are asymptotically optimal.

However, in practice, the reason for performing symbolic factorization is to construct a storage scheme for L, and very often these data structures involve far fewer than $|L|$ items of information [9]. For example, for one such data structure and problem class, the number of data structure pointers etc. is $O(N)$, even though $|L|$ is $O(N \log N)$ [3]. In this context it seems reasonable to regard an algorithm as optimal only if its execution time and its space requirement are both $O(|\mathcal{R}_L|, |A|)$, where $\mathcal{R}_L$ is the representation used to describe the structure of L.

In this paper we describe an algorithm for symbolic factorization which generates a specific representation for L. The time complexity of the algorithm is shown to be

* Received by the editors September 15, 1978, and in revised form August 7, 1979. This research was supported in part by the Canadian National Research Council under Grant A8111.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.
‡ Systems Dimensions Ltd., 111 Avenue Road, Toronto, Ontario, Canada.

optimal in the above sense (i.e., $O(|\mathcal{R}_L|, |A|))$ and can never be worse than $O(|L|)$. For a special class of finite element problems, appropriately ordered, we show that the execution time of the algorithm is $O(N)$, even though $|L|$ is $O(N \log N)$.

In general, we note that the representation produced may not be optimal with respect to $|\mathcal{R}_L|$. However, the algorithm is optimal for producing this representation.

Our algorithm has another important feature not present in previous algorithms. The algorithm *does not use any of its output during execution*; it operates only on the graph of A. Thus, if insufficient space is available to retain $\mathcal{R}_L$, it can be discarded; the algorithm can still execute and determine the *number* $|\mathcal{R}_L|$, thus furnishing the amount of space needed for $\mathcal{R}_L$. Alternatively, $\mathcal{R}_L$ can be written on secondary storage as it is generated. In either case, we argue that our algorithm is optimal in terms of space requirements.

An outline of our paper is as follows. In § 2 we describe a useful representation $\mathcal{R}_L$ for the structure of L which is the basis for at least two efficient storage schemes for sparse matrices [3], [9]. Section 3 contains a review of a *quotient graph model* of symmetric factorization, developed by the authors in [5]. Section 4 contains a development of the symbolic factorization algorithm, using the quotient graph model, and § 5 contains an example illustrating that its execution can be of an order lower than $|L|$. Section 6 contains a few numerical experiments showing that the algorithm can be implemented efficiently, along with a discussion of its advantages and disadvantages compared with alternative symbolic factorization algorithms.

2. Representation of the structure of *L*. In this section we consider the problem of efficiently representing the structure of the lower triangular matrix L. This task is clearly fundamental in the design of an efficient data structure for storing L.

It is often true that when symmetric Gaussian elimination is applied to matrices which have been ordered so as to suffer low fill-in, the resulting factor L has many "similar" columns. That is, for some column k, there are several columns j, $j > k$, having essentially the same nonzero structure. This fact motivates the following definition.

Given an N by N lower triangular matrix L, column k is a *representative* for column j, $j > k$, if and only if for all $i \geqq j$, $L_{ij} \neq 0 \Leftrightarrow L_{ik} \neq 0$. Simply stated, column k represents column j if they have identical structure below position j. Note that column k represents itself.

If we know the structure of column k, and we know which columns it represents, then their structure is completely determined. Since we are interested in representing the entire structure of L, we are led to the following definitions.

A representative map $\mathcal{M}$ for L is an ordered set of integers $(m_1, m_2, \cdots, m_N)$ such that $1 \leqq m_j \leqq j$, and such that $m_j = k$ implies that column k is a representative for column j. The subset $\mathcal{M}^* \subset \mathcal{M}$ consisting of distinct members of $\mathcal{M}$ is called a *representative set* for L. Finally, a *representation* for L by $\mathcal{M}$ is the set $\mathcal{R}_L$, where $\mathcal{R}_L$ is given by

$$\mathcal{R}_L = \{(i, j) \mid j \in \mathcal{M}^*, L_{ij} \neq 0\}.$$

An example illustrating these definitions is given in Figure 2.1.

A representative map is said to be *monotone* if $m_i \leqq m_{i+1}$, $1 \leqq i < N$. In this case it is clear that $\mathcal{M}^*$ and $\mathcal{R}_L$ are sufficient to describe the structure of L, since $\mathcal{M}$ can be inferred from $\mathcal{M}^*$.

3. The quotient graph model of symmetric elimination.

3.1. Graph theoretic preliminaries. Symbolic factorization is the process of simulating the numerical factorization of a given matrix A in order to obtain the zero–nonzero structure of its factor L. Since the numerical values of the matrix components are of no significance in this connection, the problem can be conveniently

$$L=\begin{array}{cccccccccc}
X & & & & & & & & & \\
 & X & & & & & & & & \\
 & X & X & & & & & & & \\
 & & & X & & & & & & \\
 & & X & X & X & & & & & \\
 & & & & & X & & & & \\
X & & X & X & X & & X & & & \\
X & & X & X & X & & X & X & & \\
 & & & X & X & & & & X & \\
 & & & & & X & & X & & X \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10
\end{array}$$

$\mathcal{M} = (1, 2, 3, 4, 4, 6, 1, 6, 4, 6)$

$\mathcal{M}^* = \{1, 2, 3, 4, 6\}$

$\mathcal{R}_L = \{(7, 1), (8, 1), (3, 2), (5, 3), (7, 3), (8, 3), (5, 4), (7, 4), (8, 4), (9, 4), (10, 6)\}$

FIG. 2.1. *An example illustrating the sets $\mathcal{M}$, $\mathcal{M}^*$ and $\mathcal{R}_L$.*

studied using a graph theory model. The readers are assumed to know the basic notions in graph theory, reference to which can be found in [1]. We now introduce some more definitions and establish results that are pertinent in the study of the elimination process.

Let $G = (X, E)$ be a given undirected graph. Consider a subset $S \subset X$ and a node $y \in X - S$. The node y is said to be *reachable from a node x through S* if there exists a path $(x, s_1, \cdots, s_t, y)$ such that $s_i \in S$ for $1 \leqq i \leqq t$. The *reachable set* of *y through S* is then defined to be

$$\text{Reach}\,(y, S) = \{x \in X - S \mid x \text{ is reachable from } y \text{ through } S\}.$$

A related notion is the *neighborhood set* of y in S, which is the subset

$$\text{Nbrhd}\,(y, S) = \{s \in S \mid y \text{ is reachable from } s \text{ through } S\}.$$

For convenience in later discussions, we define the *closure of y in S* to be

$$\text{Closure}\,(y, S) = \text{Reach}\,(y, S) \cup \text{Nbrhd}\,(y, S) \cup \{y\}.$$

We now review the relevance of these notions in Gaussian elimination. Let A be an N by N symmetric matrix. The labeled undirected graph of A, denoted by $G^A = (X^A, E^A)$, is the one for which X^A is labeled from 1 to N:

$$X^A = \{x_1, x_2, \cdots, x_N\},$$

and $\{x_i, x_j\} \in E^A$ if and only if $A_{ij} \neq 0$.

Let $A = LL^T$, where L is the Cholesky factor of A. The matrix $F = L + L^T$ is called the *filled matrix* of A and the corresponding graph $G^F = (X^A, E^F)$ the *filled graph* of G^A. The following result relates E^F directly to E^A using the reachable set notion. This idea was discovered independently in [8]. Let $S_i = \{x_1, \cdots, x_i\}$ for $1 \leqq i \leqq N$ and $S_0 = \varnothing$.

LEMMA 3.1 [4]. *Let $j > i$. The unordered pair $\{x_i, x_j\} \in E^F$ if and only if $x_j \in$* Reach (x_i, S_{i-1}).

COROLLARY 3.2 [4]. $|E^F| = \sum_{i=1}^N |\text{Reach}\,(x_i, S_{i-1})|$.

In subsequent sections of this paper, these various graph theoretic notions will be applied to different graphs. When the graph is not clear from context, we will attach the appropriate subscript to the nomenclature. Thus, notations of the following type will be used: $\mathrm{Adj}_G(y)$, $\deg_G(y)$, $\mathrm{Reach}_G(y, S)$, etc.

3.2. The quotient graph model. In [5], the authors introduced a quotient graph model for the study of the Gaussian elimination process. We now briefly review this model.

The central notion is that of a quotient graph. Let $G = (X, E)$ be a given graph. Let $\mathcal{P}$ be a partitioning of the node set X:

$$\mathcal{P} = \{Y_1, Y_2, \cdots, Y_p\}.$$

That is, $\bigcup_{k=1}^{p} Y_k = X$ and $Y_i \cap Y_j = \emptyset$ for $i \neq j$.

The quotient graph of G with respect to $\mathcal{P}$ is defined to be the graph $(\mathcal{P}, \mathscr{E})$, where $\{Y_i, Y_j\} \in \mathscr{E}$ if and only if $Y_i \cap \mathrm{Adj}\,(Y_j) \neq 0$. This graph will be denoted by $G/\mathcal{P}$.

An important type of partitioning is that defined by connected components. Let S be a subset of X. The *component partitioning* $\mathscr{C}(S)$ of S is defined as

$$\mathscr{C}(S) = \{Y \subset S \mid G(Y) \text{ is a connected component in the subgraph } G(S)\}.$$

This can be extended to a partitioning on X. We define the *partitioning on X induced by S* to be

$$\bar{\mathscr{C}}(S) = \mathscr{C}(S) \cup \{\{x\} \mid x \in X - S\}.$$

We are now ready to model the elimination process as a sequence of quotient graphs. Let $G^A = (X^A, E^A)$ be the graph and $x_1, x_2, \cdots, x_N$ be the sequence of node elimination. As in § 3.1, let $S_i = \{x_1, \cdots, x_i\}$ for $1 \leqq i \leqq N$ and $S_0 = \emptyset$.

Consider the partitioning $\bar{\mathscr{C}}(S_i)$ induced by S_i and the corresponding quotient graph $G/\bar{\mathscr{C}}(S_i)$. We shall denote this quotient graph by $\mathscr{G}_i$. In this way, we obtain a sequence of quotient graphs

$$\mathscr{G}_1, \mathscr{G}_2, \cdots, \mathscr{G}_N.$$

We quote the following result from [5].

THEOREM 3.3. *For $y \notin S_i$,*

$$\mathrm{Reach}_{\mathscr{G}_i}(\{y\}, \bar{\mathscr{C}}(S_i)) = \{\{x\} \mid x \in \mathrm{Reach}_{G^A}(y, S_i)\}.$$

By Lemma 3.1, the filled graph G^F is characterized by the set of reachable sets $\mathrm{Reach}_{G^A}(x_i, S_{i-1})$. Then, the above theorem implies that the filled graph can be implicitly represented by the sequence of quotient graphs.

The primary advantage of the quotient graph model is that its computer implementation is very efficient. In particular, it can be implemented in space proportional to the number $|E^A|$ of nonzeros in A. The in-place implementation of the model relies on the following results quoted from [5].

THEOREM 3.4. *Let $\mathscr{G}_i = (\bar{\mathscr{C}}(S_i), \mathscr{E}_i)$, $1 \leqq i \leqq N$, be the sequence of quotient graphs. Then, for $1 \leqq i \leqq N$*

$$|\bar{\mathscr{C}}(S_{i+1})| \leqq |\bar{\mathscr{C}}(S_i)|,$$

and

$$|\mathscr{E}_{i+1}| \leqq |\mathscr{E}_i|.$$

THEOREM 3.5. *For $x \notin S_{i+1}$*

$$|\mathrm{Adj}_{\mathscr{G}_{i+1}}(\{x\})| \leqq |\mathrm{Adj}_{\mathscr{G}_i}(\{x\})|.$$

For details of the in-place implementation, the reader is referred to [5]. In the next subsection, we shall consider some properties of the induced partitionings $\bar{\mathscr{C}}(S_i)$.

3.3. The component partitionings $\mathscr{C}(S_i)$. In the formulation of the quotient graph model, the sequence of node subsets

$$S_1, S_2, \cdots, S_N$$

defines a sequence of component partitionings

$$\mathscr{C}(S_1), \mathscr{C}(S_2), \cdots, \mathscr{C}(S_N).$$

We first establish the relation between $\mathscr{C}(S_{i-1})$ and $\mathscr{C}(S_i)$ through a series of lemmas.

LEMMA 3.6. $\text{Nbrhd}\,(x_i, S_{i-1}) = \bigcup\{Y \in \mathscr{C}(S_{i-1}) \mid x_i \in \text{Adj}\,(Y)\}$.

Proof. Consider any $Y \in \mathscr{C}(S_{i-1})$ with $x_i \in \text{Adj}\,(Y)$. For $y \in Y \subset S_{i-1}$, since $x_i \in \text{Adj}\,(Y)$, y is reachable from'a subset of Y and hence $y \in \text{Nbrhd}\,(x_i, S_{i-1})$.

On the other hand, consider $y \in \text{Nbrhd}\,(x_i, S_{i-1})$. Let $(x_i, s_1, \cdots, s_t, y)$ be a path where $\{s_1, \cdots, s_t, y\} \subset S_{i-1}$. Define $Y \in \mathscr{C}(S_{i-1})$ such that $\{s_1, \cdots, s_t, y\} \subset Y$. Clearly Adj (Y) contains x_i and hence the result follows. □

LEMMA 3.7. $\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}) \in \mathscr{C}(S_i)$.

Proof. Clearly $\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}) \subset S_i$ and is connected. That it is a maximal connected set in $G(S_i)$ is left as an exercise. □

THEOREM 3.8.

$$\mathscr{C}(S_i) = \{\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1})\} \cup \{Y \in \mathscr{C}(S_{i-1}) \mid x_i \notin \text{Adj}\,(Y)\}.$$

Proof. The proof follows from Lemmas 3.6 and 3.7. □

We now establish an interesting property on the component partitionings $\mathscr{C}(S_i)$. Define the set

$$\mathscr{X} = \bigcup\{\mathscr{C}(S_i) \mid 1 \leqq i \leqq N\},$$

that is, $\mathscr{X}$ contains *all* the component subsets in *all* $\mathscr{C}(S_i)$.

THEOREM 3.9. $\mathscr{X} = \{\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}) \mid 1 \leqq i \leqq N\}$.

Proof. By Lemma 3.7, it suffices to show that

$$\mathscr{C}(S_k) \subset \{\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}) \mid 1 \leqq i \leqq N\}$$

for $k = 1, \cdots, N$. But this can be proved by induction on k using Theorem 3.8. □

There is therefore a one-to-one correspondence between the node set X and the component set $\mathscr{X}$. Indeed, the mapping is given by

$$x_i \quad \Leftrightarrow \quad \{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}).$$

This correspondence will be useful in establishing complexity bounds for the symbolic factorization algorithm. The following result relates the set $\mathscr{X}$ with Reach (x_i, S_{i-1}).

LEMMA 3.10. $\text{Reach}\,(x_i, S_{i-1}) = \text{Adj}\,(\{x_i\} \cup \text{Nbrhd}\,(x_i, S_{i-1}))$.

4. Symbolic factorization using the quotient graph model.

4.1. Symbolic factorization. Consider a symmetric matrix A with Cholesky factor L. Let $G^A = (X^A, E^A)$ be its associated graph, where

$$X^A = \{x_1, x_2, \cdots, x_N\}.$$

In view of the result of Lemma 3.1, symbolic factorization of A may be regarded as the determination of the sets

$$\text{Reach}_{G^A}\,(x_i, \{x_1, \cdots, x_{i-1}\})$$

for $i = 1, \cdots, N$.

These reachable sets in the graph G^A can, however, be determined in terms of the structure in the quotient graphs. The correspondence is given in Theorem 3.3. With this connection, we can state a symbolic factorization algorithm in terms of quotient graphs.

Step 0. (Initialization) Let $S_0 = \varnothing$, $\mathcal{G}_0 = G^A/\bar{\mathcal{C}}(S_0)$.
Also put $i \leftarrow 1$.

Step 1. (Reachable set determination) Find the reach set

$$\text{Reach}_{\mathcal{G}_{i-1}}(\{x_i\}, \mathcal{C}(S_{i-1}))$$

in the quotient graph $\mathcal{G}_{i-1}$.

Step 2. (Quotient graph transformation) Form $S_i \leftarrow S_{i-1} \cup \{x_i\}$ and $\mathcal{C}(S_i)$ as given by Theorem 3.8.
Perform the in-place transformation [5] of the quotient graph $\mathcal{G}_i$ from $\mathcal{G}_{i-1}$, where

$$\mathcal{G}_i = G^A/\bar{\mathcal{C}}(S_i).$$

Step 3. (Loop or stop) Set $i \leftarrow i+1$. If $i > N$, stop; otherwise go to Step 1.

To consider the complexity of the algorithm, we first study the overall contribution from Step 1. The next lemma shows that it is bounded by $O(|E^F|)$. In the next section, we shall show that Step 2 can be implemented with the same bound. Thus, it is an $O(|E^F|)$ algorithm.

LEMMA 4.1. *The overall complexity in the reachable set determination step is* $O(|E^F|)$.

Proof. Consider the ith step in determining

$$\text{Reach}_{\mathcal{G}_{i-1}}(\{x_i\}, \mathcal{C}(S_{i-1})).$$

This can be done by inspecting

$$\text{Adj}_{\mathcal{G}_{i-1}}(\{x_i\}) \qquad \text{and} \qquad \text{Adj}_{\mathcal{G}_{i-1}}(Y),$$

for every $Y \in \mathcal{C}(S_{i-1}) \cap \text{Adj}_{\mathcal{G}_{i-1}}(\{x_i\})$. By Theorem 3.5,

$$\sum_i |\text{Adj}_{\mathcal{G}_{i-1}}(\{x_i\})| \leqq \sum_i |\text{Adj}_G(x_i)| = O(|E^A|).$$

On the other hand, by Theorem 3.9 and Lemma 3.10,

$$\begin{aligned}\sum_i \{|\text{Adj}_{\mathcal{G}_{i-1}}(Y)| \mid Y \in \mathcal{C}(S_{i-1}) \cap \text{Adj}_{\mathcal{G}_{i-1}}(\{x_i\})\}\\ = \sum \{|\text{Adj}(Y)| \mid Y \in \mathcal{X}\}\\ = \sum \{|\text{Reach}(x_i, S_{i-1})| \mid 1 \leqq i \leqq N\}\\ = O(|E^F|).\end{aligned}$$

Combining, we have shown that the complexity is $O(|E^F|)$. □

4.2. Incomplete quotient graph transformation. In this section, we shall introduce the technique of incomplete transformation, which helps to reduce the amount of work in Step 2 of symbolic factorization as described in § 4.1. This basic idea is similar to that exploited by Rose et al. [8] and Sherman [9], although here the technique is presented in the context of quotient graph transformations. We first prove a lemma. Recall that

$$\mathcal{X} = \cup \{\mathcal{C}(S_i) \mid 1 \leqq i \leqq N\}.$$

LEMMA 4.2. *Let $Y \in \mathscr{X}$. Then $Y \in \mathrm{Adj}_{\mathscr{G}_{i-1}}(\{x_i\})$ if and only if*

$$\min\{k \mid x_k \in \mathrm{Adj}_G(Y)\} = i.$$

Proof. (if). Assume $\min\{k \mid x_k \in \mathrm{Adj}_G\,(Y)\} = i$. By Theorem 3.9, let

$$Y = \{x_j\} \cup \mathrm{Nbrhd}\,(x_j, S_{j-1}).$$

Since the minimum subscript of neighbors of Y is i, by Theorem 3.8, we have

$$Y \in \mathscr{C}(S_j) \cap \mathscr{C}(S_{j+1}) \cdots \cap \mathscr{C}(S_{i-1}).$$

In other words, $Y \in \bar{\mathscr{C}}(S_{i-1})$ so that $Y \in \mathrm{Adj}_{\mathscr{G}_{i-1}}(\{x_i\})$.

(only if). Assume $Y \in \mathrm{Adj}_{\mathscr{G}_{i-1}}(\{x_i\})$. Clearly we have $x_i \in \mathrm{Adj}_G\,(Y)$. If $Y = \{x_j\} \cup \mathrm{Nbrhd}\,(x_j, S_{j-1})$, we have $Y \in \mathscr{C}(S_j) \cap \cdots \cap \mathscr{C}(S_{i-1})$. That means $\mathrm{Adj}\,(Y) \cap (x_{j+1}, \cdots, x_{i-1}\} = \emptyset$, and hence the result. □

The idea of incomplete quotient graph transformation is based on Lemma 4.2. In performing the transformation

$$\mathscr{G}_{i-1} = (\bar{\mathscr{C}}(S_{i-1}), \mathscr{E}_{i-1}) \rightarrow \mathscr{G}_i = (\bar{\mathscr{C}}(S_i), \mathscr{E}_i),$$

if $Y = \{x_i\} \cup \mathrm{Nbrhd}\,(x_i, S_{i-1})$, the complete transformation requires the setting up of

$$\mathrm{Adj}_{\mathscr{G}_i}\,(Y) = \{\{x\} \mid x \in \mathrm{Adj}_G\,(Y)\}$$

and for $x \in \mathrm{Adj}_G\,(Y)$,

$$\mathrm{Adj}_{\mathscr{G}_i}\,(\{x\}) = (\mathrm{Adj}_{\mathscr{G}_{i-1}}\,(\{x\}) \cup \{Y\}) \cap \bar{\mathscr{C}}(S_i).$$

However, in view of Lemma 4.2, we do not have to form all the $\mathrm{Adj}_{\mathscr{G}_i}\,(\{x\})$ for every $x \in \mathrm{Adj}_G\,(Y)$. Instead, the neighbors update need only be performed on the $x \in \mathrm{Adj}_G\,(Y)$ with the *smallest subscript*. Thus, we have proved the following result.

LEMMA 4.3. *The overall complexity in the incomplete quotient graph step is $O(|E^F|)$.*

4.3. Mass symbolic elimination. The results of Lemmas 4.1 and 4.3 imply that the symbolic factorization process can be implemented in time proportional to $O(|E^F|)$. In this section, we discuss another enhancement so that the time complexity becomes $O(|\mathscr{R}_L|)$, where $\mathscr{R}_L$ is the representation used to describe the structure of L, discussed in § 2.

The idea for the enhancement is motivated from an implementation of the minimum degree algorithm by the authors [4]. We first quote some results about reachable sets.

Let $G = (X, E)$. Consider a subset $S \subset X$, and a node $y \notin S$.

LEMMA 4.4. *Let $x \in X - S$. If*

$$\mathrm{Adj}\,(x) \subset \mathrm{Closure}\,(y, S),$$

then

$$\mathrm{Reach}\,(x, S) \subset \mathrm{Reach}\,(y, S) \cup \{y\}.$$

COROLLARY 4.5. *Let x be as in Lemma 4.4. Then*

$$\mathrm{Reach}\,(x, S \cup \{y\}) \subset \mathrm{Reach}\,(y, S).$$

Proof. Consider any $u \in \mathrm{Reach}\,(x, S \cup \{y\})$. Clearly $u \neq y$. If u can be reached from x via S, it follows from Lemma 4.4 that $u \in \mathrm{Reach}\,(y, S)$. Otherwise, the path from x to u via $S \cup \{y\}$ has to go through the node y; again this implies $u \in \mathrm{Reach}\,(y, S)$. □

LEMMA 4.6. *If $x \in \mathrm{Reach}\,(y, S)$, then*

$$\mathrm{Reach}\,(x, S \cup \{y\}) \supset \mathrm{Reach}\,(y, S) - \{x\}.$$

Proof. Consider any $u \in \text{Reach}(y, S)-\{x\}$. There exists a path $u, s_1, \cdots, s_t, y$ where $\{s_1, \cdots, s_t\} \subset S$. However, $x \in \text{Reach}(y, S)$, which implies there exists one from y to x through S,

$$y, \bar{s}_1, \cdots, \bar{s}_r, x.$$

By joining the two paths, we see that u is reachable from x through $S \cup \{y\}$.

THEOREM 4.7. *Let $x \in X-S$. If*

$$x \in \text{Reach}(y, S)$$

and

$$\text{Adj}(x) \subset \text{Closure}(y, S)$$

then

$$\text{Reach}(s, S \cup \{y\}) = \text{Reach}(y, S)-\{x\}.$$

Proof. The proof is a direct consequence of Corollary 4.5 and Lemma 4.6. □

The result in Theorem 4.7 can be used to speed up the symbolic factorization algorithm. After the reachable set

$$\text{Reach}_{\mathcal{G}_{i-1}}(\{x_i\}, \mathcal{C}(S_{i-1}))$$

has been determined at Step 2, the two conditions in Theorem 4.7 can be tested for the node $\{x_{i+1}\}$ in $\mathcal{G}_{i-1}$. If they are satisfied, we have immediately the reach set for $\{x_{i+1}\}$. This can be applied repeatedly to

$$\{x_{i+1}\}, \{x_{i+2}\}, \cdots$$

until one that violates either of the two cọnditions is encountered. Only then, the quotient graph transformation step is performed.

In this way, we need only examine the adjacent sets $\text{Adj}(x_{i+1})$, $\text{Adj}(x_{i+2}), \cdots$ in order to find the reachable sets

$$\text{Reach}(x_{i+1}, S_i), \text{Reach}(x_{i+2}, S_{i+1}), \cdots.$$

Moreover, these reachable sets can be represented implicitly by that of x_i. Thus, a set

$$\mathcal{M}^* = \{x_{i_1}, x_{i_2}, \cdots, x_{i_r}\}$$

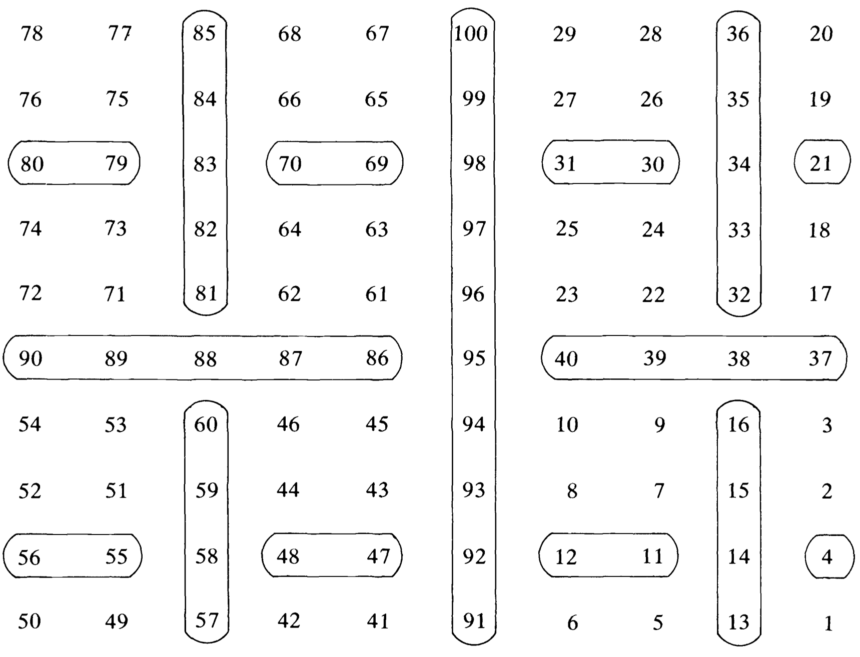
of representatives is defined naturally by the algorithm, where each x_{i_k} represents the immediately succeeding nodes until the next representative $x_{i_{k+1}}$.

Let $|\mathcal{R}_L| = \sum_{x_i \in \mathcal{M}^*} |\text{Reach}(x_i, S_{i-1})|$. It follows then that the complexity of the improved algorithm is $O(|E^A|+|\mathcal{R}_L|)$. In the next section, we consider a practical example where the improvement is significant.

5. An example—Nested dissection on an *n*×*n* grid. To demonstrate the effectiveness of the algorithm in § 4, we consider the nested dissection ordering [3], [6] of the $n \times n$ regular grid. Figure 5.1 shows such an ordering on the 10×10 grid problem.

It has been established that the nested dissection strategy produces orderings that are optimal in the order of magnitude sense. With such orderings, the amount of arithmetic required to factor the matrix problem is $O(n^3)$ and the number of nonzeros in the factored matrix is $O(n^2 \log n)$.

We now consider the symbolic factorization of the grid problem with such orderings. On applying the improved algorithm in § 4, we note that the nodes in the "dissectors" (nodes grouped by encircling lines in Fig. 5.1) satisfy the conditions in



78	77	85	68	67	100	29	28	36	20
76	75	84	66	65	99	27	26	35	19
80	79	83	70	69	98	31	30	34	21
74	73	82	64	63	97	25	24	33	18
72	71	81	62	61	96	23	22	32	17
90	89	88	87	86	95	40	39	38	37
54	53	60	46	45	94	10	9	16	3
52	51	59	44	43	93	8	7	15	2
56	55	58	48	47	92	12	11	14	4
50	49	57	42	41	91	6	5	13	1

FIG. 5.1. *A nested dissection ordering of a* 10×10 *grid.*

Theorem 4.7. Thus, as far as the symbolic factorization is concerned, a dissector can be represented by the lowest subscripted node in it, and such representation is in fact set up by the enhanced algorithm.

With the so-defined representative set, let $s(n)$ be the number of nonzeros in the representative columns of the factored matrix for the $n\times n$ grid. That is, $s(n)$ is the corresponding size $|\mathcal{R}_L|$. It is easy to establish the following recursive equations:

$$s(n)\leqq \bar{s}(n) \quad \text{where } \bar{s}(n)\leqq 12n+4\bar{s}\left(\frac{n}{2}\right).$$

On solving the equations, we have

$$s(n)\leqq \bar{s}(n)\leqq 12n^2.$$

Together with the observation in § 4.3, we have proved that the symbolic factorization of an $n\times n$ grid problem ordered by nested dissection can be done in time proportional to n^2, even though the number of nonzeros in the triangular factor is $O(n^2 \log n)$.

6. Some numerical experiments and concluding remarks. In this section we present numerical experiments demonstrating the performance of our algorithm. We also discuss its advantages and disadvantages compared with a very good "conventional" implementation supplied in the Yale Sparse Matrix Package [2], which was kindly provided to us by Professor Stanley Eisenstat. In what follows, we refer to the Yale routine as SSF, and to ours as SFQG.

In order to demonstrate experimentally the results of § 4, we applied SFQG to a sequence of "graded L" problems taken from [6]. This is a set of similar problems of increasing size, typical of those arising in finite element applications. The ordering used was produced by the implementation of the minimum degree algorithm described in [4].

Execution times reported are in seconds on an IBM 360/75 computer. The program was written in Fortran, and the optimizing version of the H-level compiler was used. The results of the experiment are summarized in Table 6.1. As the theory developed in § 4 predicts, the execution time for SFQG appears to be proportional to $|\mathscr{R}_L|$.

TABLE 6.1

Execution time and $|\mathscr{R}_L|$ *for SFQG, along with their ratio, for the sequence of graded-L problems from* [4].

N	Time	$\lvert\mathscr{R}_L\rvert$	Time/$\lvert\mathscr{R}_L\rvert$
265	.16	1353	1.19
406	.24	2252	1.08
577	.33	3293	1.05
778	.48	4604	1.04
1009	.58	5842	.99
1270	.80	7856	1.02
1560	.95	9424	1.00
1882	1.19	11707	1.01
			$(\times 10^{-4})$

How does SFQG compare with a good conventional implementation of symbolic factorization, such as the subroutine SSF from [2]? To a substantial degree, their relative merits depend upon the computing environment.

We have already observed that the major advantage of our subroutine is that it "fails gracefully." Since it does not use its output during execution, if insufficient storage is available, the output can be discarded but the program can still execute, producing the number $|\mathscr{R}_L|$. Alternatively, the output $\mathscr{R}_L$ could be printed on an auxiliary file as it is generated, and the subroutine then only needs storage for G^A.

To be fair, we must point out that in some contexts this storage argument is not relevant. Some ordering algorithms can be quite easily modified so as to provide either $|\mathscr{R}_L|$ or a good upper bound, so that we can be assured when we use either SSF or SFQG with such orderings that they will not fail. However, it is not clear that all ordering algorithms can efficiently provide a good upper bound for $|\mathscr{R}_L|$; for example, we do not know how to appropriately modify the automatic nested dissection algorithm in [6] so that an inexpensive estimate for $|\mathscr{R}_L|$ is provided.

In terms of execution times our implementation is somewhat slower than SSF for small problems. The distributed version of SSF, applied to the graded-L problems of Table 6.1, produced the times shown in Table 6.2, along with which we have included the SFQG times from Table 6.1. After discussing our work with Professor Eisenstat, he showed us how to modify SSF so that it too employed our "mass elimination" technique, described in § 4.3. The column in Table 6.2 labeled SSF* contains the execution times of this modified subroutine.

The execution time of SSF is apparently growing faster than $|\mathscr{R}_L|$, so for large enough problems, the SFQG subroutine would be faster than SSF. However, the improved version SSF* appears to execute in time proportional to $|\mathscr{R}_L|$, and continues to enjoy a substantial execution time advantage over SFQG, even for the larger problems. We should note that neither SSF nor SSF*, as implemented, can be *proved* to run in $O(|\mathscr{R}_L|)$ time. Modifications to these subroutines which allow the complexity bounds to be established considerably increase their execution times.

TABLE 6.2

Comparison of execution times of SFQG, SSF and a modified version of SSF.

	SFQG		SSF		SSF*	
N	Time	Time/$\lvert\mathscr{R}_L\rvert$	Time	Time/$\lvert\mathscr{R}_L\rvert$	Time	Time/$\lvert\mathscr{R}_L\rvert$
265	.16	1.19	.01	.07	.08	.06
406	.24	1.08	.17	.75	.14	.62
577	.33	1.05	.26	.78	.20	.60
778	.48	1.04	.37	.80	.27	.59
1009	.58	.99	.47	.80	.35	.60
1270	.80	1.02	.64	.81	.46	.59
1561	.95	1.00	.82	.87	.57	.60
1882	1.19	1.01	1.04	.88	.68	.58
		$(\times 10^{-4})$		$(\times 10^{-4})$		$(\times 10^{-4})$

Thus, to summarize, our implementation of symbolic factorization, based on quotient graphs, has some advantages in terms of storage and being able to "fail gracefully." In particular, since its execution is independent of its output, it is very attractive when storage is scarce, and auxiliary storage devices must be used. In exchange, it appears to execute more slowly than the best conventional implementation of which we are aware.

It is important for the reader to understand the sense in which our implementation is "optimal." We have shown only that it is optimal in the sense that it executes in time proportional to the size of its output, *but we have not shown that its output is optimal.* For purposes of data structure construction, it is desirable to produce an $\mathscr{R}_L$ having as few members as possible because this will tend to reduce the "overhead" storage requirements of the data structure for L. It is not difficult to construct examples where the $|\mathscr{R}_L|$ produced by our algorithm is larger than necessary. (In particular, note that SFQG only generates monotone representative maps $\mathscr{M}$.) Investigation into the development of an algorithm which is optimal in both of the above senses is an interesting area of future research.

REFERENCES

[1] C. BERGE, *The Theory of Graphs and its Applications*, John Wiley, New York, 1962.

[2] S. C. EISENSTAT, M. C. GURSKY, H. M. SCHULTZ AND A. H. SHERMAN, *Yale sparse matrix package I—The Symmetric Codes*, Research Report #112, Dept. of Computer Science, Yale University, 1977.

[3] ALAN GEORGE, *Numerical experiments using dissection methods to solve n by n grid problems*, SIAM J. Numer. Anal., 14 (1977), pp. 161–180.

[4] ALAN GEORGE AND JOSEPH W. H. LIU, *A minimal storage implementation of the minimum degree algorithm*, Ibid., to appear.

[5] ———, *A quotient graph model for symmetric factorization*, Research Report CS-78-04, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, February 1978.

[6] ———, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[7] S. V. PARTER, *The use of linear graphs in Gauss elimination*, SIAM Rev. 3 (1961), pp. 364–369.

[8] D. J. ROSE, R. E. TARJAN AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, this Journal, 5 (1975), pp. 266–283.

[9] A. H. SHERMAN, *On the efficient solution of sparse systems of linear and nonlinear equations*, Research Rept. #46, Dept. of Computer Science (doctoral thesis), Yale University, 1975.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0012 $01.00/0

DESIGN AND ANALYSIS OF A DATA STRUCTURE FOR REPRESENTING SORTED LISTS*

MARK R. BROWN† AND ROBERT E. TARJAN‡

Abstract. In this paper we explore the use of 2-3 trees to represent sorted lists. We analyze the worst-case cost of sequences of insertions and deletions in 2-3 trees under each of the following three assumptions: (i) only insertions are performed; (ii) only deletions are performed; (iii) deletions occur only at the small end of the list and insertions occur only away from the small end. Our analysis leads to a data structure for representing sorted lists when the access pattern exhibits a (perhaps time-varying) locality of reference. This structure has many of the properties of the representation proposed by Guibas, McCreight, Plass and Roberts [*A new representation for linear lists*, Proc. Ninth Annual Symposium on Theory of Computing, Boulder, CO, 1977, pp. 49–60], but it is substantially simpler and may be practical for lists of moderate size.

Key words. analysis of algorithms, deletion, finger, insertion, sorted list, 2-3 tree

Introduction. The 2-3 tree [1] is a data structure which allows both fast accessing and fast updating of stored information. For example, 2-3 trees may be used to represent a sorted list of length n so that a search for any item in the list takes $O(\log n)$ steps. Once the position to insert a new item or delete an old one has been found (via a search), the insertion or deletion can be performed in $O(\log n)$ additional steps.

If each insertion or deletion in a 2-3 tree is preceded by a search requiring $\Omega(\log n)$ time,[1] then there is little motivation for improving the above bounds on the worst-case time for insertions and deletions. But there are several applications of 2-3 trees in which the regularity of successive insertions or deletions allows searches to proceed faster than $\Omega(\log n)$. One example is the use of a sorted list represented as a 2-3 tree to implement a priority queue [6, p. 152]. In a priority queue, insertions are allowed anywhere, but only the smallest item in the list at any moment can be deleted. Since no searching is ever required to find the next item to delete, an improved bound on the cost of consecutive deletions might lead to a better bound on the cost of the method as a whole.

In this paper, we prove several results about the cost of sequences of operations on 2-3 trees. In § 1 we derive a bound on the total cost of a sequence of insertions (as a function of the positions of the insertions in the tree) which is tight to within a constant factor. In § 2 we derive a similar bound for a sequence of deletions. If the sequence of operations is allowed to include intermixed insertions and deletions, there are cases in which the naive bound cannot be improved: $\theta(\log n)$ steps per operation may be required. However, we show in § 3 that for the priority queue application mentioned above, a mild assumption about the distribution of insertions implies that such bad cases cannot occur.

In § 4 we explore some consequences of these results. We propose a modification of the basic 2-3 tree structure which allows us to save a *finger* to an arbitrary position in the tree, with the property that searching d positions away from the finger costs $O(\log d)$ steps (independent of the tree size). Fingers are inexpensive to move, create,

* Received by the editors January 24, 1979.

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

‡ Computer Science Department, Stanford University, Stanford, California 94305. The work of this author was supported in part by the National Science Foundation under Grant MCS 75-22870 AO2, the Office of Naval Research under Contract N0014-76-C-0688, and a Guggenheim Fellowship.

[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_1, c_2, and n_0 with $c_1 f(n) \leqq g(n) \leqq c_2 f(n)$ for all $n \geqq n_0$. Hence the "θ" can be read "order exactly" and the "Ω" as "order at least"; Knuth [7] gives further discussion of the θ and Ω notations.

or abandon, and several fingers into the same structure can be maintained simultaneously. We use the bound on sequences of insertions to show that even when fingers are used to speed up the searches, the cost of a sequence of insertions is dominated by the cost of the searches leading to the insertions. The same result holds for a sequence of deletions and for a sequence of intermixed insertions and deletions satisfying the assumptions of § 3. Our structure is similar to one proposed earlier by Guibas, McCreight, Plass and Roberts [4], but it is much simpler to implement and may be practical for representing moderate-sized lists. Their structure has the interesting property that *individual* insertions and deletions are guaranteed to be efficient, while operations on our structure are efficient only when averaged over a sequence. Our structure has the compensating advantage that fingers are much easier to move. An obvious generalization of our structure to B-trees [2] makes it suitable for larger lists kept in secondary storage.

In the final section we discuss some practical issues arising in an implementation of the structure, describe some of its applications, and indicate directions for future work.

1. Insertions into 2-3 trees. A 2-3 tree [1], [6] is a tree such that 2- or 3-way branching takes place at every internal node, and all external nodes occur on the same level. An internal node with 2-way branching is called a *2-node*, and one with 3-way branching a *3-node*. It is easy to see that the height of a 2-3 tree with n external nodes lies between $\lceil \log_3 n \rceil$ and $\lfloor \lg n \rfloor$.[2] An example of a 2-3 tree is given in Fig. 1.

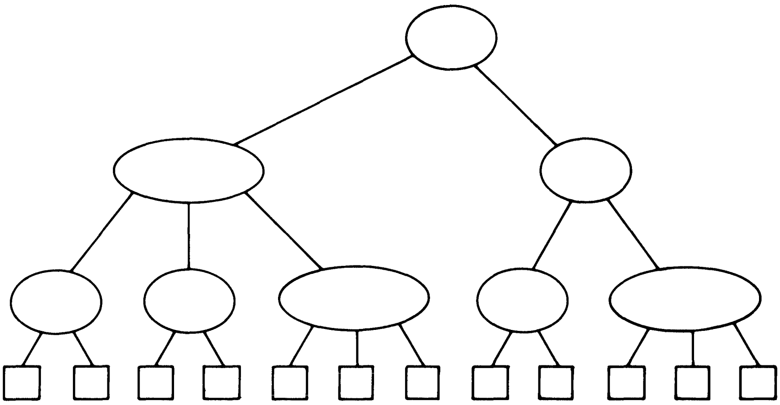


FIG. 1. *A 2-3 tree.*

There are several schemes for associating data with the nodes of a 2-3 tree; the usefulness of a particular organization depends upon the operations to be performed on the data. All of these schemes use essentially the same method for updating the tree structure to accomodate insertions, where *insertion* means the addition of a new external node at a given position in the tree. (Sometimes the operation of insertion is considered to include searching for the position to add the new node, but we shall consistently treat searches separately in what follows.)

Insertion is accomplished by a sequence of node expansions and splittings, as shown by example in Fig. 2. When a new external node is attached to a terminal node p (an internal node having only external nodes as offspring), this node *expands* to accomodate the extra edge. If p was a 2-node prior to the expansion, it is now a 3-node,

[2] We use lg n to denote $\log_2 n$.

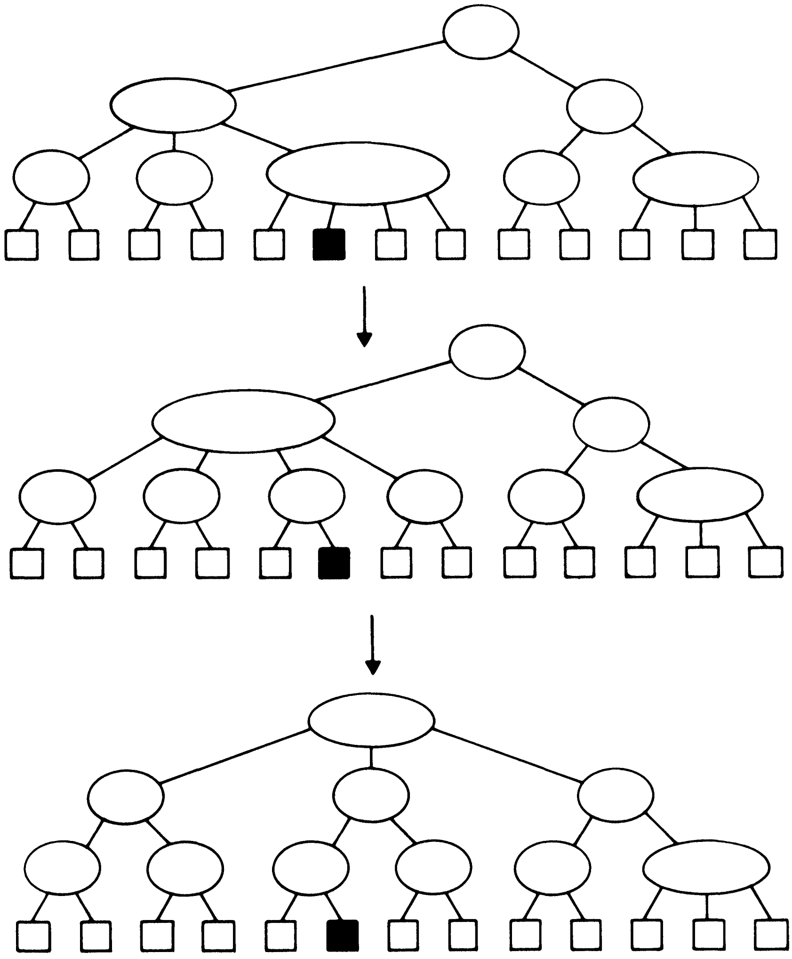


FIG. 2. *A* 2-3 *tree insertion.*

and the insertion is complete. If p was a 3-node prior to expansion, it is now a "4-node", which is not allowed in a 2-3 tree; therefore, p is *split* into a pair of 2-nodes. This split causes an expansion of p's parent, and the process repeats until either a 2-node expands into a 3-node or the root is split. If the root splits, a new 2-node is created which has the two parts of the old root as its children, and this new node becomes the root. An insertion in a 2-3 tree can be accomplished in $\theta(1+s)$ steps, where s is the number of node splittings which take place during the insertion.

One way to represent a sorted list using a 2-3 tree is shown in Fig. 3. The elements of the list are assigned to the external nodes of the tree, with key values of the list elements increasing from left to right. Keys from the list elements are also assigned to internal nodes of the tree in a "symmetric" order analogous to that of binary search trees. More precisely, each internal node is assigned one key for each of its subtrees other than the rightmost, this key being the largest which appears in an external node of the subtree. Therefore each key except the largest appears in an internal node, and by starting from the root of the tree we can locate any element of the list in $O(\log n)$ steps, using a generalization of binary tree search. (Several 2-3 search tree organizations have been proposed which are similar but not identical to this one [1, p. 147], [6, p. 468].)

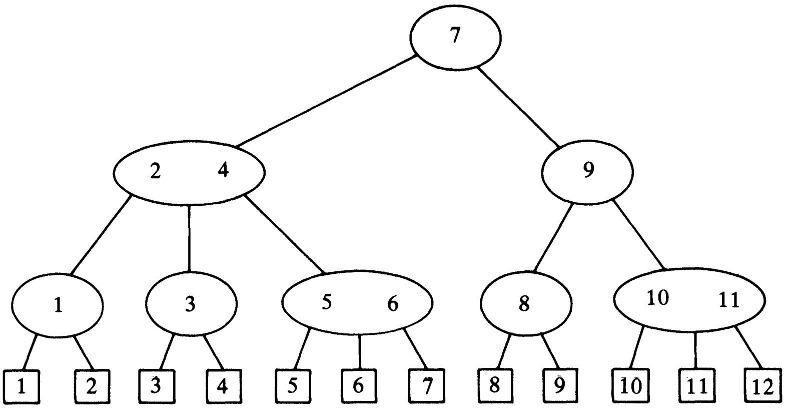


FIG. 3. *A* 2-3 *tree structure for sorted lists.*

Any individual insertion into a 2-3 tree of size n can cause up to about $\lg n$ splittings of internal nodes to take place. On the other hand, if n consecutive insertions are made into a tree initially of size n, the total number of splits is bounded by about $\frac{3}{2}n$ instead of $n \lg n$, because each split generates a new internal node and the number of internal nodes is initially at least $(n-1)/2$ and finally at most $2n-1$. The following theorem gives a general bound on the worst-case splitting which can occur due to consecutive insertions into a 2-3 tree.

THEOREM 1. *Let T be a* 2-3 *tree of size n, and suppose that k insertions are made into T. If the positions of the newly-inserted nodes in the resulting tree are $p_1 < p_2 < \cdots < p_k$, then the number of node splittings which take place during the insertions is bounded by*

$$2\left(\lceil \lg (n+k)\rceil + \sum_{1<i\leqq k} \lceil \lg (p_i - p_{i-1}+1)\rceil\right).$$

The proof divides into two parts. In the first part, we define a rule for (conceptually) *marking* nodes during a 2-3 tree insertion. This marking rule has two important properties when a sequence of insertions is made: the number of marked nodes bounds the number of splits, and the marked nodes are arranged to form paths from the inserted external nodes toward the root of the tree.

The effect of marking the tree in this way is to shift our attention from dealing with a dynamic situation (the 2-3 tree as it changes due to insertions) to focus on a static object (the 2-3 tree which results from the sequence of insertions). The second part of the proof then consists of showing that in any 2-3 tree, the number of nodes lying on the paths from the external nodes in positions $p_1 < p_2 < \cdots < p_k$ to the root is bounded by the expression given in the statement of the theorem.

We now define the marking rule described above. On each insertion into a 2-3 tree, one or more nodes are marked as follows:

(1) The inserted (external) node is marked.

(2) When a marked node splits, both resulting nodes are marked. When an unmarked node splits, a choice is made and one of the resulting nodes is marked; if possible, a node is marked which has a marked child.

We establish the required properties of these rules by a series of lemmas.

LEMMA 1. *After a sequence of insertions, the number of marked internal nodes equals the number of splits.*

Proof. No nodes are marked initially, and each split causes the number of marked internal nodes to increase by one. □

LEMMA 2. *If a 2-node is marked, then at least one of its children is marked; if a 3-node is marked, then at least two of its children are marked.*

Proof. We use induction on the number of marked internal nodes. Since both assertions hold vacuously when there are no marked internal nodes, it is sufficient to show that a single application of the marking rules preserves the assertions. There are two cases to consider when a 3-node X splits:

Case 1. X is marked. Then before the insertion which causes X to split, X has at least two marked children. When the insertion expands X to overflow, this adds a third marked child (by rule 1 or rule 2). Thus the two marked 2-nodes which result from the split of X each have at least one marked child.

Case 2. X is unmarked. Then before the insertion which causes X to split, X may have no marked children. When the insertion expands X to overflow, a new marked child is created. Thus the single marked 2-node which results from the split of X can be chosen to have a marked child.

A marked 3-node is created when a marked 2-node expands. This expansion always increases the number of marked children by one. Since a marked 2-node has at least one marked child, it follows that a marked 3-node has at least two marked children. □

LEMMA 3. *After a sequence of insertions, there is a path of marked nodes from any marked node to a marked external node.*

Proof. Obvious from Lemma 2. □

LEMMA 4. *The number of splits in a sequence of insertions is no greater than the number of internal nodes in the resulting tree which lie on paths from the inserted external nodes to the root.*

Proof. Immediate from Lemmas 1 and 3. □

This completes the first part of the proof as outlined earlier; to finish the proof we must bound the quantity in Lemma 4. We shall require the following two facts about binary arithmetic. For any nonnegative integer k, let $\nu(k)$ be the number of one-bits in the binary representation of k.

LEMMA 5 [5, p. 483 (answer to Ex. 1.2.6–11)]. *Let a and b be nonnegative integers, and let c be the number of carries when the binary representations of a and b are added. Then $\nu(a)+\nu(b)=\nu(a+b)+c$.*

LEMMA 6. *Let a and b be nonnegative integers such that $a<b$ and let i be the number of bits to the right of and including the leftmost bit in which the binary representations of a and b differ. Then $i \leqq \nu(a)-\nu(b)+2\lceil \lg (b-a+1)\rceil$.*

Proof. If k is any positive integer, the length of the binary representation of k is $\lceil \lg (k+1)\rceil$. Let c be the number of carries when a and $b-a$ are added. By Lemma 5, $\nu(a)+\nu(b-a)=\nu(b)+c$. When a and $b-a$ are added, at least $i-\lceil \lg (b-a+1)\rceil$ carries are required to produce a number which differs from a in the ith bit. Thus $i-\lceil \lg (b-a+1)\rceil \leqq c$. Combining inequalities, we find that

$$\begin{aligned} i &\leqq c+\lceil \lg (b-a+1)\rceil \leqq \nu(a)-\nu(b)+\nu(b-a)+\lceil \lg (b-a+1)\rceil \\ &\leqq \nu(a)-\nu(b)+2\lceil \lg (b-a+1)\rceil. \quad □ \end{aligned}$$

LEMMA 7. *Let T be a 2-3 tree with n external nodes numbered* $0, 1, \cdots, n-1$ *from left to right. The number M of nodes (internal and external) which lie on the paths from external nodes $p_1<p_2<\cdots<p_k$ to the root of T satisfies*

$$M \leqq 2\Big(\lceil \lg n\rceil + \sum_{1<i\leqq k} \lceil \lg (p_i-p_{i-1}+1)\rceil\Big).$$

Proof. For any two external nodes p and q, let $M(p, q)$ be the number of nodes which are on the path from q to the root but not on the path from p to the root. Since the path from p_1 to the root contains at most $\lceil \lg n \rceil + 1$ nodes, we have

$$M \leqq \lceil \lg n \rceil + 1 + \sum_{1<i\leqq k} M(p_{i-1}, p_i).$$

We define a *label* l for each external node as follows. If t is an internal node of T which is a 2-node, we label the left edge out of t with a 0 and the right edge out of t with a 1. If t is a 3-node, we label the left edge out of t with a 0 and the middle and right edges out of t with a 1. Then the label $l(p)$ of an external node p is the integer whose binary representation is the sequence of 0's and 1's on the path from the root to p.

Note that if p and q are external nodes such that q is the right neighbor of p, then $l(q) \leqq l(p) + 1$. It follows by induction that $l(p_i) - l(p_{i-1}) \leqq p_i - p_{i-1}$ for $1 < i \leqq k$.

Consider any two nodes p_{i-1}, p_i. Let t be the internal node which is farthest from the root and which is on the path from the root to p_{i-1} and on the path from the root to p_i. We must consider two cases.

Case 1. The edge out of t leading toward p_{i-1} is labeled 0 and the edge out of t leading toward p_i is labeled 1. Then $l(p_i) > l(p_{i-1})$. Furthermore $M(p_{i-1}, p_i)$, which is the number of nodes on the path from t to p_i (not including t), is equal to the number of bits to the right of and including the leftmost bit in which the binary representations of $l(p_{i-1})$ and $l(p_i)$ differ. By Lemma 6,

$$\begin{aligned} M(p_{i-1}, p_i) &\leqq \nu(l(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (l(p_i) - l(p_{i-1}) + 1)\rceil \\ &\leqq \nu(l(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (p_i - p_{i-1} + 1)\rceil. \end{aligned}$$

Case 2. The edge out of t leading toward p_{i-1} is labeled 1 and the edge out of t leading toward p_i is also labeled 1. Let $l'(p_{i-1})$ be the label of p_{i-1} if the edge out of t leading toward p_{i-1} is relabeled 0. Then $l(p_i) - l'(p_{i-1}) \leqq p_i - p_{i-1}$ and $l(p_i) > l'(p_{i-1})$. Furthermore $M(p_{i-1}, p_i)$ is equal to the number of bits to the right of and including the leftmost bit in which the binary representations of $l'(p_{i-1})$ and $l(p_i)$ differ. By Lemma 6,

$$\begin{aligned} M(p_{i-1}, p_i) &\leqq \nu(l'(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (l(p_i) - l'(p_{i-1}) + 1)\rceil \\ &\leqq \nu(l'(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (p_i - p_{i-1} + 1)\rceil \\ &\leqq \nu(l(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (p_i - p_{i-1} + 1)\rceil \end{aligned}$$

since $\nu(l(p_{i-1})) = \nu(l'(p_{i-1})) + 1$.

Substituting into the bound on M given above yields

$$M \leqq \lceil \lg n \rceil + 1 + \sum_{1<i\leqq k} (\nu(l(p_{i-1})) - \nu(l(p_i)) + 2\lceil \lg (p_i - p_{i-1} + 1)\rceil).$$

But much of this sum telescopes, giving

$$\begin{aligned} M &\leqq \lceil \lg n \rceil + 1 + \nu(l(p_1)) - \nu(l(p_k)) + 2 \sum_{1<i\leqq k} \lceil \lg (p_i - p_{i-1} + 1)\rceil \\ &\leqq 2\Big(\lceil \lg n \rceil + \sum_{1<i\leqq k} \lceil \lg (p_i - p_{i-1} + 1)\rceil\Big) \end{aligned}$$

(since $\nu(l(p_k)) \geqq 1$ and $\nu(l(p_1)) \leqq \lceil \lg n \rceil$ unless $k = 1$). This completes the proof of Lemma 7 and Theorem 1. □

The bound given in Theorem 1 is tight to within a constant factor; that is, for any n and k there is a 2-3 tree with n external nodes and some sequence of k insertions which causes within a constant factor of the given number of splits. We omit a proof of this fact.

2. Deletions from 2-3 trees. The operation of *deletion* from a 2-3 tree means the elimination of a specified external node from the tree. As with insertion, the algorithm for deletion is essentially independent of the particular scheme used for associating data with the tree's nodes.

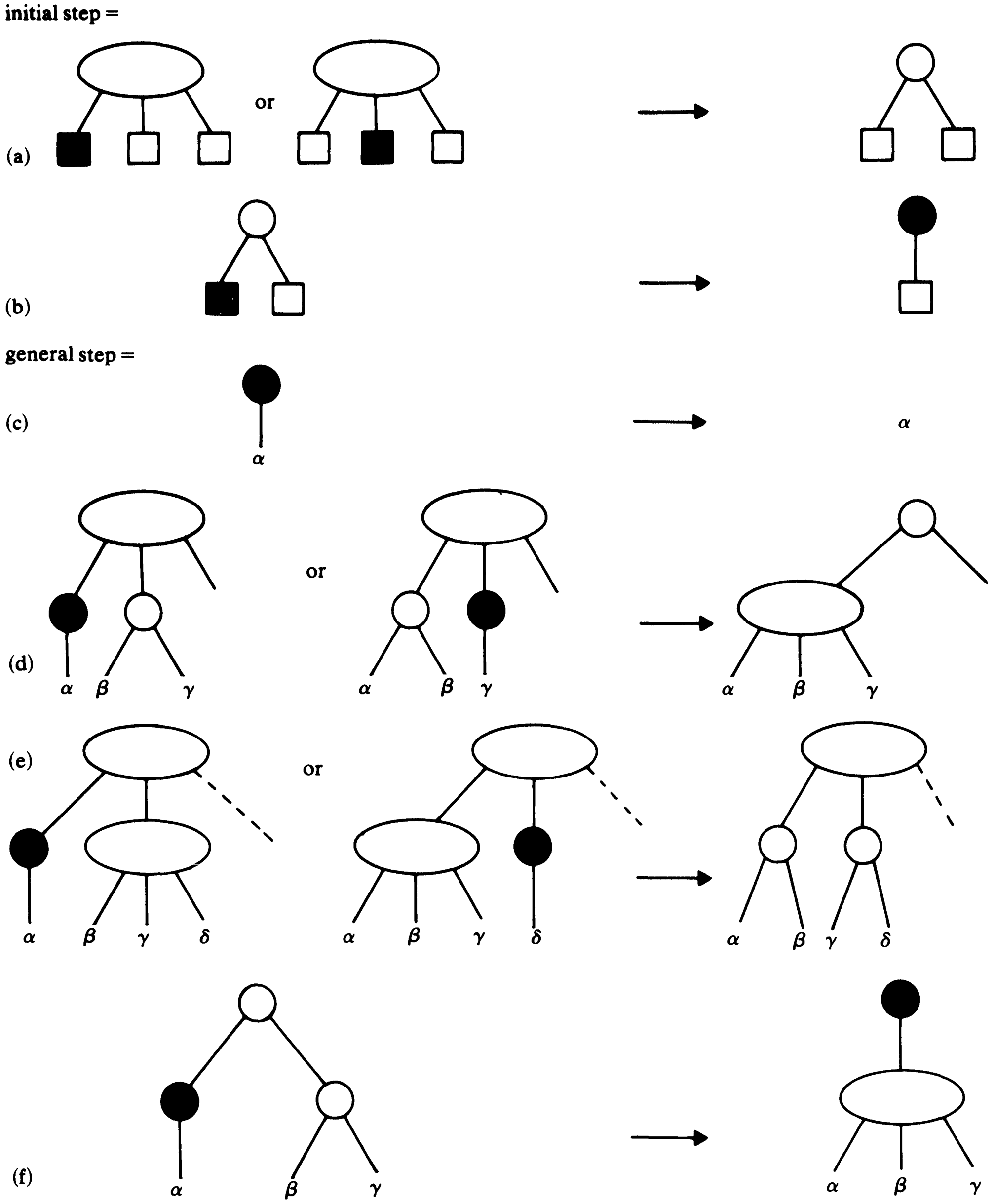


FIG. 4. *Transformations for 2-3 tree deletion. (Mirror-images of all transformations are possible.)*

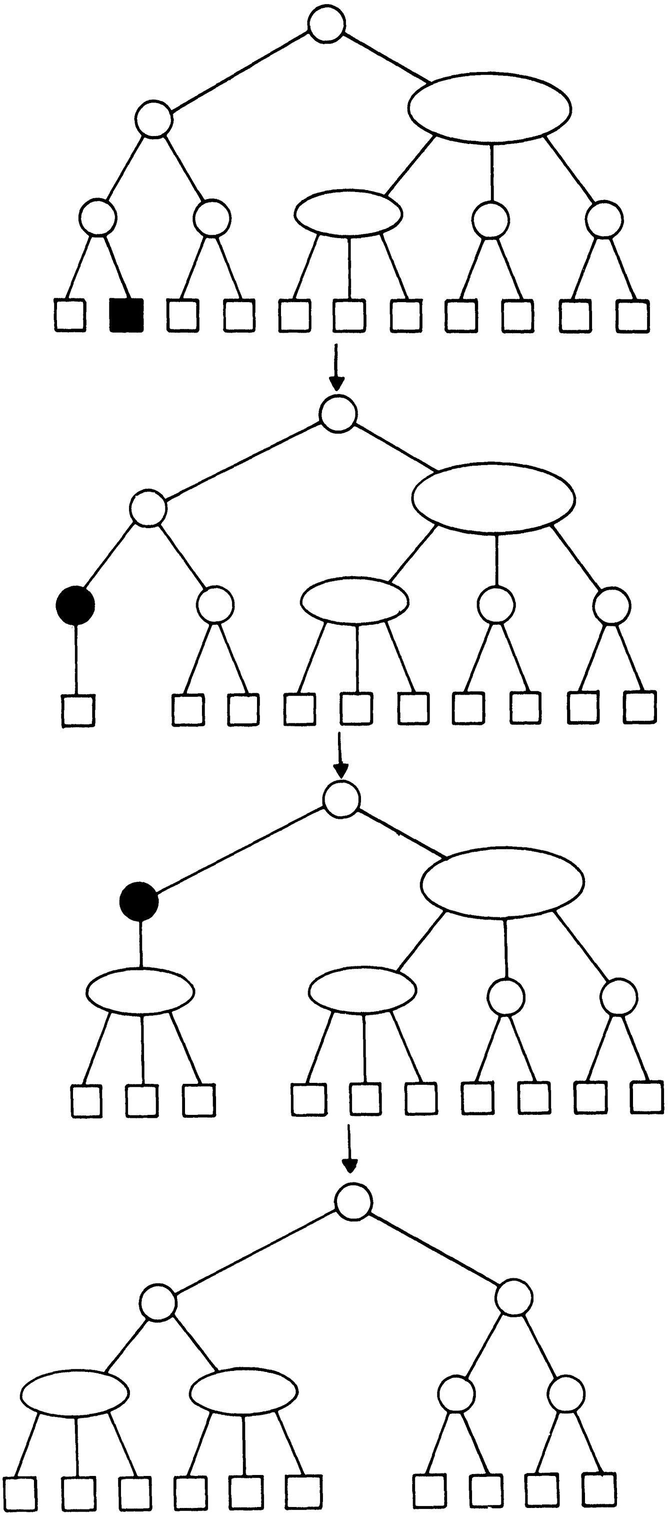


FIG. 5. *A* 2-3 *tree deletion.*

The first step of a deletion is to remove the external node being deleted. If the parent of this node was a 3-node before the deletion, it becomes a 2-node and the operation is complete. If the parent was a 2-node, it is now a "1-node", which is not allowed in a 2-3 tree; hence some additional changes are required to restore the tree. The local transformations shown in Fig. 4 are sufficient, as we shall now explain. If the 1-node is the root of the tree, it can simply be deleted, and its child is the final result (Fig. 4(c)). If the 1-node has a 3-node as a parent or as a sibling, then a local rearrangement will eliminate the 1-node and complete the deletion (Figs. 4(d), 4(e)). Otherwise we *fuse* the 1-node with its sibling 2-node (Fig. 4(f)); this creates a 3-node with a 1-node as parent. We then must repeat the transformations until the 1-node is eliminated. Figure 5 shows an example of a complete deletion.

A deletion in a 2-3 tree requires $O(1+f)$ steps, where f is the number of node fusings required for the deletion. Since the propagation of fusings up the path during a deletion is similar to the propagation of splittings during an insertion, it is not surprising that a result analogous to Theorem 1 holds for deletions.

THEOREM 2. *Let T be a 2-3 tree of size n, and suppose that $k \leqq n$ deletions are made from T. If the positions of the deleted external nodes in the original tree were $p_1 < p_2 < \cdots < p_k$, then the number of node fusings which took place during the deletions is bounded by*

$$2\Big(\lceil \lg n \rceil + \sum_{1<i\leqq k} \lceil \lg (p_i - p_{i-1} + 1) \rceil\Big).$$

Proof. We shall initially mark all nodes in T which lie on a path from the root of T to one of the deleted nodes. By Lemma 7, the number of marked nodes is bounded by the given expression; hence the proof is complete if we show that during the sequence of deletions it is possible to remove one mark from the tree for each fusing.

During the sequence of deletions, we shall maintain the invariant property that every 2-node on the path from a marked external node to the root is marked. This is clearly true initially. During a deletion, the marks are handled as indicated in Fig. 6. An "x" on the left side of a transformation indicates a node which the invariant (or a previous application of transformation (b) or (f)) guarantees will be marked; an "x" on the right side indicates a node to be marked after the transformation. These rules make only local rearrangements and create only *marked* 2-nodes, and hence they maintain the invariant. The fusing transformation (f) removes at least one mark from the tree. One of the terminating transformations (e) may create a new mark, but this is compensated by the starting transformation (b) which always destroys a mark. Hence a deletion always removes at least one mark from the tree per fusing, which proves the result. □

The bound of Theorem 2 is tight to within a constant factor; that is, for any n and $k \leqq n$ there is a 2-3 tree with n external nodes and a sequence of k deletions which causes within a constant factor of the given number of fusings. We omit a proof.

3. Mixed sequences of operations. When both insertions and deletions are present in a sequence of operations on a 2-3 tree, there are cases in which $\Omega(\log n)$ steps are required for each operation in the sequence. A simple example of this behavior is shown in Fig. 7, where an insertion causes splitting to go to the root of the tree, and deletion of the inserted element causes the same number of fusings. We expect that when insertions and deletions take place in separate parts of the tree, it is impossible for them to interact in this way. The following result shows that this intuition is justified, at least for a particular access pattern arising from priority queues.

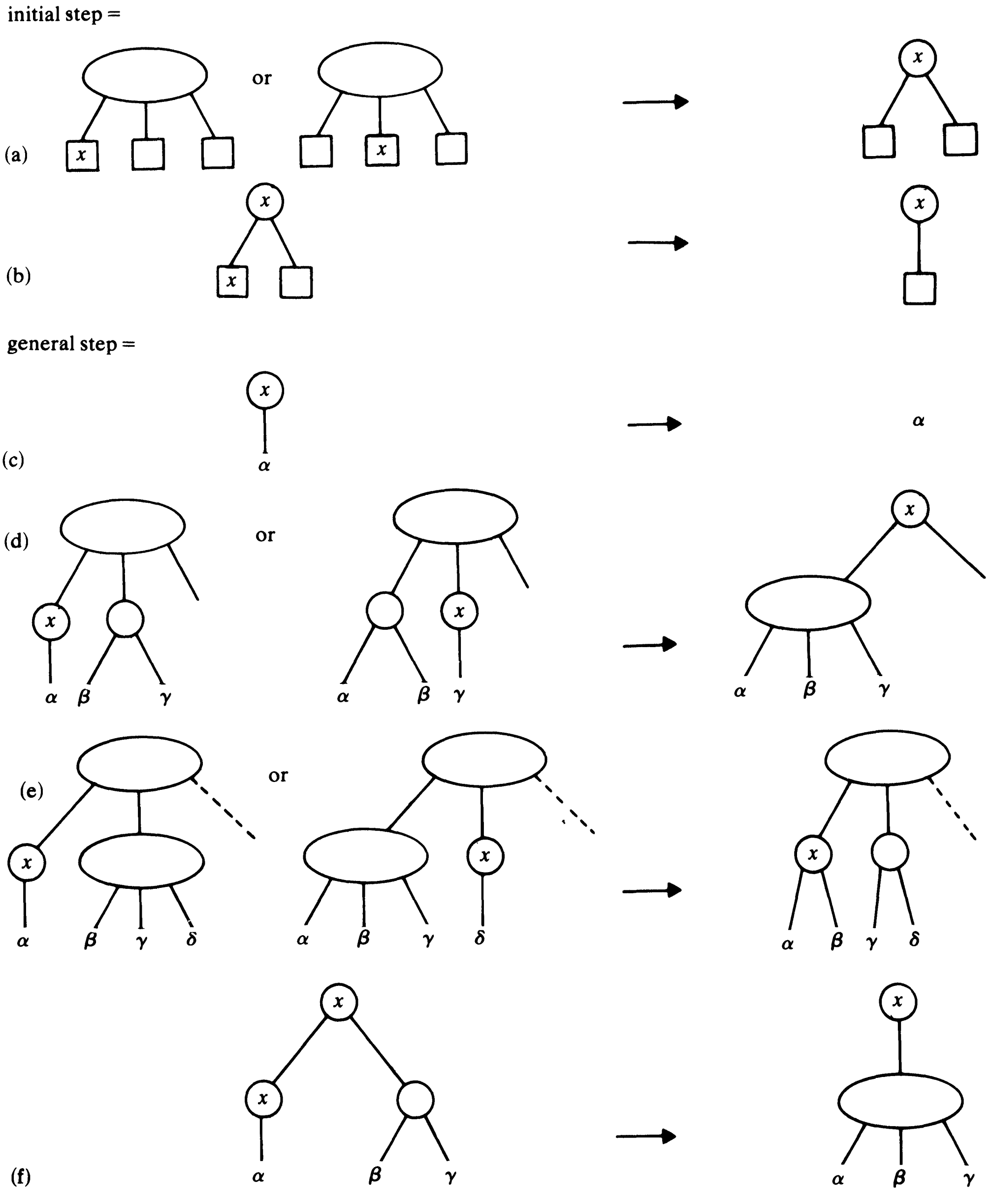


FIG. 6. *Deletion transformations for proof of Theorem* 2.

THEOREM 3. *Let T be a* 2-3 *tree of size n, and suppose that a sequence of k insertions and l deletions is performed on T. If all deletions are made on the leftmost external node of T, and no insertion is made closer than* $(\lg m)^{1.6}$ *positions from the point of the deletions* (*where m is the tree size when the insertion takes place*), *then the total cost of the operations is*

$$O\Big(\log n + k + l + \sum_{1<i\leqq k'} \log (p_i - p_{i-1})\Big),$$

where $k' \leqq k$ *is the number of inserted nodes that have not been deleted and* $p_1 < p_2 < \cdots < p_{k'}$ *are the positions of these nodes in the final tree.*

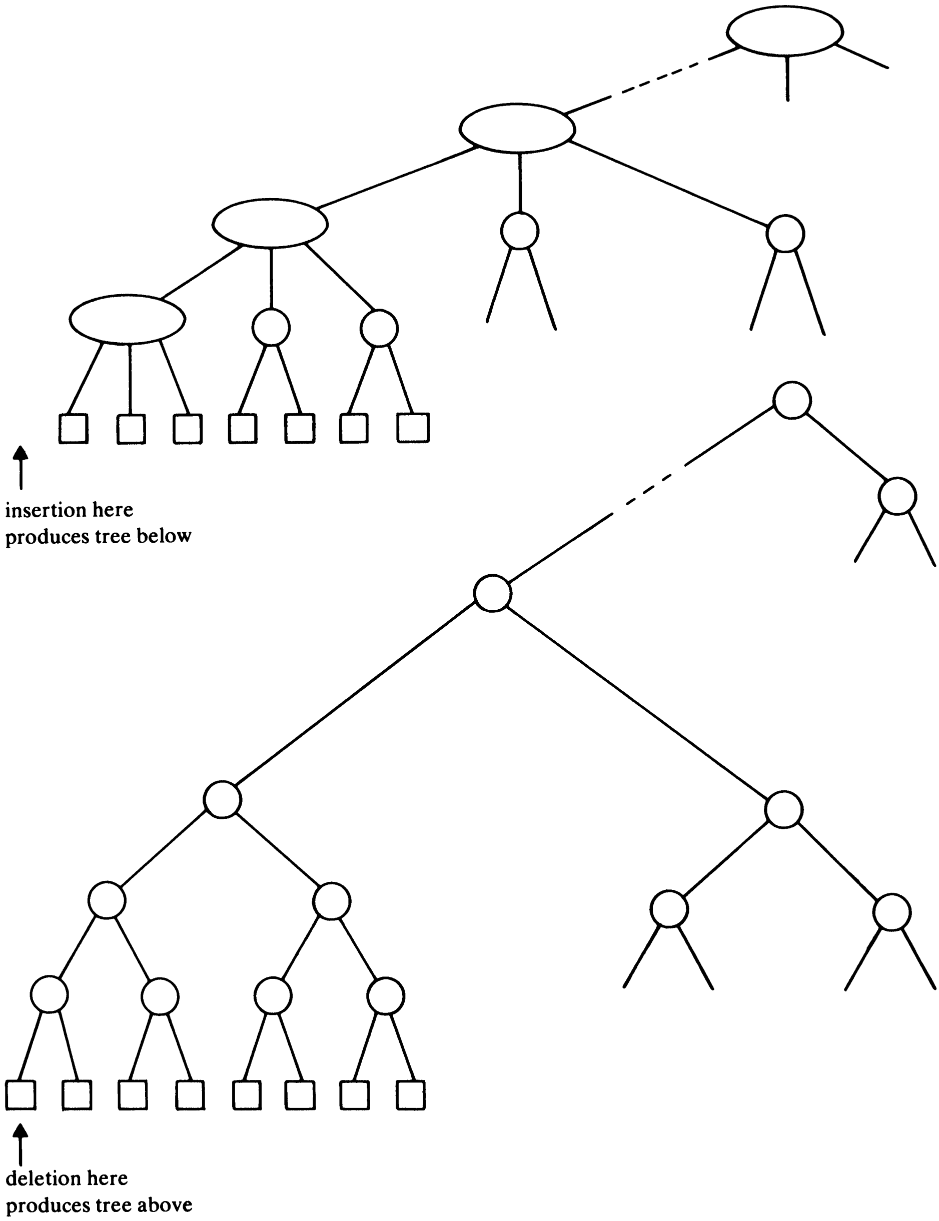


FIG. 7. *An expensive insert/delete pair.*

Proof. We shall first sketch the argument and then give it in more detail. Insertions are accounted for by marking the tree in a manner almost identical to that used in proving Theorem 1. Deletions may destroy some of these marks, so we charge a deletion for the marks it removes; the remaining marks are then counted using Lemma 7. Because we assume that insertions are bounded $(\lg m)^{1.6}$ positions away from the point of deletions, the left path is unaffected by insertions up to a height of at least $\lg \lg m$. Therefore roughly $\lg m$ deletions occur between successive deletions that reference an "unprotected" section of the left path. These $\lg m$ deletions cost $O(\log m)$ altogether, as does a single deletion that goes above the protected area, so l deletions cost roughly $O(l)$ steps to execute. Adding this to the cost of the insertions gives the bound.

We shall present the full argument as a sequence of lemmas. First we need some terminology. The *left path* is the path from the root to the leftmost external node. Note that deletions will involve only left-path nodes and the children of such nodes. We say that an insertion *changes* the left path if it splits a 3-node or expands a 2-node on the left path.

LEMMA 8. *Under the assumptions of Theorem* 3, *the cost of the sequence of insertions is*

$$O\Big(\log n + k + \sum_{1 \leqq i \leqq k'} \log (p_i - p_{i-1})\Big) + O(\textit{cost of deletions}).$$

Proof. On each insertion, we mark the nodes of T according to rules (1) and (2) in the proof of Theorem 1, while observing the following additional rule:

(3) When a marked 2-node on the left path expands, an *unmarked* 3-node is created.

As in the proof of Theorem 1, the cost of all insertions is bounded by the number of marks created using rules (1) and (2). Rule (3), which destroys a mark, can be applied at most once per insertion, and hence the number of marks removed by this rule is $O(k)$.

This marking scheme preserves the property that on the left path, no 3-node ever becomes marked. It does not preserve any stronger properties on the left path; for example, a marked 2-node with no marked offspring may occur. But it is easy to prove by induction on the number of insertion steps that the stronger properties used in the proof of Theorem 1 (a marked 2-node has at least one marked offspring, a marked 3-node has at least two marked offspring) *do* hold on the rest of the tree. The intuitive reason why the corruption on the left path cannot spread is that it could do so only through the splitting of 3-nodes on the path; since these nodes aren't marked, they never create "unsupported" 2-nodes off the left path.

The motivation for these marking rules is that deletions will necessarily corrupt the left path. During deletions, we treat marks according to the following rule:

(4) Any node involved in a deletion transformation (i.e., any node shown explicitly in Fig. 4) is unmarked during the transformation.

This rule removes a bounded number of marks per step, and hence over l deletions the number of marks removed is O(cost of deletions). Since this rule never creates a marked node, it preserves the property of no marked 3-nodes on the left path. It also preserves the stronger invariants on the rest of the tree, since it will only unmark a node whose parent is on the left path.

It follows that after the sequence of insertions and deletions, all marked nodes lie on paths from the inserted external nodes to the root, except possibly some marked 2-nodes on the left path. The number of nodes on the left path is $O(\log(n+k-l))$, and by Lemma 7 the number of marked nodes in the rest of the tree is

$$O\Big(\log(n+k-l) + \sum_{1 < i \leqq k'} \log(p_i - p_{i-1} + 1)\Big).$$

Adding these bounds to our previous estimates for the number of marks removed by rules (3) and (4), and noting that $\lg(x+y) \leqq \lg x + y$ for $x, y \geqq 1$, gives the result. □

LEMMA 9. *Suppose that a sequence of j deletions is made on the leftmost external node of a* 2-3 *tree, such that the deletions do not reference any left-path nodes changed by an insertion made during the sequence. Then the cost of the sequence is $O(j) + O$ (height of the tree before the deletions).*

Proof. The cost of a deletion is $O(1+f)$, where f is the number of fusings required. Each fusing destroys a 2-node on the left path, so the total cost of the j deletions is $O(j)+O$(number of left-path 2-nodes destroyed). But each deletion creates at most one left-path 2-node, and insertions do not create any 2-nodes that are referenced by the deletions, so the cost is in fact $O(j)$ + O (number of originally present left-path 2-nodes destroyed). This is bounded by the quantity given above. □

LEMMA 10. *Under the assumptions of Theorem* 3, *if the tree T has size m then an insertion cannot change any left-path node of height less than* $\lg \lg m$.

Proof. A 2-3 tree of height h contains at most 3^h external nodes. Hence a subtree of height $\lg \lg m$ contains $\leqq 3^{\lg \lg m} = (\lg m)^{\lg 3}$ external nodes, which is strictly less than the $(\lg m)^{1.6}$ positions that are protected from insertions under the conditions of Theorem 3. □

LEMMA 11. *Suppose that the bottommost k nodes on the left path are all* 3*-nodes, and deletions are performed on the leftmost external node. If insertions do not change any nodes of height $\leqq k$ on the left path, then at least 2^k deletions are required to make a deletion reference above height k on the left path.*

Proof. Let us view the left path as a binary integer, where a 2-node is represented by a zero and a 3-node by a one, and the root corresponds to the most significant bit. Then deletion of the leftmost external node corresponds roughly to subtracting one from this binary number. Consideration of the deletion algorithm shows that the precise effect is as follows: if the left path is $xx\cdots x1$ then a deletion causes it to become $xx\cdots x0$ (subtraction of 1), and if the path is

$$xx\cdots x1\overbrace{00\cdots 0}^{i}$$

then it becomes either

$$xx\cdots x0\overbrace{11\cdots 1}^{i}\quad\text{(subtraction of 1)}\quad\text{or}\quad xx\cdots x10\overbrace{1\cdots 1}^{i-1}\quad\text{(addition of }2^{i-1}-1).$$

Only this final possibility (corresponding to using the transformation in Fig. 4(e)) differs from subtraction by one. Note that under these rules everything to the left of the rightmost one-bit is unreferenced by a deletion.

Before a deletion reference above height k can take place, the number represented by the rightmost k bits must be transformed from 2^k-1 into 0 by operations which either subtract one or add a positive number. Thus 2^k-1 subtractions are required, corresponding to 2^k-1 deletions. □

LEMMA 12. *Under the assumptions of Theorem* 3, *the cost of the sequence of deletions is $O(\log n+k+l)$.*

Proof. For accounting purposes we shall divide the sequence of l deletions into disjoint *epochs*, with the first epoch starting immediately before the first deletion. Intuitively, epochs represent intervals during which insertions do not interact directly with deletions. We define the current epoch to end immediately *before* a deletion that references any node on the left path that has been changed by an insertion since the first deletion of the epoch. This deletion is then the first in the new epoch; the final epoch ends with the last deletion of the sequence. According to this definition, each epoch contains at least one deletion.

Let l_i denote the number of deletions during the ith epoch, k_i the number of insertions during this epoch, and m_i the tree size at the start of the epoch. The first deletion of epoch i costs $O(\log m_i)$. By Lemma 9, the final l_i-1 deletions cost

$O(l_i + \log m_i)$ since they operate on a section of the left path that is unaffected by insertions. Hence the total cost of the deletions in epoch i is $O(l_i + \log m_i)$. We shall prove that except for the first and last epochs, this cost is $O(l_i + k_{i-1})$, so that the total cost of these epochs is $O(l + k)$. Since $m_i \leqq n + k$, each of the first and last epochs costs $O(l_i + \log (n + k))$. Combining gives the bound in the lemma.

Consider an epoch i that is not the first or the last. The first deletion of an epoch transforms all nodes below height h on the left path into 3-nodes, where h is the height of some left-path node that has been changed by an insertion since the start of epoch $i - 1$. Let $h_i = \lfloor \lg \lg m_i \rfloor - 1$. By Lemma 10, the allowable insertions at this point cannot change the left path below height h_i. This remains true even if the tree size grows to m_i^2 or shrinks to $\sqrt{m_i}$, since this changes the value of lg lg m by only 1. Hence if $h \geqq h_i$ (i.e., all left-path nodes below height h are 3-nodes), Lemma 11 shows that $2^{h_i} = \Omega(\log m_i)$ deletions are necessary to reference a node above height h_i. Thus $l_i = \Omega(\log m_i)$, which means that $O(l_i + \log m_i)$, the cost of the epoch, is $O(l_i)$. If on the other hand $h < h_i$, this implies that at some point during epoch $i - 1$ the tree size m was much smaller than m_i, in particular $m < \sqrt{m_i}$. But this shows that $k_{i-1} = \Omega(m_i)$, so $O(l_i + \log m_i) = O(l_i + k_{i-1})$. In summary, we have shown that the cost of epoch i is $O(l_i + k_{i-1})$ regardless of the value of h. □

Combining the results of Lemmas 8 and 12 proves Theorem 3. □

Theorem 3 is certainly not the ultimate result of its kind. For example, it is possible to allow some number of insertions to fall close to the point of deletion and still preserve the time bound. (Note that Lemma 8 does not depend on any assumption about the distribution of insertions, so only the proof of the bound on deletions needs to be modified.) It may also be possible to prove a nontrivial bound when deletions are less highly constrained; for example, we might consider a "queue-like" access pattern in which insertions fall only in the right subtree of the root, and deletions are made only from the left subtree.

4. Level-linked trees. The results in §§ 1–3 show that in several interesting cases the $O(\log n)$ bound on individual insertions and deletions in a 2-3 tree is overly pessimistic. In order to use this information we must examine the cost of searching for the positions where the insertions and deletions are to take place. If the pattern of accesses is random, there is little hope of reducing the average search time below $O(\log n)$; it is impossible for any algorithm based solely on comparisons to beat $\Omega(\log n)$. But in many circumstances there is a known regularity in the reference pattern that we can exploit.

One possible method of using the correlation between accesses is to keep a *finger*—a pointer to an item in the list. For a suitable list representation it should be much more efficient to search for an item near the finger than one far away. Since the locale of interest may change with time, the list representation should make it easy to move a finger while still enjoying fast access near it. There may be more than one busy area in the list, so it should be possible to efficiently maintain multiple fingers.

The basic 2-3 tree structure for sorted lists shown in Fig. 3 is not suitable for finger searching, since there are items adjacent in the list whose only connection through the tree structure is a path of length $\theta(\log n)$. Figure 8 shows an extension of this structure that does support efficient access in the neighborhood of a finger. The arrangement of list elements and keys is unchanged, but the edges between internal nodes are made traversible upwards as well as downwards, and horizontal links are added between external nodes that are *neighbors* (adjacent on the same level). We shall call this list representation a *level-linked* 2-3 *tree*.

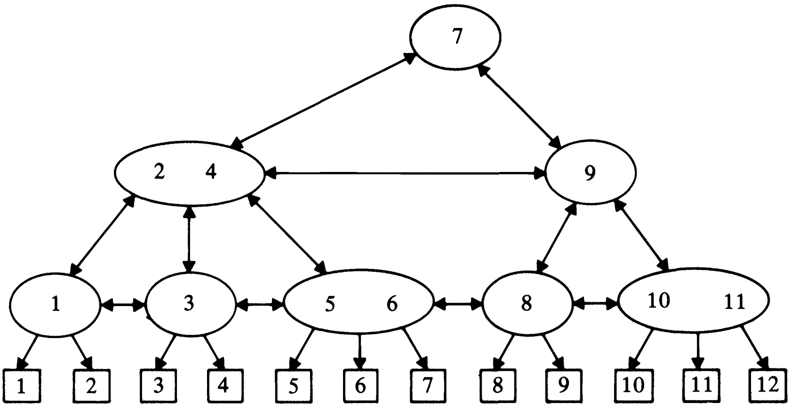


FIG. 8. *A level-linked* 2-3 *tree.*

A finger into this structure consists of a pointer to a terminal node of the tree. It would seem more natural for the finger to point directly to an external node, but no upward links leading away from the external nodes are provided in a level-linked tree; the reasons for this decision will become evident when implementation considerations are discussed in § 5. Note that the presence of a finger requires no change to the structure.

Roughly speaking, the search for a key k using a finger f proceeds by climbing the path from f toward the root of the tree. We stop ascending when we discover a node (or a pair of neighboring nodes) which subtends a range of the key space in which k lies. We then search downward for k using the standard search technique.

A more precise description of the entire search procedure is given below in an Algol-like notation. If t is an internal node, then we define LargestKey(t) and SmallestKey(t) to be the largest and smallest keys contained in t, and let LeftmostLink(t) and RightmostLink(t) denote respectively the leftmost and rightmost downward edges leaving t. The fields *lNbr*(t) and *rNbr*(t) give the left and right neighbors of t, and are Nil if no such nodes exist; *Parent*(t) is the parent of t, and is Nil if t is the root.

```
procedure FingerSearch ($f$, $k$)
    comment Here $f$ is a finger (a pointer to a terminal node) and $k$ is a key. If there is
    an external node with key $k$ in the structure fingered by $f$, then FingerSearch
    returns a pointer to the parent of the rightmost such node. Otherwise the
    procedure returns a pointer to a terminal node beneath which an external node
    with key $k$ may be inserted. Hence in either case the result may be used as a (new)
    finger.
    if      $k \geqq$ LargestKey($f$) then return SearchUpRight($f$, $k$)
    elseif  $k <$ SmallestKey($f$) then return SearchUpLeft($f$, $k$)
    else return $f$
    endif
end FingerSearch

procedure SearchUpRight($p$, $k$)
    loop
        comment At this point either $f = p$, or $f$ lies to the left of $p$'s right subtree.
        The key $k$ is larger than the leftmost (smallest) descendant of $p$.
        if      $k <$ LargestKey($p$) or $rNbr(p) =$ Nil then return SearchDown($p$, $k$)
        else    $q \leftarrow rNbr(p)$
```

```
                if      $k <$ SmallestKey($q$) **then return** SearchDownBetween($p, q, k$)
                elseif  $k <$ LargestKey($q$) **then return** SearchDown($q, k$)
                else    $p \leftarrow Parent(q)$
                endif
            endif
        repeat
    end SearchUpRight

    procedure SearchUpLeft($p, k$)
        {similar to the above}

    procedure SearchDownBetween($p, q, k$)
        loop until $p$ and $q$ are terminal:
            comment Here $p$ is the left neighbor of $q$, and $k$ is contained in the range of
            key values spanned by the children of $p$ and $q$.
            if      $k <$ LargestKey($p$) then return SearchDown($p, k$)
            elseif  $k \geqq$ SmallestKey($q$) then return SearchDown($q, k$)
            else    $p \leftarrow$ RightmostLink($p$)
                    $q \leftarrow$ LeftmostLink($q$)
            endif
        repeat
        if  $k <$ Key[RightmostLink($p$)]   then return $p$
                                            else return $q$
        endif
    end SearchDownBetween

    procedure SearchDown($p, k$)
        {the standard 2-3 tree search procedure} .
```

This algorithm allows very fast searching in the vicinity of fingers. In spite of this, we shall show that if a sequence of intermixed searches, insertions, and deletions is performed on a level-linked 2-3 tree, the cost of the insertions and deletions is dominated by the search cost, at least in the cases studied in §§ 1–3. In order to carry out this analysis we must first examine the cost of individual operations on a level-linked tree.

LEMMA 13. *If the key k is d keys away from a finger f, then* FingerSearch(f, k) *runs in* $\theta(\log d)$ *steps.*

Proof. The running time of *FingerSearch* is bounded by a constant times the height of the highest mode examined, since the search procedure examines at most four of the nodes at each level. It is not hard to see from the invariants in *SearchUpRight* (and *SearchUpLeft*) that in order for the search to ascend l levels in the tree, there must exist a subtree of size $l-2$ all of whose keys lie between k and the keys of the finger node. The lemma follows. □

LEMMA 14. *A new external node can be inserted in a given position in a level-linked* 2-3 *tree in* $\theta(1+s)$ *steps, where s is the number of node splittings caused by the insertion.*

Proof. We sketch an insertion method which can be implemented to run in the claimed time bound. Suppose we wish to insert a new external node with key k. During the insertion process we must update the links and the keys in the internal nodes. Let node p be the prospective parent of node e. If e would not be the rightmost child of p, we make e a child of p, insert the key k in node p and proceed with node-splitting as necessary. If e would be the rightmost child of p but e has a right neighbor, we make e a child of the right neighbor. Otherwise k is larger than all keys in the tree. In this case we make e a child of p and place the previously largest key in node p. (The key k is not used in an internal node until it is no longer the largest.)

When a 4-node q splits during insertion, it is easy to update the links in constant time. To maintain the internal key organization, we place the left and right keys of q in the new 2-nodes produced by the split, and the middle key in the parent of q. □

LEMMA 15. *An external node can be deleted from a level-linked* 2-3 *tree in* $\theta(1+f)$ *steps, where f is the number of node fusings.*

Proof. Similar to the proof of Lemma 14. □

LEMMA 16. *Creation or removal of a finger in a level-linked* 2-3 *tree requires* $\theta(1)$ *time.*

Proof. Obvious. □

Now we apply the results of §§ 1-3 to show that even though the search time in level-linked 2-3 trees can be greatly reduced by maintaining fingers, it still dominates the time for insertions and deletions in several interesting cases.

THEOREM 4. *Let L be a sorted list of size n represented as a level-linked* 2-3 *tree with one finger established. Then in any sequence of searches, finger creations, and k insertions, the total cost of the k insertions is* $O(\log n + \text{total cost of searches})$.

Proof. Let S be any sequence of searches, finger creations, and insertions which includes exactly k insertions. Let the external nodes of L after the insertions have been performed be named $0, 1, \cdots, n+k-1$ from left to right. Assign to each external node p a label $l(p)$, whose value is the number of external nodes lying strictly to the left of p which were present before the insertions took place; these labels lie in the range $0, 1, \cdots, n$.

Consider the searches in S which lead either to the creation of a new finger (or the movement of an old one) or to the insertion of a new item. Call an item of L *accessed* if it is either the source or the destination of such a search. (We regard an inserted item as the destination of the search which discovers where to insert it.) Let $p_1 < p_2 < \cdots < p_l$ be the accessed items.

We shall consider graphs whose vertex set is a subset of $\{p_i | 1 \leqq i \leqq l\}$. We denote an edge joining $p_i < p_j$ in such a graph by $p_i - p_j$ and we define the *cost* of this edge to be $\max(\lceil \lg (l(p_j) - l(p_i) + 1)\rceil, 1)$. For each item p_i (except the initially fingered item) let q_i be the fingered item from which the search to p_i was made. Each q_i is also in $\{p_i | 1 \leqq i \leqq l\}$ since each finger except the first must be established by a search. Consider the graph G with vertex set $\{p_i | 1 \leqq i \leqq l\}$ and edge set $\{(q_i, p_i) | 1 \leqq i \leqq l$ and p_i is not the originally fingered item$\}$.

Some constant times the sum of edge costs in G is a lower bound on the total search cost, since $|l(p_i) - l(q_i)| + 1$ can only underestimate the actual distance between q_i and p_i when p_i is accessed. We shall describe a way to modify t, while never increasing its cost, until it becomes

$$r_1 - r_2 - \cdots - r_k,$$

where $r_1 < r_2 < \cdots < r_k$ are the k inserted items. Since the cost of this graph is $\sum_{1 < i \leqq k} \lceil \lg (r_i - r_{i-1} + 1)\rceil$, the theorem then follows from Theorem 1.

The initial graph G is connected, since every accessed item must be reached from the initially fingered item. We first delete all but $l-1$ edges from G so as to leave a spanning tree; this only decreases the cost of G.

Next, we repeat the following step until it is no longer applicable: let $p_i - p_j$ be an edge of G such that there is an accessed item p_k satisfying $p_i < p_k < p_j$. Removing edge $p_i - p_j$ now divides G into exactly two connected components. If p_k is in the same connected component as p_i, we replace $p_i - p_j$ by $p_k - p_j$; otherwise, we replace $p_i - p_j$ by $p_i - p_k$. The new graph is still a tree spanning $\{p_i | 1 \leqq i \leqq l\}$ and the cost has not increased.

Finally, we eliminate each item p_j which is not an inserted item by transforming $p_i-p_j-p_k$ to p_i-p_k, and by removing edges p_j-p_k where there is no other edge incident to p_j. This does not increase cost, and it results in the tree of inserted items

$$r_1-r_2-\cdots-r_k$$

as desired. □

THEOREM 5. *Let L be a sorted list of size n represented as a level-linked* 2-3 *tree with one finger established. Then in any sequence of searches, finger creations, and k deletions, the cost of the deletions is* $O(\log n + \text{total cost of searches})$.

Proof. Similar to the proof of Theorem 4, using Theorem 2. □

THEOREM 6. *Let L be a sorted list of size n represented as a level-linked* 2-3 *tree with one finger established. For any sequence of searches, finger creations, k insertions, and l deletions, the total cost of the insertions and deletions is* $O(\log n + \text{total cost of searches})$ *if the insertions and deletions satisfy the assumptions of Theorem* 3.

Proof. Similar to the proof of Theorem 4, using Theorem 3. □

5. Implementation and applications. In § 4 we described a level-linked 2-3 tree in terms of internal and external nodes. The external nodes contain the items stored in the list, while the internal nodes are a form of "glue" which binds the items together. The problem remains of how to represent these objects in storage.

External nodes present no difficulty: they can be represented by the items themselves, since we only maintain links going *to* these nodes (and none coming *from* them). Internal nodes may be represented in an obvious way by a suitable record structure containing space for up to two keys and three downward links, a tag to distinguish between 2- and 3-nodes, and other fields. One drawback of this approach is that because the number of internal nodes is unpredictable, the insertion and deletion routines must allocate and deallocate nodes. 'In random 2-3 trees [9] the ratio of 2-nodes to 3-nodes is about 2 to 1, so we waste storage by leaving room for two keys in each node. Having different record structures for the two node types might save storage at the expense of making storage management much more complicated.

Figure 9 shows a representation which avoids these problems. A 3-node is represented in a linked fashion, analogous to the binary tree structure for 2-3 trees [6, p. 469]. The internal node component containing a key k is combined as a single record with the representation of the item (external node) with key k. Hence storage is allocated and deallocated only when items are created and destroyed, and storage is saved because the keys in the internal nodes are not represented explicitly. (The idea of combining the representations of internal and external nodes is also found in the "loser-oriented" tree for replacement selection [6, p. 256].)

An example which illustrates this representation is shown in Fig. 10. Each external node except the largest participates in representing an internal node, so it is convenient to assume the presence of an external node with key $+\infty$ in the list. This node need not be represented explicitly, but can be given by a null pointer as in the figure. Null *rLink*s are also used to distinguish a 3-node from a pair of neighboring 2-nodes. There are several ways to identify the *lLink*s and *rLink*s that point to external nodes: one is to keep track of height in the tree during FingerSearch, since all external nodes lie on the same level. Another method is to note that a node p is terminal if and only if $lLink(p)=p$.

We now consider the potential applications of this list representation. One application is in sorting files which have a bounded number of inversions. The result proved by Guibas et al. [4], that insertion sort using a list representation with one finger

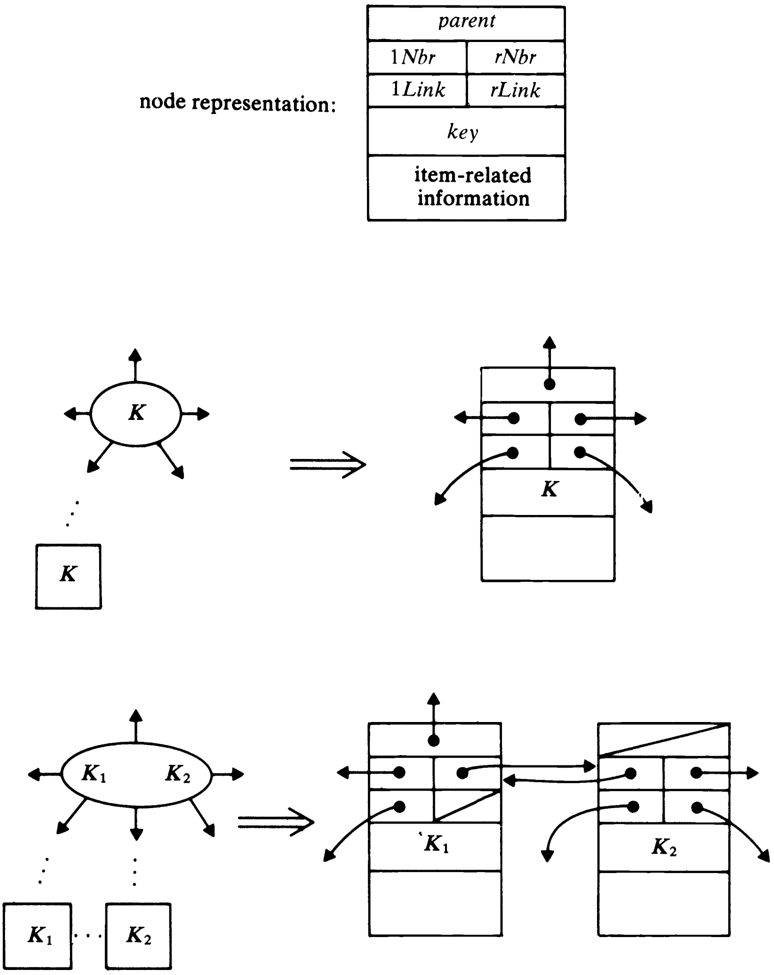


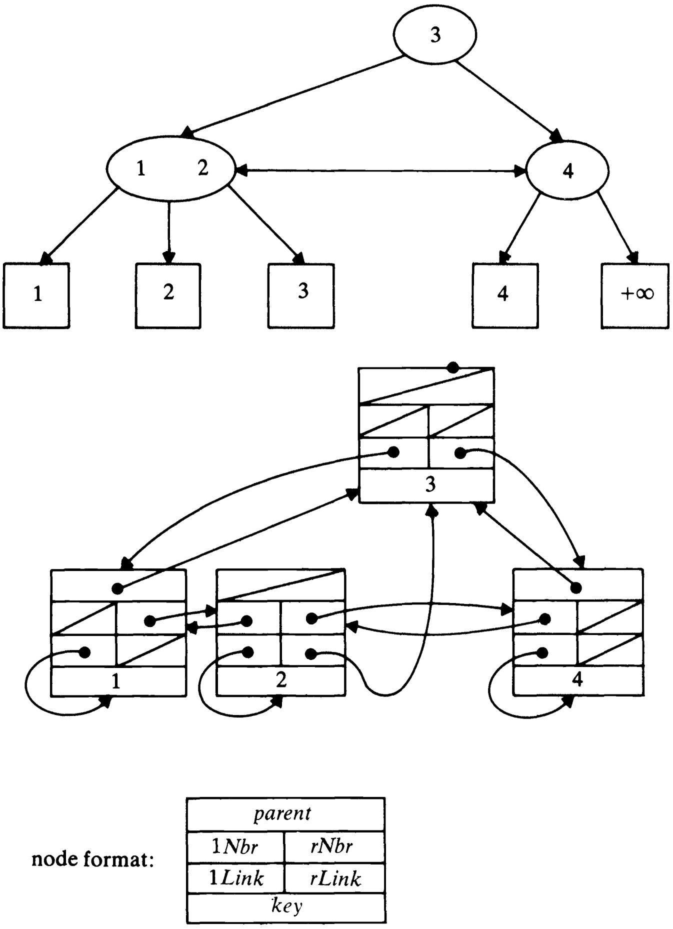
FIG. 9. *A storage representation for internal and external nodes.*

gives asymptotically optimal results, applies equally to our structure since insertion sort does not require deletions.

A second application is in merging: given sorted lists of lengths m and n, with $m \leqq n$, we wish to merge them into a single sorted list. Any comparison-based algorithm for this problem must use at least

$$\left\lceil \lg \binom{m+n}{m} \right\rceil = \theta\left(m \log \frac{n}{m}\right)$$

comparisons; we would like an algorithm whose running time has this magnitude. We solve this problem using our list structure by inserting the items from the smaller list in increasing order into the larger list, keeping the finger positioned at the most recently inserted item. This process requires $O(m)$ steps to dismantle the smaller list, and $O(\log n + \sum_{1<i\leqq m} \log d_i)$ steps for the insertions, where d_i is the distance from the finger to the ith insertion. Since the items are inserted in increasing order, the finger moves from left to right through the larger list, and thus $\sum_{1<i\leqq m} d_i \leqq n$. To maximize $\sum_{1<i\leqq m} \log d_i$ subject to this constraint we choose the d_i to be equal, and this gives the



node format:

parent	
1Nbr	rNbr
1Link	rLink
key	

FIG. 10. *A structure and its storage representation.*

desired bound of $O(m \log (n/m))$ steps for the algorithm. (The usual height-balanced or 2-3 trees can be used to perform fast-merging [3], but the algorithm is not obvious and the time bound requires an involved proof.)

When an ordered set is represented as a sorted list, the merging algorithm just described can be modified to perform the set union operation: we simply check for, and discard, duplicates when inserting items from the smaller list into the larger list. This obviously gives an $O(m \log (n/m))$ algorithm for set intersection as well, if we retain the duplicates rather than discarding them. Trabb–Pardo [8] has developed algorithms based on trie structures which also solve the set intersection problem (and the union or merging problems) in $O(m \log (n/m))$ time, but only on the average.

Another application for the level-linked 2-3 tree is in implementing a priority queue used as a simulation event list. In this situation the items being stored in the list are procedures to be executed at a known instant of simulated "time"; to perform one simulation step we delete the item from the list having the smallest time and then execute it, which may cause new events to be inserted into the list. Theorem 3 shows that unless these new events are often very soon to be deleted, a 2-3 tree can process a long sequence of such simulation steps with only a constant cost per operation (independent of the queue size). Furthermore, searches using fingers will usually be very efficient since the simulation program produces events according to known patterns. (Some simulation languages already give programmers access to crude "fingers", by allowing the search to begin from a specified end of the event list.)

Another interesting application is to text editing. A text can be represented as a list of component strings that, when concatenated, form the entire text string. If the list of components is represented as a level-linked 2-3 tree, then insertion, deletion, and searching in the text can all be performed efficiently. There is a great deal of reference locality in text editing, so the average search will be very fast while the worst case is only logarithmic in the number of components. Thus the proper use of this data structure might significantly improve the responsiveness of an interactive text editor.

An obvious question relating to our structure is whether it can be generalized so that arbitrary deletions will not change the worst-case time bound for a sequence of accesses. This seems to be difficult, since the requirement for a movable finger conflicts with the need to maintain path regularity constraints [4]. Thus a compromise between the unconstrained structure given here and the highly constrained structure of Guibas et al. [4] should be explored.

Even if such a more general structure could be found, it might be less practical than ours. To put the problem of deletions in perspective, it would be interesting to derive bounds on the average case performance of our structure under insertions and deletions, using a suitable model of random insertions and deletions. It may be possible, even without detailed knowledge of random 2-3 trees, to show that operations which require $\theta(\log n)$ time are very unlikely.

REFERENCES

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

[2] RUDOLF BAYER AND EDWARD M. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Informatica, 1 (1972), pp. 173–189.

[3] MARK R. BROWN AND ROBERT E. TARJAN, *A fast merging algorithm*, J. Assoc. Comput. Mach. 26 (1979), pp. 211–226.

[4] LEO J. GUIBAS, EDWARD M. MCCREIGHT, MICHAEL F. PLASS AND JANET R. ROBERTS, *A new representation for linear lists*, Proc. Ninth Annual ACM Symposium on Theory of Computing, Boulder, CO, 1977, pp. 49–60.

[5] DONALD E. KNUTH, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, 2nd edition, Addison-Wesley, Reading, MA, 1975.

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

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

[8] LUIS TRABB-PARDO, *Set Representation and Set Intersection*, Stanford Computer Science Dept. Rep. STAN-CS-78-681, Stanford Univ., Stanford, CA, December 1978.

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

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

0097-5397/80/0903-0013 $01.00/0

APPLICATIONS OF A PLANAR SEPARATOR THEOREM*

RICHARD J. LIPTON† AND ROBERT ENDRE TARJAN‡

Abstract. Any n-vertex planar graph has the property that it can be divided into components of roughly equal size by removing only $O(\sqrt{n})$ vertices. This separator theorem, in combination with a divide-and-conquer strategy, leads to many new complexity results for planar graph problems. This paper describes some of these results.

Key words. algorithm, Boolean circuit complexity, divide-and-conquer, graph embedding, lower bounds, matching, maximum independent set, nonserial dynamic programming, pebbling, planar graphs, separator, space-time tradeoffs

1. Introduction. One efficient approach to solving computational problems is "divide-and-conquer" [1]. In this method, the original problem is divided into two or more smaller problems. The subproblems are solved by applying the method recursively, and the solutions to the subproblems are combined to give the solution to the original problem. Divide-and-conquer is especially efficient when the subproblems are substantially smaller than the original problem. In this paper we explore the efficient application of divide-and-conquer to a variety of problems on planar graphs. We employ the following theorem.

THEOREM 1 [20]. *Let G be any n-vertex planar graph with nonnegative vertex costs summing to no more than one. Then the vertices of G can be partitioned into three sets A, B, C, such that no edge joins a vertex in A with a vertex in B, neither A nor B has total vertex cost exceeding $\frac{2}{3}$, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices. Furthermore A, B, C can be found in $O(n)$ time.*

In the special case of equal-cost vertices, this theorem becomes

COROLLARY 1. *Let G be any n-vertex planar graph. The vertices of G can be partitioned into three sets A, B, C, such that no edge joins a vertex in A with a vertex in B, neither A nor B contains more than $2n/3$ vertices, and C contains no more than $2\sqrt{2}\sqrt{n}$ vertices.*

Corollary 1 verifies a conjecture of Ungar [32], who obtained a similar result but with a bound of $O(\sqrt{n}\log n)$ on the size of C. It is easy to construct examples to show that Corollary 1 is tight to within a constant factor in the worst case [20].

Each section of this paper describes a different use of Theorem 1. The results range from an efficient algorithm for finding maximum independent sets in planar graphs to lower bounds on the complexity of planar Boolean circuits. In each case, the only property of planar graphs that we use is Theorem 1, and our results generalize easily to any class of graphs which can be separated into small components by removing a small number of vertices. For instance, by employing the following result of Sider, we can extend our results to graphs of arbitrary genus.

LEMMA 1 [2], [30]. *Let G be any n-vertex graph of genus $g>0$. Then there exists a subset of no more than $\sqrt{2n}$ vertices whose removal reduces the genus of G by at least one.*

* Received by the editors August 9, 1978, and in final revised form September 11, 1979.

† Computer Science Department, Yale University, New Haven, Connecticut. Now at Computer Science Division, University of California, Berkeley, California 94720. This research was partially supported by the U.S. Army Research Office, Grant No. DAAG 29-76-G-0338.

‡ Computer Science Department, Stanford University, Stanford, California 94305. This research was partially supported by National Science Foundation Grant MCS-75-22870 and by the Office of Naval Research Contract N00014-76-C-0688.

THEOREM 2. *If G is an n-vertex graph of genus $g > 0$, there is a subset of no more than $g\sqrt{2n}$ vertices whose removal leaves a planar graph.*

Proof. The proof is by induction on g employing Lemma 1. □

We state our results only for the planar case, since it seems the most interesting, and leave as an exercise the extension of these results to graphs of higher genus.

2. Approximation algorithms for *NP*-complete problems. Divide-and-conquer in combination with Theorem 1 can be used to rapidly find good approximate solutions to certain *NP*-complete problems on planar graphs. As an example we consider the *maximum independent set* problem, which asks for a maximum number of pairwise non-adjacent vertices in a planar graph. We need the following generalization of Theorem 1.

THEOREM 3. *Let G be an n-vertex planar graph with nonnegative vertex costs summing to no more than one and let $0 \leqq \varepsilon \leqq 1$. Then there is some set C of $O(\sqrt{n/\varepsilon})$ vertices whose removal leaves G with no connected component of cost exceeding ε. Furthermore the set C can be found in $O(n \log n)$ time.*

Proof. If $\varepsilon \leqq 1/\sqrt{n}$, let C contain all the vertices of G. Then the theorem holds. Otherwise, apply the following algorithm to G.

Initialization. Let $C = \varnothing$.

General Step. Find some connected component K in G minus C with cost exceeding ε. Apply Theorem 1 to K, producing a partition A_1, B_1, C_1 of its vertices. Let $C = C \cup C_1$.

Repeat the general step until G minus C has no component with cost exceeding ε.

The effect of one execution of the general step is to divide the component K into smaller components, each with no more than two-thirds the cost of K. Consider all components that arise during the course of the algorithm. Assign a *level* to each component as follows. If the component exists when the algorithm halts, the component has level zero. Otherwise the level of the component is one greater than the maximum level of the components formed when it is split by the general step. With this definition, any two components on the same level are vertex-disjoint.

Each level one component has cost greater than ε, since it is eventually split by the general step. Thus, for $i \geqq 1$, each level i component has cost at least $(\frac{3}{2})^{i-1}\varepsilon$. Since the total cost of G is at most one, the total number of components of level i is at most $(\frac{2}{3})^{i-1}/\varepsilon$. In particular, the maximum level k must satisfy $1 \leqq (\frac{2}{3})^{k-1}/\varepsilon \leqq (\frac{2}{3})^{k-1}\sqrt{n}$, which means $k \leqq (\log_{3/2} n)/2 + 1$. Since the time to split a component is linear in its number of vertices, and since any two components on the same level are vertex-disjoint, the total running time of the algorithm is $O(n \log n)$.

It remains for us to bound the size of the set C produced by the algorithm. Let $K_1, K_2, \ldots, K_l$, of sizes $n_1, n_2, \ldots, n_l$, respectively, be the components of some level $i \geqq 1$. The number of vertices added to C by splitting $K_1, K_2, \cdots, K_l$ is bounded by $2\sqrt{2}\sum_{j=1}^{l}\sqrt{n_j}$. We have $l \leqq (\frac{2}{3})^{i-1}/\varepsilon$ and $\sum_{j=1}^{l} n_j \leqq n$. For fixed l, the sum $\sum_{j=1}^{l}\sqrt{n_j}$ subject to $\sum_{j=1}^{l} n_j \leqq n$ is maximized by setting $n_j = n/l$ for $1 \leqq j \leqq l$; thus $2\sqrt{2}\sum_{j=1}^{l}\sqrt{n_j} \leqq 2\sqrt{2}\sqrt{nl} \leqq 2\sqrt{2}\sqrt{n/\varepsilon}(\frac{2}{3})^{(i-1)/2}$. It follows that $|C| \leqq \sum_{i=1}^{\infty} 2\sqrt{2}\sqrt{n/\varepsilon}(\frac{2}{3})^{(i-1)/2} = O(\sqrt{n/\varepsilon})$. □

The following algorithm uses Theorem 3 to find an approximately maximum independent set I in a planar graph $G = (V, E)$. The algorithm uses a function $k(n)$ to be chosen later.

Step 1. Apply Theorem 3 to G with $\varepsilon = k(n)/n$ and each vertex having cost $1/n$ to find a set of vertices C of size $O(n/\sqrt{k(n)})$ whose removal leaves no connected component with more than $k(n)$ vertices.

Step 2. In each connected component of G minus C, find a maximum independent set by checking every subset of vertices for independence. Form I as a union of maximum independent sets, one from each component.

Let I^* be a maximum independent set of G. The restriction of I^* to one of the connected components formed when C is removed from G can be no larger than the restriction of I to the same component. Thus $|I^*|-|I|=O(n/\sqrt{k(n)})$. Since G is planar, G is four-colorable, and $|I^*|\geqq n/4$. Thus $(|I^*|-|I|)/|I^*|=O(1/\sqrt{k(n)})$, and the relative error in the size of I tends to zero with increasing n as long as $k(n)$ tends to infinity with increasing n.

Step 1 of the algorithm requires $O(n \log n)$ time by Theorem 2. Step 2 requires $O(n_i\, 2^{n_i})$ time on a connected component of n_i vertices. The total time required by Step 2 is thus

$$O\left(\max\left\{\sum_{i-1}^{n} n_i 2^{n_i} \,\middle|\, \sum_{i=1}^{n} n_i = n \text{ and } O\leqq n_i \leqq k(n)\right\}\right)=O\left(\frac{n}{k(n)}k(n)2^{k(n)}\right)=O(n2^{k(n)}).$$

Hence the entire algorithm requires $O(n \cdot \max\{\log n, 2^{k(n)}\})$ time. If we shoose $k(n)=\log n$, we get an $O(n^2)$-time algorithm with $O(1/\sqrt{\log n})$ relative error. If we choose $k(n)=\log\log n$, we get an $O(n \log n)$ algorithm with $O(1/\sqrt{\log\log n})$ relative error.

3. Nonserial dynamic programming. Many *NP*-complete problems, such as the maximum independent set problem, the graph coloring problem, and others, can be formulated as *nonserial dynamic programming problems* [3], [27]. Such a problem is of the following form: maximize the objective function $f(x_1,\cdots,x_n)$, where f is given as a sum of terms $f_k(\cdot)$, each of which is a function of only a subset of the variables. We shall assume that all variables x_i take on values from the same finite set S, and that the values of the terms $f_k(\cdot)$ are given by tables. Associated with such an objective function f is an interaction graph $G=(V,E)$, containing one vertex v_i for each variable x_i in f, and an edge joining x_i and x_j for any two variables x_i and x_j which appear in a common term $f_k(\cdot)$.

We can formulate the maximum independent set problem as a nonserial dynamic programming problem as follows. Let $G=(V,E)$ be an undirected graph. For each vertex v_i, $1\leqq i\leqq n$, let x_i be an associated variable which can assume a value of either zero or one. Let the objective function $f(x_1,x_2,\cdots,x_n)$ be defined by

$$f(x_1,x_2,\cdots,x_n)=\sum_{(v_i,u_j)\in E} f_e(x_i,x_j)+\sum_{i=1}^{n} x_i,$$

where $f_e(x_i,x_j)=-\infty$ if $x_i=x_j=1$, $f_e(x_i,x_j)=0$ otherwise. Then a maximum independent set in G corresponds to an assignment of values 0, 1 to $x_1,x_2,\cdots,x_n$ which maximizes f; $x_i=1$ means x_i is in the independent set, $x_i=0$ means x_i is not in the independent set. Other graph problems can be formulated similarly. Note that G is the interaction graph of f.

By trying all possible values of the variables, a nonserial dynamic programming problem can be solved in $2^{O(n)}$ time. We shall show that if the interaction graph of the problem is planar, the problem can be solved in $2^{O(\sqrt{n})}$ time. This means that substantial savings are possible when solving typical *NP*-complete problems restricted to planar graphs. Note that if the interaction graph of f is planar, no term $f_k(\cdot)$ of f can contain more than four variables, since the complete graph on five vertices is not planar.

In order to describe the algorithm, we need one additional concept. The *restriction* of an objective function $f=\sum_{k=1}^{m} f_k$ to a set of variables $x_{i_1},\cdots,x_{i_j}$ is the objective function $f'=\sum\{f_k|f_k$ depends only upon $x_{i_1},\cdots,x_{i_j}\}$.

Given an objective function $f(x_1, \cdots, x_n) = \sum_{k=1}^{m} f_k$ and a subset S of the variables $x_1, \cdots, x_n$ which are constrained to have specific values, the following algorithm solves the problem: maximize f subject to the constraints on the variables in S. In the presentation, we do not distinguish between the variables $x_1, \cdots, x_n$ and the corresponding vertices in the interaction graph.

Step 1. If $n < 100$, solve the problem by exhaustively trying all possible assignments to the unconstrained variables. Otherwise, go to Step 2.

Step 2. Apply Corollary 1 to the interaction graph G of f. Let A, B, C be the resulting vertex partition. Let f_1 be the restriction of f to $A \cup C$ and let f_2 be the restriction of f to $B \cup C$. For each possible assignment of values to the variables in $C - S$, perform the following steps:

(a) Maximize f_1 with the given values for the variables in $C \cup S$ by applying the method recursively;

(b) maximize f_2 with the given values for the variables in $C \cup S$ by applying the method recursively;

(c) combine the solutions to (a) and (b) to obtain a maximum value of f with the given values for the variables in $C \cup S$.

Choose the assignment of values to variables in $C \cup S$ which maximizes f and return the appropriate value of f as the solution.

The correctness of this algorithm is obvious. If $n \geqq 100$, the algorithm solves at most $2^{O(\sqrt{n})}$ subproblems in Step 2, since C is of $O(\sqrt{n})$ size. Each subproblem contains at most $2n/3 + 2\sqrt{2}\sqrt{n} \leqq 29n/30$ variables. Thus if $t(n)$ is the running time of the algorithm, we have $t(n) \leqq O(n) + 2^{O(\sqrt{n})} \cdot t(29n/30)$ if $n \geqq 100$, $t(n) = O(1)$ if $n < 100$. An inductive proof shows that $t(n) \leqq 2^{O(\sqrt{n})}$.

4. Pebbling.

The following one-person game arises in register allocation problems [28], the conversion of recursion to iteration [23], and the study of time-space tradeoffs [4], [12], [25]. Let $G = (V, E)$ be a directed acyclic graph with maximum in-degree k. If (v, w) is an edge of G, v is a *predecessor* of w and w is a *successor* of v. The game involves placing pebbles on the vertices of G according to certain rules. A given step of the game consists of either placing a pebble on an empty vertex of G (called *pebbling* the vertex) or removing a pebble from a previously pebbled vertex. A vertex may be pebbled *only* if all its predecessors have pebbles. The object of the game is to successively pebble each vertex of G (in any order) subject to the constraint that at most a given number of pebbles are ever on the graph simultaneously.

It is easy to pebble any vertex of an n-vertex graph in n steps using n pebbles. We are interested in pebbling methods which use fewer than n pebbles but possibly many more than n steps. It is known that any vertex of an n-vertex graph can be pebbled with $O(n/\log n)$ pebbles [12] (where the constant depends upon the maximum in-degree), and that in general no better bound is possible [25]. We shall show that if the graph is planar, only $O(\sqrt{n})$ pebbles are necessary, generalizing a result of [25]. An example of Cook [4] shows that no better bound is possible for planar graphs.

THEOREM 4. *Any n-vertex planar acyclic directed graph with maximum in-degree k can be pebbled using $O(\sqrt{n} + k \log_2 n)$ pebbles.*

Proof. Let $\alpha = 2\sqrt{2}$ and $\beta = \frac{2}{3}$. Let G be the graph to be pebbled. Use the following recursive pebbling procedure. If $n = 1$, pebble the single vertex of G. If $n > 1$, find a vertex partition A, B, C satisfying Corollary 1. Pebble the vertices of G in topological order.[1] To pebble a vertex v, delete all pebbles except those on C. For each predecessor

[1] That is, an order such that if v is a predecessor of w, v is pebbled before w.

u of v, let $G(u)$ be the subgraph of G induced by the set of vertices with pebble-free paths to u. Apply the method recursively to each $G(u)$ to pebble all predecessors of v, leaving a pebble on each such predecessor. Then pebble v.

If $p(n)$ is the maximum number of pebbles required by this method on any n-vertex graph, then

$$p(1) = 1,$$
$$p(n) \leqq \alpha\sqrt{n} + k + p(\lfloor 2n/3 \rfloor) \quad \text{if } n > 1.$$

An inductive proof shows that $p(n)$ is $O(\sqrt{n} + k \log_2 n)$. □

It is also possible to obtain a substantial reduction in pebbles while preserving a polynomial bound on the number of pebbling steps, as the following theorem shows.

THEOREM 5. *Any n-vertex planar acyclic directed graph with maximum in-degree k can be pebbled using $O(n^{2/3} + k)$ pebbles in $O(n^{5/3})$ time.*

Proof. Let C be a set of $O(n^{2/3})$ vertices whose removal leaves G with no weakly connected component[2] containing more than $n^{2/3}$ vertices. Such a set C exists by Theorem 2. The following pebbling procedure places pebbles permanently on the vertices of C. Pebble the vertices of G in topological order. To pebble a vertex v, pebble each predecessor u of v and then pebble v. To pebble a predessor u, delete all pebbles from G except those on vertices in C or on predecessors of v. Find the weakly connected component in G minus C containing u. Pebble all vertices in this component, in topological order.

The total number of pebbles required by this strategy is $O(n^{2/3})$ to pebble vertices in C plus $n^{2/3}$ to pebble each weakly connected component plus k to pebble predecessors of the vertex v to be pebbled. We can bound the number of pebbling steps as follows. To pebble a vertex v requires $d_I(v)n^{2/3} + 1$ steps, where $d_I(v)$ is the in-degree of vertex v. The total pebbling time is thus $n + \sum_{v \in V} d_I(v)n^{2/3} \leqq n + (3n-3)n^{2/3} = O(n^{5/3})$. □

5. Lower bounds on Boolean circuit size. A *Boolean circuit* is an acyclic directed graph such that each vertex has in-degree zero or two, the predecessors of each vertex are ordered, and corresponding to each vertex v of in-degree two is a binary Boolean operation b_v. With each vertex of the circuit we associate a Boolean function which the vertex computes, defined as follows. With each of the k vertices v_i of in-degree zero (inputs) we associate a variable x_i and an identity function $f_{v_i}(x_i) = x_i$. With each vertex w of in-degree two having predecessors u, v we associate the function $f_w = b_w(f_u, f_v)$. The circuit computes the set of functions associated with its vertices of out-degree zero (outputs).

We are interested in obtaining lower bounds on the size (number of vertices) of Boolean circuits which compute certain common and important functions. Using Theorem 1 we can obtain such lower bounds under the assumption that the circuits are planar. Any circuit can be converted into a planar circuit by the following steps. First, embed the circuit in the plane, allowing edges to cross if necessary. Next, replace each pair of crossing edges by the crossover circuit illustrated in Figure 1. It follows that any lower bound on the size of planar circuits is also a lower bound on the total number of vertices and edge crossings in any planar representation of a nonplanar circuit. In a technology for which the total number of vertices and edge crossings is a reasonable measure of cost, our lower bounds imply that it may be expensive to realize certain commonly used functions in hardware.

[2] A *weakly connected component* of a directed graph is a connected component of the undirected graph formed by ignoring edge directions.

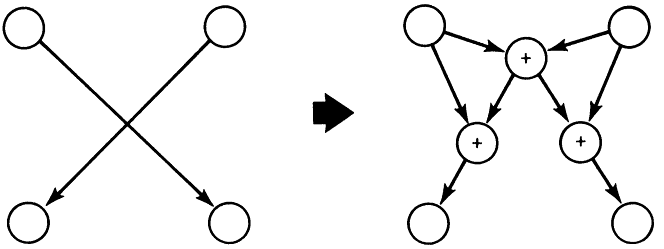


FIG. 1. *Elimination of a crossover by use of three "exclusive or" gates. Reference* [11] *contains a crossover circuit which uses only "and" and "not".*

A *superconcentrator* is an acyclic directed graph with m inputs and m outputs such that any set of k inputs and any set of k outputs are joined by k vertex-disjoint paths, for all k in the range $1 \leqq k \leqq m$.

THEOREM 6. *Any m-input, m-output planar superconcentrator contains at least* $m^2/72$ *vertices.*

Proof. Let G be an m-input, m-output planar superconcentrator. Assign to each input and output of G a cost of $1/(2m)$, and to every other vertex a cost of zero. Let A, B, C be a vertex partition satisfying Theorem 1 on G (ignoring edge directions). Suppose C contains p inputs and outputs. Without loss of generality, suppose that A is no more costly than B, and that A contains no more outputs than inputs. A contains between $2m/3-p$ and $m-p/2$ inputs and outputs. Hence A contains at least $m/3-p/2$ inputs and at most $m/2-p/4$ outputs. B contains at least $m-p-(m/2-p/4)=m/2-3p/4$ outputs. Let $k=\min\{\lceil m/3-p/2\rceil, \lceil m/2-3p/4\rceil\}$. Since G is a superconcentrator, any set of k inputs in A and any set of k outputs in B are joined by k vertex-disjoint paths. Each such path must contain a vertex in C which is neither an input nor an output. Thus $2\sqrt{2}\sqrt{n}-p \geqq \min\{m/3-p/2,\ m/2-3p/4\} \geqq m/3-p$, and $n \geqq m^2/72$. □

The property of being a superconcentrator is a little too strong to be useful in deriving lower bounds on the complexity of interesting functions. However, there are weaker properties which still require $\Omega(m^2)$ vertices. Let $G=(V, E)$ be an acyclic directed graph with m numbered inputs $v_1, v_2, \cdots, v_m$ and m numbered outputs $w_1, w_2, \cdots, w_m$. G is said to have the *shifting property* if, for any k in the range $1 \leqq k \leqq m$, any l in the range $0 \leqq l \leqq m-k$, and any subset of k sources $\{v_{i_1}, \cdots, v_{i_k}\}$ such that $i_1, i_2, \cdots, i_k \leqq m-l$, there are k vertex-disjoint paths joining the set of inputs $\{v_{i_1}, \cdots, v_{i_k}\}$ with the set of outputs $\{w_{i_1+l}, \cdots, w_{i_k+l}\}$.

THEOREM 7. *Let G be a planar acyclic directed graph with the shifting property. Then G contains at least* $\lfloor m/2 \rfloor^2/162$ *vertices.*

Proof. Suppose that G contains n vertices. Assign a cost of $1/m$ to each of the first $\lfloor m/2 \rfloor$ inputs and to each of the last $\lfloor m/2 \rfloor$ outputs of G, and a cost of zero to every other vertex of G. Call the first $\lfloor m/2 \rfloor$ inputs and the last $\lfloor m/2 \rfloor$ outputs of G *costly*. Let A, B, C be a vertex partition satisfying Theorem 1 on G (ignoring edge directions).

Without loss of generality, suppose that A is no more costly than B, and that A contains no more costly outputs than costly inputs. Let A' be the set of costly inputs in A, B' the set of costly outputs in B, p the number of costly inputs and outputs in C, and q the number of costly inputs and outputs in A. Then $2\lfloor m/2 \rfloor/3-p \leqq q \leqq \lfloor m/2 \rfloor - p/2$. Hence $|A'| \geqq q/2 \geqq \lfloor m/2 \rfloor/3-p/2$. Also

$$\begin{aligned}|A'| \cdot |B'| &\geqq |A'| \cdot (\lfloor m/2 \rfloor - p - (q-|A'|)) \\ &\geqq q/2 \cdot (\lfloor m/2 \rfloor - p - q/2).\end{aligned}$$

The function $x(\lfloor m/2\rfloor-p-x)$ for $\lfloor m/2\rfloor/3-p/2\leqq x\leqq\lfloor m/2\rfloor/2-p/4$ is minimized either at $x=\lfloor m/2\rfloor/3-p/2$ or at $x=\lfloor m/2\rfloor/2-p/4$. If $x=\lfloor m/2\rfloor/3-p/2$, we have $x(\lfloor m/2\rfloor-p-x)=2\lfloor m/2\rfloor^2/9-p\lfloor m/2\rfloor/2+p^2/4$. If $x=\lfloor m/2\rfloor/2-p/4$, we have $x(\lfloor m/2\rfloor-p-x)=\lfloor m/2\rfloor^2/4-p\lfloor m/2\rfloor/2+3p^2/16$. It follows that $|A'|\cdot|B'|\geqq 2\lfloor m/2\rfloor^2/9-p\lfloor m/2\rfloor/2$.

For $v_i\in A'$, $w_j\in B'$, and l in the range $1\leqq l\leqq\lfloor m/2\rfloor$, call v_i, w_j, l a *match* if $j-i=l$. For every $v_i\in A'$ and $w_j\in B'$ there is exactly one value of l which produces a match; hence the total number of matches for all possible v_i, w_j, l is $|A'|\cdot|B'|\geqq 2\lfloor m/2\rfloor^2/9-p\lfloor m/2\rfloor/2$. Since there are only $\lfloor m/2\rfloor$ values of l, some value of l produces at least $2\lfloor m/2\rfloor/9-p/2$ matches. Thus, for $k=2\lfloor m/2\rfloor/9-p/2$, there is some value of l and some set of k inputs $A''=\{v_{i_1},v_{i_2},\cdots,v_{i_k}\}\subseteq A'$ such that $B''=\{w_{i_1+l},w_{i_2+l},\cdots,w_{i_k+l}\}\subseteq B'$. Since G has the shifting property, there must be k vertex-disjoint paths between A'' and B''. But each such path must contain a vertex of C which is neither an input nor an output. Hence $2\sqrt{2}\sqrt{n}-p\geqq 2\lfloor m/2\rfloor/9-p/2$, and $n\geqq\lfloor m/2\rfloor^2/162$. □

A *shifting circuit* is a Boolean circuit with *m primary inputs* $x_1,x_2,\cdots,x_m$, zero or more *auxiliary inputs*, and m outputs $z_1,z_2,\cdots,z_m$, such that, for any k in the range $0\leqq k\leqq m$, there is some assignment of the constants 0, 1 to the auxiliary inputs so that output z_{i+k} computes the identity function x_i, for $0\leqq i\leqq m-k$. The *Boolean convolution* of two Boolean vectors $(x_1,x_2,\cdots,x_m)$ and $(y_1,y_2,\cdots,y_m)$ is the vector $(z_2,z_3,\cdots,z_{2m})$ given by $z_k=\sum_{i+j=k}x_iy_j$.

COROLLARY 2. *Any planar shifting circuit has at least* $\lfloor m/2\rfloor^2/162$ *vertices.*

Proof. Any shifting circuit has the shifting property. See [31], [33]. □

COROLLARY 3. Any planar circuit for computing Boolean convolution has at least $\lfloor m/2\rfloor^2/162$ vertices.

Proof. A circuit for computing Boolean convolution is a shifting circuit if we regard $x_1,\cdots,x_m$ as the primary inputs and $z_2,\cdots,z_{m+1}$ as the outputs. □

COROLLARY 4. Any planar circuit for computing the product of two m bit binary integers has at least $\lfloor m/2\rfloor^2/162$ vertices.

Proof. A circuit for multiplying two m-bit binary integers is a shifting circuit. □

The last result of this section is an $\Omega(m^4)$ lower bound on the size of any planar circuit for multiplying two $m\times m$ Boolean matrices. We shall assume that the inputs are x_{ij}, y_{ij} for $1\leqq i,j\leqq m$ and the outputs are z_{ij} for $1\leqq i,j\leqq m$. The circuit computes $Z=X\cdot Y$, where $Z=(z_{ij})$, $X=(x_{ij})$, and $Y=(y_{ij})$. We use the following property of circuits for multiplying Boolean matrices, called the *matrix concentration property* [31], [33]. For any k in the range $1\leqq k\leqq m^2$, any set $\{x_{i_rj_r}|1\leqq r\leqq k\}$ of k inputs from X, and any permutation σ of the integers one through m, there exist k vertex-disjoint paths from $\{x_{i_rj_r}|1\leqq r\leqq k\}$ to $\{z_{i_r\sigma(j_r)}|1\leqq r\leqq k\}$. Similarly, for any k in the range $1\leqq k\leqq m^2$, any set $\{y_{i_rj_r}|1\leqq r\leqq k\}$ of k inputs from Y, and any permutation σ of one through m, there exist k vertex-disjoint paths from $\{y_{i_rj_r}|1\leqq r\leqq k\}$ to $\{z_{\sigma(i_r)j_r}|1\leqq r\leqq k\}$.

THEOREM 8. *Any planar circuit G for multiplying two $m\times m$ Boolean matrices contains at least cm^4 vertices, for some positive constant c.*

Proof. This proof is somewhat involved, and we make no attempt to maximize the constant factor. Suppose G contains n vertices, and that m is even. Assign a cost of $1/(4m^2)$ to each input x_{ij} and each input y_{ij}, a cost of $1/(2m^2)$ to each output z_{ij}, and a cost of zero to every other vertex. By Theorem 2, there is a partition A, B, C of the vertices of G such that neither A nor B has total cost exceeding $\frac{1}{2}$, no edge joins a vertex in A with a vertex in B, and C contains no more than $c_1\sqrt{n}$ vertices. Without loss of generality, suppose that B contains no fewer outputs than A, and that A contains no fewer inputs x_{ij} than inputs y_{ij}. Then B contains at least $(m^2-c_1\sqrt{n})/2$ outputs, which contribute at least $1/4-c_1\sqrt{n}/(4m^2)$ to the cost of B. Thus inputs contribute at most $1/4-c_1\sqrt{n}/(4m^2)$ to the cost of B, and B contains at most $m^2+c_1\sqrt{n}$ inputs. A contains at least $2m^2-(m^2+c_1\sqrt{n})-c_1\sqrt{n}=m^2-2c_1\sqrt{n}$ inputs, of which at least $m^2/2-c_1\sqrt{n}$ are inputs x_{ij}. One of the following cases must hold.

Case 1. A contains at least $3m^2/5$ inputs x_{ij}. *Let* p be the number of columns of X which contain at least $4m/7$ elements of A. Then $pm+(m-p)(4m/7)\geqq 3m^2/5$, and $p\geqq m/15$. Let q be the number of columns of Z which contain at least $4m/9$ elements of B. Then $qm+(m-q)(4m/9)\geqq m^2/2-c_1\sqrt{n}/2$, and $q\geqq m/10-9c_1\sqrt{n}/(10m)$.

Let $k=\min\{m/15,\ m/10-9c_1\sqrt{n}/(10m)\}$. Choose any k columns of X, each of which contains at least $4m/7$ elements of A. Match each such column of X with a column of Z which contains at least $4m/9$ elements of B. For each pair of matched columns x_{*i}, z_{*j}, select a set of $4m/7+4m/9-m=m/63$ rows l such that x_{li} is in A and z_{lj} is in B. Such a selection gives a set of $km/63$ elements in $X\cap A$ and a set of $km/63$ elements in $Z\cap B$ which must be joined by $km/63$ vertex-disjoint paths, since G has the matrix concentration property. Each such path must contain a vertex of C. Thus $km/63\leqq c_1\sqrt{n}$, which means either $m^2/(15\cdot 63)\leqq c_1\sqrt{n}$ (i.e., $(m^2/(15\cdot 63c_1))^2\leqq n$) or $m/63(m/10-9c_1\sqrt{n}/(10m))\leqq c_1\sqrt{n}$ (i.e., $(m^2/(9\cdot 71c_1))^2\leqq n$).

Case 2. A contains fewer than $3m^2/5$ inputs x_{ij}. Then A contains at least $2m^2/5-2c_1\sqrt{n}$ inputs y_{ij}. Let S be the set of $m/2$ columns of Z which contain the most elements in B.

Subcase 2a. S contains at least $3m^2/10$ elements in B. Let p be the number of columns of X which contain at least $4m/9$ elements of A. Then $pm+4(m-p)m/9\geqq m^2/2-c_1\sqrt{n}$, and $p\geqq m/10-9c_1\sqrt{n}/(5m)$. Let q be the number of columns of S which contain at least $4m/7$ elements of B. Then $qm+4(m/2-q)m/7\geqq 3m^2/10$, and $q\geqq m/30$. A proof similar to that in Case 1 shows that $n\geqq \mathrm{c}m^4$ for some positive constant c.

Subcase 2b. S contains fewer than $3m^2/10$ elements in B. Then the $m/2$ columns of Z not in S contain at least $m^2/5-c_1\sqrt{n}/2$ elements in B. Let q be the number of columns of Z not in S which contain at least $m/10$ elements in B. Then $qm+(m/2-q)(m/10)\geqq m^2/5-c_1\sqrt{n}/2$, and $q\geqq m/6-5c_1\sqrt{n}/(9m)$. If $0\geqq q\geqq m/6-5c_1\sqrt{n}/(9m)$, then $(3m^2/(10c_1))^2\geqq n$. Hence assume $q>0$. Then all columns in S must contain at least $m/10$ elements in B, and $2m/3-5c_1\sqrt{n}/(9m)$ columns of Z must contain at least $m/10$ elements in B.

Let p be the number of columns of Y which contain at least $m/25$ elements of A. Then $pm+(m-p)(m/25)\geqq 2m^2/5-2c_1\sqrt{n}$, and $p\geqq 3m/8-25c_1\sqrt{n}/(12m)$.

For any input $y_{ij}\in A$ and integer l in the range $-m+1\leqq l\leqq m-1$, call y_{ij}, l a *match* if $z_{i+l,j}\in B$. By the previous computations, there are at least $2m/3-5c_1\sqrt{n}/(9m)+3m/8-25c_1\sqrt{n}/(12m)-m=m/25-95c_1\sqrt{n}/(36m)=m/25-c_2\sqrt{n}/m$ columns j such that y_{*j} contains $m/25$ elements of A and z_{*j} contains $m/10$ elements of B. Each such column produces $m^2/250$ matches; thus the total number of matches is at least $m^3/6250-mc_2\sqrt{n}/250$. Since there are only $2m-1$ values of l, some value of l produces at least $k=m^2/12{,}500-c_2\sqrt{n}/500$ matches. Since G has the matrix concentration property, this set of matches corresponds to a set of k elements in $Y\cap A$ and a set of k elements in $Z\cap B$ which must be joined by k vertex-disjoint paths. Each such path must contain a vertex in C. Thus $k\leqq c_1\sqrt{n}$, which means $m^4/(12{,}500(c_1+c_2/500))^2\leqq n$.

In all cases $n\geqq cm^4$ for some positive constant c. Choosing the minimum c over all cases gives the theorem for even m. The theorem for odd m follows immediately. □

The bounds in Theorems 6–8 and Corollaries 2–4 are tight to within a constant factor. We leave the proof of this fact as an exercise.

6. Embedding of data structures. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be undirected graphs. An *embedding* of G_1 in G_2 is a one-to-one map $\phi\colon V_1 \to V_2$. The *worst-case proximity* of the embedding is $\max\{d_2(\phi(v), \phi(w)) | \{v, w\} \in E_1\}$, where $d_2(x, y)$ denotes the distance between x and y in G_2. The *average proximity* of the embedding is $(1/|E_1|)/\sum\{d_2(\phi(v), \phi(w))|\{v, w\} \in E_1\}$. These notions arise in the following context. Suppose we wish to represent some kind of data structure by another kind of data structure, in such a way that if two records are logically adjacent in the first data structure, their representations are close together in the second. We can model the data structures by undirected graphs, with vertices denoting records and edges denoting logical adjacencies. The representation problem is then a graph embedding problem in which we wish to minimize worst-case or average proximity. See [5], [18], [26] for research in this area.

THEOREM 9. *Any planar graph with maximum degree k can be embedded in a binary tree so that the average proximity is $O(k)$.*

Proof. Let G be an n-vertex graph of maximum degree k. Embed G in a binary tree T by using the following recursive procedure. If G has one vertex v, let T be the tree of one vertex, the image of v. Otherwise, apply Corollary 1 to find a partition A, B, C of the vertices of G. Let v be any vertex in C (if C is empty, let v be a vertex in A). Embed the subgraph of G induced by $A \cup C - \{v\}$ in a binary tree T_1 by applying the method recursively. Embed the subgraph of G induced by B in a binary tree T_2 by applying the method recursively. Let T consist of a root (the image of v) with two children, the root of T_1 and the root of T_2. Note that the tree T constructed in this way has exactly n vertices.

Let $h(n)$ be the maximum depth of a tree T of n vertices produced by this algorithm. Then

$$\begin{aligned} &h(n) < 100 && \text{if } n < 100,\\ &h(n) \leqq h(2n/3 + 2\sqrt{2}\sqrt{n} - 1) + 1 \leqq h(29n/30) + 1 && \text{if } n \geqq 100. \end{aligned}$$

It follows that $h(n)$ *is* $O(\log n)$.

Let $G = (V, E)$ be an n-vertex graph to which the algorithm is applied, let G_1 be the subgraph of G induced by $A \cup C$, and let G_2 be the subgraph induced by B. If $s(G) = \sum\{d_2(\phi(v), \phi(w))|(v, w) \in E\}$, then $s(G) = 0$ if $n = 1$, and $s(G) \leqq s(G_1) + s(G_2) + 2k|C|h(n)$ if $n > 1$. This follows from the fact that any edge of G not in G_1 or G_2 must be incident to a vertex of C.

If $s(n)$ is the maximum value of $s(G)$ for any n-vertex graph G, then

$$\begin{aligned} &s(1) = 0;\\ &s(n) \leqq \max\{s(i) + s(n-i-1) + ck\sqrt{n}\log n | n/3 - 2\sqrt{2}\sqrt{n} \leqq i \leqq 2n/3 + 2\sqrt{2}\sqrt{n}\}\\ &\qquad\qquad\qquad \text{if } n > 1, \text{ for some positive constant } c. \end{aligned}$$

An inductive proof shows that $s(n)$ is $O(kn)$.

If G is a connected n-vertex graph embedded by the algorithm, then G contains at least $n - 1$ edges, and the average proximity is $O(k)$. If G is not connected, embedding each connected component separately and combining the resulting trees arbitrarily achieves an $O(k)$ average proximity. □

It is natural to ask whether *any* graph of bounded degree can be embedded in a binary tree with $O(1)$ average proximity. (Graphs of unbounded degree cannot be so embedded; a star consisting of a single vertex adjacent to $n - 1$ other vertices requires $\Omega(\log n)$ proximity.) Such is not the case, and in fact the property of being embeddable in a binary tree with $O(1)$ average proximity is closely related to the property of having a good separator. To make this statement more precise, let S be a class of graphs. The class S has an *$f(n)$-separator theorem* if there exist constants $\alpha < 1$, $\beta > 0$ such that the vertices of any n-vertex graph in S can be partitioned into three sets A, B, C such that $|A|$, $|B| \leqq \alpha n$, $|C| \leqq \beta f(n)$, and no vertex in A is adjacent to any vertex in B.

THEOREM 10. *Let S be any class of graphs of maximum degree k closed under the subgraph relation (i.e., if $G_1 \in S$ and G_2 is a subgraph of G_1, then $G_2 \in S$). Suppose S satisfies an $n/(\log n)^{2+\varepsilon}$ separator theorem for some fixed ε. Then any graph in S can be embedded in a binary tree with $O(k)$ average proximity.*

Proof. Similar to the proof of Theorem 9.

THEOREM 11. *Let $G = (V, E)$ be any graph of n vertices and m edges which is embeddable in a binary tree T with average proximity p. Then V can be partitioned into three sets A, B, C such that $|A|$, $|B| \leqq 2n/3$, $|C| \leqq cmp/\log n$ for some positive constant c, and no edge joins a vertex in A with a vertex in B.*

Proof. We can assume $m \geqq 2n/3$; otherwise the theorem is immediate. Let v be a vertex whose removal divides T into two or three connected components, each containing fewer than $2n/3$ vertices. Such a vertex can be found by initializing v to be the root of T and repeating the following step until it is no longer applicable: if some child w of v has at least $2n/3$ descendants, replace v by w. Let A be the set of vertices in G corresponding to the largest component of T when v is removed, let C be the set of vertices in $V - A$ adjacent to at least one vertex in A, and let $B = V - A - C$. By the choice of v and A, $|A| \leqq 2n/3$ and $|B| \leqq 2n/3$. Let $T(A)$, $T(B)$, $T(C)$ be the sets of vertices in T corresponding to A, B, C respectively. Since T is a binary tree, the number of vertices in $T(B) \cup T(C)$ within a distance of i from at least one vertex in $T(A)$ is at most $2^i - 1$. Thus the average proximity of the embedding of G in T is at least $|C| \cdot \lfloor \log_2 |C| \rfloor / (2m)$. This means $|C| \log |C| = O(mp)$, and $|C| = O(mp/\log n)$. □

Erdös, Graham, and Szemerédi [7] have shown that for c a large enough constant, almost all graphs of cn edges cannot be separated into small components without removing $\Omega(n)$ vertices. It follows from Theorem 11 that almost all graphs of cn edges require $\Omega(\log n)$ average proximity when embedded in binary trees.

7. Maximum matching.

Let $G = (V, E)$ be an undirected graph. A *matching* $M \subseteq E$ is a set of edges no two of which have a common endpoint. A *maximum cardinality* matching is a matching M such that $|M|$ is maximum. If each edge $e \in E$ has an associated real-valued *weight* $w(e)$, a *maximum weight matching* is a matching M such that $\sum_{e \in M} w(e)$ is maximum. By using Corollary 1, we can find maximum cardinality matchings in planar graphs in $O(n^{3/2})$ time and maximum weight matchings in $O(n^{3/2} \log n)$ time. For arbitrary graphs, the best known algorithms require $O(\sqrt{n}\, m \log \log n)$ time to find maximum cardinality matchings [14] and $O(mn \log n)$ time to find maximum weight matchings [8], where $m = |E|$. For planar graphs, these bounds are $O(n^{3/2} \log \log n)$ and $O(n^2 \log n)$, respectively.

To describe the method, we need a few ideas from matching theory. If M is a matching in a graph C, an *unmatched vertex* is a vertex incident to no edge of M. An *alternating path* is a simple path or simple cycle whose edges are alternately in M and not in M. The *net weight* of an alternating path is the total weight of its unmatched edges minus the total weight of its matched edges. An alternating path is *augmenting* if its net weight is positive and it is either a cycle or each of its first and last edges is either in M or incident to an unmatched vertex. Given an augmenting path, we can increase the weight of the matching by adding to M all previously unmatched edges on the path and deleting from M all previously matched edges on the path. Conversely, if there is no augmenting path, then M is of maximum weight. The next lemma provides a way to update a maximum weight matching when a single vertex is added to a graph.

LEMMA 2. *Let $G=(V,E)$ be an undirected graph with edge weights $w(e)$, let $v \in V$, and let $G-v$ be the subgraph of G induced by the vertex set $V-\{v\}$. Suppose M is a maximum weight matching in $G-v$. If G contains no augmenting path (with respect to M) with v as one endpoint, then M is a maximum weight matching of G. Otherwise, let P be the edge set of an augmenting path of maximum net weight. Then $M \oplus P = M \cup P - (M \cap P)$ is a maximum weight matching in G.*

Proof. Let M_0 be a maximum weight matching in G. Consider $M \oplus M_0 = M \cup M_0 - (M \cap M_0)$. Every vertex in G is incident to at most two edges of $M \cup M_0$; thus $M \oplus M_0$ consists of a set of simple cycles and simple paths in G, each of which is an alternating path with respect to M_0. Any augmenting path in $M \oplus M_0$ must have v as an endpoint, or else M would not be of maximum weight in $G-v$. (Note that v is incident to at most one edge in $M \oplus M_0$). Thus $M \oplus M_0$ contains at most one augmenting path, and such a path has v as one endpoint. The lemma follows. □

Given a suitable representation of a maximum weight matching M in $G-v$, a maximum weight matching M_0 in G can be found in $O(m \log n)$ time by applying Lemma 2; see [8] for details. Thus applying Lemma 2 to a planar graph requires $O(n \log n)$ time. In the maximum cardinality case, all the weights are one, and application of Lemma 2 requires $O(m)$ time on an arbitrary graph, $O(n)$ time on a planar graph [13].

The following recursive algorithm makes use of Corollary 1 and Lemma 2 to find maximum weight matchings.

Step 1. If G contains at most one vertex, return the empty set as a maximum weight matching.

Step 2. Otherwise, apply Corollary 1 to G. Let A, B, C be the resulting vertex partition and let G_A, G_B be the subgraphs of G induced by the vertex sets A, B, respectively. Apply the algorithm recursively to find maximum weight matchings M_A in G_A, M_B in G_B. Let $M = M_A \cup M_B$, $S = A \cup B$.

Step 3. Add C one vertex at a time to S. Each time a vertex is added to S, apply Lemma 2 to replace M by a maximum weight matching in G_S, the subgraph of G induced by the vertex set S. Stop when $S = V$.

After Step 2, $M = M_A \cup M_B$ is a maximum weight matching of $G_{A \cup B}$. It follows from Lemma 2 that after Step 3, M is a maximum weight matching of G_V. If $t(n)$ is the running time of the algorithm on an n-vertex graph, then

$$t(l) = c_1;$$

$$t(n) \leqq \max\{t(n_1) + t(n_2) + c_2 n^{3/2} \log n \,|\, n_1 + n_2 \leqq n;\ n_1, n_2 \leqq 2n/3\} \quad \text{if } n > 1,$$

where c_1 and c_2 are suitable positive constants, since $|C| = O(\sqrt{n})$. An inductive proof shows that $t(n) = O(n^{3/2} \log n)$. In the maximum cardinality case, the algorithm requires only $O(n^{3/2})$ time.

8. Remarks. Theorem 1 and its corollaries have applications beyond those in this paper. For instance, the planar separator theorem can be used to generalize George's "nested dissection" method [10] for carrying out sparse Gaussian elimination on a system of linear equations whose sparsity structure corresponds to a square grid. The generalized method solves any linear system whose sparsity structure corresponds to an n-vertex planar graph in $O(n^{3/2})$ time and $O(n \log n)$ space [19]. Theorem 2 can be employed to give a rather complicated $O(\log n)$ time, $O(n)$-space solution [21] to the closest-point searching problem in two dimensions, sometimes called the post office problem [16]. The previously best solutions to this problem required either $O(\log n)$ time and $O(n^2)$ space [29], or $O((\log n)^2)$ time and $O(n)$ space [6], [29]. Recently Kirkpatrick [15] has discovered a simple $O(\log n)$-time, $O(n)$-space solution which does not use the separator theorem. We leave further applications of the separator theorem to the reader.

Although most sparse graphs do not have good separators, there are other classes besides planar graphs and graphs of fixed genus which do (see e.g. [19]). The results discussed in this paper generalize to any such class. In some of the problems we have examined, such as graph embedding (§ 6) and sparse Gaussian elimination [19], the existence of good separators is not only a sufficient but also a necessary condition for efficient solution of the problem. This phenomenon deserves more study, and suggests that for certain graph problems it may be valuable to define the concept of "usefully sparse" as meaning that a graph has good separators.

Acknowledgment. We would like to thank Hal Gabow for his very perceptive comments and for suggesting the results in § 7. A preliminary version of this paper was presented at the Eighteenth Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, 1977.

REFERENCES

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

[2] M. O. ALBERTSON AND J. P. HUTCHINSON, *On the independence ratio of a graph*, J. Graph Theory, 2 (1978), pp. 1–8.

[3] U. BERTELE AND F. BRIOSCHI, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

[4] S. A. COOK, *An observation on time-storage tradeoff*, Proc. Fifth Annual ACM Symp. on Theory of Computing (1973), pp. 29–33.

[5] R. A. DEMILLO, S. C. EISENSTAT AND R. J. LIPTON, *Preserving average proximity in arrays*, Comm. ACM, 21(1978), pp. 228–230.

[6] D. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, this Journal, 5 (1976), pp. 181–186.

[7] P. ERDÖS, R. L. GRAHAM AND E. SZEMERÉDI, *On sparse graphs with dense long paths*, Comp. and Math. with Appl., 1 (1975), pp. 365–369.

[8] H. GABOW, *An efficient implementation of Edmonds' algorithm for maximum weight matching on graphs*, Technical Report CU-CS-075-75, University of Colorado, Boulder, Colorado (1975).

[9] M. R. GAREY, D. S. JOHNSON, F. P. PREPARATA, AND R. E. TARJAN, *Triangulating a simple polygon*, Informat. Processing, Letters, 7 (1978), pp. 175–179.

[10] J. A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[11] L. GOLDSCHLAGER, *The monotone and planar circuit value problems are log space complete for P*, ACM SIGACT News 9, 2 (1977), pp. 25–29.

[12] J. HOPCROFT, W. PAUL AND L. VALIANT, *On time versus space*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.

[13] T. KAMEDA AND I. MUNRO, *A $O(VE)$ algorithm for maximum matching of graphs*, Computing 12 (1974), pp. 91–98.

[14] O. KARIV, *An $O(n^{2.5})$ algorithm for finding a maximum matching on a general graph*, Ph.D dissertation, Weizmann Institute of Science, Rehovot, Israel, 1976.

[15] D. KIRKPATRICK, private communication, 1979.

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

[17] D. KOZEN, *On parallelism in Turing machines*, Proc. Seventeenth Annual Symp. on Foundations of Computer Science, 1976, pp. 89–97.

[18] R. J. LIPTON, S. C. EISENSTAT, AND R. A. DEMILLO, *Space and time hierarchies for control structures and data structures*, J. Assoc. Comput. Mach., 23 (1976), pp. 720–732.

[19] R. J. LIPTON, D. J. ROSE, AND R. E. TARJAN, *Generalized nested dissection*, SIAM J. Numer. Anal., 16 (1979), pp. 346–358.

[20] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36(1979), pp. 177–189.

[21] ———, *Applications of a planar separator theorem*, Proc. 18th Annual Symp. on Foundations of Computer Science (1977), pp. 162–170.

[22] H. C. MARTIN AND G. F. CAREY, *Introduction to Finite Element Analysis*, McGraw-Hill, New York, 1973.

[23] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Record of Project MAC Conf. on Concurrent Systems and Parallel Computation (1970), pp. 119–128.

[24] M. S. Paterson, *Tape bounds for time-bounded Turing machines*, J. Comput. System Sci., 6 (1972), pp. 116–124.

[25] W. J. PAUL, R. E. TARJAN, AND J. R. CELONI, *Space bounds for a game on graphs*, Math. Systems Theory 10 (1977), pp. 239–251.

[26] A. L. ROSENBERG, *Managing storage for extendible arrays*, this Journal, 4 (1975), pp. 287–306.

[27] A. ROSENTHAL, *Nonserial dynamic programming is optimal*, Proc. Ninth Annual ACM Symp. on Theory of Computing (1977), pp. 98–105.

[28] R. SETHI, *Complete register allocation problems*, this Journal, 4 (1975), pp. 226–248.

[29] M. J. SHAMOS, *Geometric complexity*, Proc. Seventh Annual ACM Symp. on Theory of Computing (1975), pp. 224–233.

[30] N. SIDER, *Partial colorings and limiting chromatic numbers*, Ph.D. dissertation, Syracuse University, Syracuse, NY (1971).

[31] L. G. VALIANT, *On non-linear lower bounds in computational complexity*, Proc. Seventh Annual ACM Symp. on Theory of Computing (1975), pp. 45–53.

[32] P. UNGAR, *A theorem on planar graphs*, J. London Math. Soc., 26 (1951), pp. 256–262.

[33] L. G. VALIANT, *Graph-theoretic arguments in low-level complexity*, Computer Science Dept., University of Edinburgh, 1977.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

RANDOM GRAPH ISOMORPHISM*

LÁSZLÓ BABAI†, PAUL ERDŐS‡ AND STANLEY M. SELKOW§

Abstract. A straightforward linear time canonical labeling algorithm is shown to apply to almost all graphs (i.e. all but $o(2^{\binom{n}{2}})$ of the $2^{\binom{n}{2}}$ graphs on n vertices). Hence, for almost all graphs X, any graph Y can be easily tested for isomorphism to X by an extremely naive linear time algorithm. This result is based on the following: In almost all graphs on n vertices, the largest $n^{0.15}$ degrees are distinct. In fact, they are pairwise at least $n^{0.03}$ apart.

Key words. graph, isomorphism testing, canonical labeling, random graph, naive algorithm, average-case analysis, linear time, degree sequence of a graph

1. A straightforward algorithm.

The problem of testing graphs for isomorphism belongs to those combinatorial search problems for which no polynomial-time algorithm is available as yet. It is, however, striking, that even the most trivial isomorphism testing algorithms have a good performance if tested on randomly generated graphs. The aim of the present note is to give some theoretical background for this.

By a *canonical labeling algorithm* of the class $\mathcal{K}$ of graphs we mean an algorithm which assigns the numbers $1, \cdots, n$ to the vertices of each graph in $\mathcal{K}$, having n vertices, in such a way that two graphs in $\mathcal{K}$ are isomorphic (if and) only if the obtained labeled graphs coincide. (We assume that $\mathcal{K}$ is closed under isomorphisms.) Clearly, given a canonical labeling algorithm of $\mathcal{K}$, and an algorithm deciding whether a given graph belongs to $\mathcal{K}$ or not, we also have an algorithm, deciding whether $X \cong Y$ for any two graphs X, Y provided $X \in \mathcal{K}$. Namely, if $Y \notin \mathcal{K}$ then $X \not\cong Y$; and if $Y \in \mathcal{K}$ then we have to check whether X and Y coincide after canonical labeling.

We describe a class $\mathcal{K}$ of graphs (closed under isomorphisms) and a canonical labeling algorithm of $\mathcal{K}$. Deciding whether $X \in \mathcal{K}$ and subsequently, canonically labeling X will require *linear time* (i.e. $O(n^2)$, where n is the number of vertices) on a random access machine which operates in one step on binary words of length $O(\log n)$. We shall prove, that $\mathcal{K}$ contains *almost all* graphs on n vertices (i.e. all but $o(2^{\binom{n}{2}})$ of the graphs on a fixed vertex set of cardinality n). In particular, we prove

THEOREM 1.1. *There is an algorithm which, for almost all graphs X, tests any graph Y for isomorphism to X within linear time.*

The algorithm is as follows:

Input: a graph X having n vertices. (The graph is represented by its adjacency matrix.)

1. Compute $r = [3 \log n/\log 2]$.
2. Compute the degree of each vertex of X.
3. Order the vertices by degree; call them $v(1), \cdots, v(n)$. Denote by $d(i)$ the degree of $v(i)$: $d(1) \geqq d(2) \geqq \cdots \geqq d(n)$.
4. If $d(i) = d(i+1)$ for some i, $1 \leqq i \leqq r-1$, set $X \notin \mathcal{K}$, end. Otherwise
5. Compute

$$f(v(i)) = \sum_{j=1}^{r} a(i, j)2^j \qquad (i = r+1, \cdots, n)$$

* Received by the editor October 9, 1977.

† Department of Algebra and Number Theory, Eötvös L. University, H-1445 Budapest 8, Pf. 323, Hungary.

‡ Mathematics Institute of the Hungarian Academy of Sciences, Reáltanoda u. 13–15, H-1053 Budapest, Hungary.

§ Department of Computer Science, Concordia University, Montréal, Quebec, Canada.

(the "code of $v(i)$ with respect to $v(1), \cdots, v(r)$"), where $a(i, j) = 1$ if $v(i)$ and $v(j)$ are adjacent and $a(i, j) = 0$ otherwise.

6. Order the vertices $v(r+1), \cdots, v(n)$ according to their f-value: $w(r+1), \cdots, w(n)$ where $f(w(r+1)) \geqq \cdots \geqq f(w(n))$.
7. If $f(w(i)) = f(w(i+1))$ for some $i, r+1 \leqq i \leqq n-1$, set $X \notin \mathcal{K}$, end. Otherwise
8. Label $v(i)$ by i for $i = 1, \cdots, r$, and $w(i)$ by i for $i = r+1, \cdots, n$. This labeling will be called canonical. Set $X \in \mathcal{K}$. End.

In other words, the first r labels will be assigned to the vertices with largest degrees, in decreasing order of the degree. If this is not unique, then $X \notin \mathcal{K}$. The rest of the labels will be assigned to the remaining $n-r$ vertices in decreasing order of their codes with respect to the first r vertices, as defined in step 5. Again, if two vertices get the same code then $X \notin \mathcal{K}$.

Obviously, this algorithm defines a canonical labeling, indeed, and $\mathcal{K}$ is closed under isomorphisms. The running time of the algorithm is $O(n^2)$, as readily verified. Our principal result is the following:

THEOREM 1.2. *The probability that a random graph on n vertices belongs to the class $\mathcal{K}$, specified by our canonical labeling algorithm, is greater than* $1 - \sqrt[7]{1/n}$ *(for sufficiently large n).*

This clearly implies Theorem 1.1.

At this point we have to stress that our algorithm is not intended for practical use: more involved but still very natural heuristic algorithms are much better. Our purpose is to show that *even such an extremely naive, fast algorithm solves the problem for almost all graphs.*

The referee and the first named author share the responsibility for almost two years delay in publishing this paper. Since 1977, the paper has been circulated as a preprint essentially in its present form (except for the introduction and a simplification of the proof in § 4, suggested by the referee).

In the preprint we formulated the following two problems:

(i) Find a fast canonical labeling algorithm with exponentially small probability of rejection.

(ii) Find a canonical labeling algorithm of *all* graphs, with polynomial expected running time.

The preprint seems to have inspired further work instantaneously. Both problems have been solved shortly after submission of this paper. R. Lipton [8] gives a canonical labeling algorithm with $O(n^6 \log n)$ running time and exponentially small probability of rejection ($c^{-n}, c > 1$). R. M. Karp [7] improves this, giving an $O(n^2 \log n)$ algorithm, with $O(n^{3/2} 2^{-n/2})$ probability of rejection. Babai and Kučera [1] prove that the standard vertex classification algorithm gives a canonical labeling in $O(n^2)$ time with c^{-n} probability of rejection. In addition, it is proved in [1] that the rejected graphs can be handled such as to obtain a canonical labeling algorithm of *all* graphs with *linear expected time*, i.e. the average running time over the $2^{\binom{n}{2}}$ graphs is $O(n^2)$.

This short survey tends to convince us that, despite of the long delay, the present note may merit some attention. Apart from [1], it still appears to be the only example of a *linear* time canonization of almost all graphs. [1] definitely outscores our results, but the simplicity of our algorithm can hardly be improved on, and it may be worth noting that still, such an algorithm canonizes almost all graphs.

The performance of our algorithm relies on our results on the degree sequence of a random graph. This aspect of the paper, which extends the idea of [4], may have interest on its own. The results of § 3 are stronger than what would be necessary to prove the main theorem. Recently, B. Bollobás [2] has obtained finer and more detailed results on this subject.

More about random graphs can be found in Erdős–Spencer [3].

Concerning the probabilistic analysis of some hard combinatorial problems we refer to Karp [6].

2. Preliminaries. Throughout this paper, we shall use the following notation:

$$P(m, l) = \sum_{s=l+1}^{m} \binom{m}{s}. \tag{1}$$

Clearly,

$$P(m, l) = P(m-1, l) + P(m-1, l-1). \tag{2}$$

We shall refer to the following well-known asymptotic formula:

$$\binom{m}{m/2+t} = \binom{m}{[m/2]} e^{-2t^2/m + O(\tau)} \tag{3}$$

where $\tau = t^2/m^2 + t^4 m^3$ (cf. Feller [5, Chap. VII/2]). The O notation always refers to absolute constants (not depending on any of our parameters). Of course, $m/2+t$ should be an integer. This means that t is either an integer or a half-integer, depending on the parity of m. Similar restrictions on the possible values of parameters are understood throughout without explicit mention.

Random variables are denoted by block letters. A *random graph* $\mathbf{X}$ on the vertex set $V = \{1, \cdots, n\}$ assumes as its values each graph on V with probability $2^{-\binom{n}{2}}$.

We start with some elementary computation with binomial coefficients.

PROPOSITION 2.1. *If* $l = m/2 + t$ $(0 < t < m/2)$ *and* $f > r(\log 2/2)\dfrac{m}{t}$, *then*

$$\binom{m}{l+f} < 2^{-r}\binom{m}{l}.$$

Proof.

$$\binom{m}{l+f} \Big/ \binom{m}{l} = (m-l) \cdots (m-l-f+1)/(l+f) \cdots (l+1) < \left(\frac{m-l}{l+1}\right)^f < \left(\frac{m-l}{m/2}\right)^f$$

$$= (1 - 2t/m)^f < \exp(-2tf/m) < \exp(-r \log 2) = 2^{-r}. \quad \square$$

COROLLARY 2.2. *If* $l = m/2 + t$ $(0 < t < m/2)$ *then*

$$P(m, l) \Big/ \binom{m}{l} < \frac{m}{t}.$$

Proof. Let $g = [m \log 2/(2t)] + 1$. Then, by Proposition 2.1,

$$P(m, l) < \binom{m}{l} \cdot (g + g/2 + g/4 + \cdots) < 2g\binom{m}{l} < \frac{m}{t}\binom{m}{l}. \quad \square$$

On the other hand, a lower bound of the same order of magnitude also holds. To this end, we need another simple estimate:

PROPOSITION 2.3. *If* $l = m/2 + t$ $(0 < t < m/2)$ *and* $0 < f < t$, *then*

$$\binom{m}{l} > \binom{m}{l-f}(1 - 4tf/m).$$

Proof.

$$\binom{m}{l}\Big/\binom{m}{l-f} = (m-l+f)\cdots(m-l+1)/l(l-1)\cdots(l-f+1) > ((m-l)/l)^f$$

$$= \left(\frac{m/2-t}{m/2+t}\right)^f > (1-2t/m)^{2f} > 1-4tf/m. \qquad \square$$

COROLLARY 2.4. *If* $l = m/2+t$ $(2\sqrt{m} < t < m/30)$, *then*

$$P(m, l)\Big/\binom{m}{l} > \frac{m}{23t}.$$

Proof. For any natural number f, obviously

$$P(m, l)\Big/\binom{m}{l} > f\binom{m}{l+f}\Big/\binom{m}{l}.$$

By Proposition 2.3, the right side exceeds $f(1-4(t+f)f/m)$. Set $f = [m/9t]+1$. So, $f > m/9t$ and $f < m/9t+1 < t/27$; hence our quantity exceeds

$$\frac{m}{9t}\left(1-\frac{4\cdot 28}{9\cdot 27}-\frac{4\cdot 28}{27}\frac{t}{m}\right) > \frac{m}{23t}. \qquad \square$$

3. The largest degrees are distinct. Let $\mathbf{X}$ be a random graph on the vertex set $\{1, \cdots, n\}$. Let $\mathbf{d}(x)$ denote the degree of the vertex x. Let us fix a natural number d, and set $\mathbf{z}_x = 0$ if $\mathbf{d}(x) \leqq d$, $\mathbf{z}_x = 1$ otherwise. Let $\mathbf{z} = \sum_{x=1}^{n} \mathbf{z}_x$.

We are interested in the behavior of the expected value $\mathrm{E}(\mathbf{z})$ (depending on the choice of d).

LEMMA 3.1. *Let* $m = n-1$, $d = m/2+t$ *where*

$$t = t_0 + \omega_m(m/\log m)^{1/2},$$

where

$$t_0 = (\tfrac{1}{2}m \log m)^{1/2} - \tfrac{1}{8}(2m/\log m)^{1/2} \log\log m,$$

and

$$-\log m/\sqrt{2} < \omega_m < m^{0.7}.$$

If $\omega_m < 0$ *then*

$$\mathrm{E}(\mathbf{z}) > c_1\, e^{-1.4\omega_m};$$

if $\omega_m > 0$ *then*

$$\mathrm{E}(\mathbf{z}) < c_2 \exp(-2.8\omega_m - 2\omega_m^2/\log m).$$

If $\omega_m/\log m \to -\varepsilon/\sqrt{2}$ $(m \to \infty)$ *where* $O < \varepsilon < 1$ (ε *is fixed*) *then*

$$\mathrm{E}(\mathbf{z}) > m^{\varepsilon(2-\varepsilon+o(1))}.$$

Proof. Clearly, for any $x (1 \leqq x \leqq n)$,

$$\mathrm{E}(\mathbf{z}) = n\mathrm{E}(\mathbf{z}_x) = n2^{-n+1}P(n-1, d)$$
$$= (1+o(1))m2^{-m}P(m, d).$$

Now we apply Corollaries 2.2 and 2.4 to obtain θ, $0 < \theta < 1$ such that

$$P(m, d) = \frac{1}{1+22\theta}\binom{m}{d}\frac{m}{t} = \frac{1+o(1)}{1+22\theta}2^m m\, e^{-2t^2/m}/(t(\tfrac{1}{2}\pi m)^{1/2})$$

(by (3)). Hence,

$$\log \mathrm{E}(\mathbf{z}) = O(1) + 3 \log m/2 - \log t - 2t^2/m.$$

For $t = t_0$ the right side is bounded; hence in the general case,

$$\begin{aligned}\log \mathrm{E}(\mathbf{z}) &= O(1) - \log (t/t_0) - 2(t^2 - t_0^2)/m \\ &= O(1) - \log(1 + \omega_m\sqrt{2}/\log m) - 2\omega_m(\sqrt{2} - \sqrt{2} \log\log m/4 \log m + \omega_m/\log m).\end{aligned}$$

Now our assertions can be readily checked. □

COROLLARY 3.2. *With the notation of Lemma* 3.1, *the probability that* $\mathbf{x}$ *has a vertex of degree* $> t_0 + \omega_m(m/\log m)^{1/2}$ *is less than* $c_2 \exp(-2.8\omega_m - 2\omega_m^2/\log m)(\omega_m > 0)$.

In order to obtain the counterpart of Corollary 3.2 for $\omega_m < 0$, we have to compute the variance of $\mathbf{z}$.

LEMMA 3.3. *Let* $m = n - 1$ *and* $d = m/2 + t$, *where* $2\sqrt{m} < t < m/30$. *Then*

$$\mathrm{Var}(\mathbf{z})/\mathrm{E}(\mathbf{z})^2 < 1/\mathrm{E}(\mathbf{z}) + 67t^2/m^2.$$

Proof. Clearly,

$$\mathrm{Var}\,\mathbf{z} = \mathrm{E}(\mathbf{z}^2) - \mathrm{E}(\mathbf{z})^2 = mA + \binom{n}{2}B,$$

where

$$\begin{aligned} A &= \mathrm{E}(\mathbf{z}_x)(1 - \mathrm{E}(\mathbf{z}_x) < \mathrm{E}(\mathbf{z}_x) &\quad& (\text{hence } nA < \mathrm{E}(\mathbf{z})), \\ B &= \mathrm{E}(\mathbf{z}_x\mathbf{z}_y) - \mathrm{E}(\mathbf{z}_x)^2 && (\text{for any } 1 \leqq x < y \leqq n).\end{aligned}$$

Clearly, for $x \neq y$

$$\mathrm{E}(\mathbf{z}_x\mathbf{z}_y) = \mathrm{Prob}(\mathbf{d}(x) > d \text{ and } \mathbf{d}(y) > d) = (P_1 + P_2)/2,$$

where P_1, P_2 are conditional probabilities:

$$\begin{aligned} P_1 &= \mathrm{Prob}(\mathbf{d}(x) > d \text{ and } \mathbf{d}(y) > d \mid x \text{ and } y \text{ are adjacent}) \\ &= 2^{-2n+4}P(n-2, d-1)^2; \\ P_2 &= \mathrm{Prob}(\mathbf{d}(x) > d \text{ and } \mathbf{d}(y) > d \mid x \text{ and } y \text{ are not adjacent}) \\ &= 2^{-2n+4}P(n-2, d)^2.\end{aligned}$$

It follows (using (2)), that

$$\begin{aligned} B &= 2^{-2n+2}(2P(n-2, d-1)^2 + 2P(n-2, d)^2 - P(n-1, d)^2) \\ &= 2^{-2n+2}(P(n-2, d-1) - P(n-2, d))^2 \\ &= 2^{-2n+2}\binom{n-2}{d}^2.\end{aligned}$$

Hence

$$\begin{aligned} \frac{\mathrm{Var}\,\mathbf{z}}{\mathrm{E}(\mathbf{z})^2} &< \frac{1}{\mathrm{E}(\mathbf{z})} + \binom{n}{2}B/\mathrm{E}(\mathbf{z})^2 < \frac{1}{\mathrm{E}(\mathbf{z})} + \frac{1}{2}\binom{n-2}{d}^2/P(n-1, d)^2 \\ &< \frac{1}{\mathrm{E}(\mathbf{z})} + \frac{1}{8}\left(\binom{m}{d}/P(m, d)\right)^2 < \frac{1}{\mathrm{E}(\mathbf{z})} + \frac{1}{8}\left(\frac{m}{23t}\right)^2 \\ &< 1/\mathrm{E}(\mathbf{z}) + (23t/m)^2/8 < 1/\mathrm{E}(\mathbf{z}) + 67t^2/m^2.\end{aligned}$$

(We have used here the inequality $\binom{n-2}{d} < \frac{1}{2}\binom{n-1}{d}$ which trivially holds for $d > n/2$; and subsequently Corollary 2.4.) □

COROLLARY 3.4. *If the sequence d_n is so chosen that (setting $d=d_n$) we obtain* $\mathrm{E}(\mathbf{z})\to\infty$ $(n\to\infty)$, *then*

$$\operatorname{Prob}(\mathbf{z}<\mathrm{E}(\mathbf{z})/2)\to 0.$$

Using the notation of Lemma 3.1, *for* $-\log m\sqrt{2}<\omega_m<0$ *we have*

$$\operatorname{Prob}(\mathbf{z}<\mathrm{E}(\mathbf{z})/2)<c_3\, e^{1.4\omega_m}.$$

If $\omega_m/\log m\to\varepsilon/\sqrt{2}$ *where* $0<\varepsilon<1$ (ε *is fixed*), *then*

$$\operatorname{Prob}(\mathbf{z}<\mathrm{E}(\mathbf{z})/2)<c_4 m^{-\varepsilon(2-\varepsilon+o(1))}.$$

Proof. By Chebyshev's inequality,

$$\operatorname{Prob}(\mathbf{z}<\mathrm{E}(\mathbf{z})/2)<4\operatorname{Var}\mathbf{z}/\mathrm{E}(\mathbf{z})^2.$$

This implies our second statement, by Lemmas 3.3 and 3.1. Namely,

$$t^2<m\log m+2\omega_m^2 m/\log m=O(m\log m),$$

hence

$$1/\mathrm{E}(\mathbf{z})+67t^2/m^2<e^{-1.4\omega_m}+O(\log m/m)$$

and $\log m/m=\exp(\log\log m-\log m)=o(\exp(-1.4\log m/\sqrt{2}))=o(e^{-1.4\omega_m})$. The third statement follows similarly.

For the first statement, by Lemma 3.3 and Chebyshev's inequality we only have to prove that if $\mathrm{E}(\mathbf{z})\to\infty$ then $t/m\to 0$. This is obvious from Lemma 3.1. □

LEMMA 3.5. *Let* $0<k<\sqrt{n}$, $\sqrt{3}/2<\alpha<1$ *and* $t=\alpha(n\log n)^{1/2}$. *Then the probability of the event that* $\mathbf{X}$ *has two vertices* x, y *of degrees exceeding* $n/2+t$ *such that* $|\mathbf{d}(x)-\mathbf{d}(y)|<k$ *is*

$$p=o(kn^{3/2-2\alpha^2})\qquad(n\to\infty).$$

Proof. Let $n/2<a\leqq b$. The probability that $\mathbf{d}(x)=a$ and $\mathbf{d}(y)=b$ $(x\neq y)$ is clearly

$$\frac{1}{2}\left(\binom{n-2}{a}\binom{n-2}{b}+\binom{n-2}{a-1}\binom{n-2}{b-1}\right)\Big/2^{2n-4}<2^{-2n+4}\binom{n-2}{a-1}^2.$$

Hence, the probability that $a\leqq\mathbf{d}(x)\leqq\mathbf{d}(y)\leqq\mathbf{d}(x)+k$ is at most

$$k\cdot 2^{-2n+4}\sum_{s=a}^{n-1}\binom{n-2}{s-1}^2.$$

By Proposition 2.1, the sum here is less than

$$\frac{n}{a-n/2}\binom{n-2}{a-1}^2.$$

Setting $a=[n/2+t]$, we obtain (by (3))

$$\begin{aligned}p&<\binom{n}{2}k\frac{n}{t}e^{-4t^2/n}/(\tfrac{1}{2}\pi n)(1+o(1))\\&=\frac{\sqrt{2}(1+o(1))}{\pi\alpha}n^2k(n\log n)^{-1/2}n^{-2\alpha^2}\\&<kn^{3/2-2\alpha^2}/(\log n)^{1/2}\end{aligned}$$

(for n not too small). □

Now we are in the position to prove

THEOREM 3.6. *Let $\mathbf{d}_1 \geqq \mathbf{d}_2 \geqq \cdots \geqq \mathbf{d}_n$ denote the degrees of the vertices of the random graph $\mathbf{X}$. Let $k = [n^{0.03}]$ and $l = [n^{0.15}]$. For n sufficiently large, the event that $\mathbf{d}_i - \mathbf{d}_j \geqq k$ for every i, j satisfying $1 \leqq i < j \leqq l$ has probability exceeding $1 - n^{-0.15}$.*

Proof. Set $\varepsilon = \frac{1}{12}$, $\alpha = 1 - \varepsilon$, $t = \alpha(n \log n)^{1/2}/\sqrt{2}$, $d = [n/2 + t]$. Then, with the notation of Lemma 3.1, $t = t_0 + \omega_m (m/\log m)^{1/2}$ where $\omega_m/\log m \to -\varepsilon/\sqrt{2}$, whence, by 3.1,

$$\mathrm{E}(\mathbf{z}) > m^{\varepsilon(2-\varepsilon+o(1))}.$$

$\varepsilon(2-\varepsilon) > 0.157$, hence $\mathrm{E}(\mathbf{z})/2 > n^{0.15} \geqq l$ for sufficiently large n. By Cor. 3.4, this implies that $\mathbf{X}$ has at least $\mathrm{E}(\mathbf{z})/2$ vertices of degree $> d$ with probability $> 1 - c_4 m^{-\varepsilon(2-\varepsilon+o(1))} > 1 - m^{-0.155}$. Finally, by Lemma 3.5, the difference between the degrees of any two of these vertices is at least k with probability $> 1 - kn^{3/2 - 2\alpha^2} > 1 - n^{0.03+1.5-2\alpha^2} > 1 - n^{-0.1505}$. Hence the probability that $\mathbf{X}$ does not satisfy the theorem is less than

$$n^{-0.155} + n^{-0.1505} < n^{-0.15}. \qquad \square$$

Remark. The particular corollary to Theorem 3.6, that the vertex having maximum degree is unique in almost all graphs, appears in Erdős–Wilson [4].

4. Uniqueness of the codes of the vertices. As in § 3, let $\mathbf{X}$ be a random graph having $V = \{1, \cdots, n\}$ for its vertex set. Let $\mathbf{d}_1 \geqq \mathbf{d}_2 \geqq \cdots \geqq \mathbf{d}_n$ denote the degree sequence of $\mathbf{X}$. Set $r = [3 \log n/\log 2]$, and let C denote the event that $\mathbf{d}_i \geqq \mathbf{d}_{i+1} + 3$ for $i = 1, \cdots, r+2$. We write $\bar{C}$ for the negation of C. By Theorem 3.6,

$$\text{Prob}\,(\bar{C}) < n^{-0.15}.$$

For $i \neq j$, let $C(i, j)$ denote the event that in the graph $\mathbf{X}(i, j)$ obtained from $\mathbf{X}$ by deleting i and j, the largest r degrees are distinct. Clearly, C implies $C(i, j)$ for all $i, j (1 \leqq i \leqq j \leqq n)$. Let $A(i, j)$ denote the event that either $C(i, j)$ fails or i and j have identical codes with respect to the vertices having the largest r degrees in $\mathbf{X}(i, j)$.

The probability that $\mathbf{X}$ is rejected by our algorithm is less than

$$\begin{aligned} &\text{Prob}\,(\bar{C}) + \text{Prob}\,(C \text{ and the graph } \mathbf{X} \text{ has two vertices} \\ &\qquad \text{with identical codes}) \\ &\leqq \text{Prob}\,(\bar{C}) + \sum_{i<j} \text{Prob}\,(C \text{ and } A(i, j)) \\ &\leqq \text{Prob}\,(\bar{C}) + \sum_{i<j} \text{Prob}\,(C(i, j) \text{ and } A(i, j)) \\ &\leqq \text{Prob}\,(\bar{C}) + \sum_{i<j} \text{Prob}\,(A(i, j) | C(i, j)) \\ &= \text{Prob}\,(\bar{C}) + \binom{n}{2} 2^{-r} < n^{-0.15} + O\left(\frac{1}{n}\right) < n^{-1/7}. \end{aligned}$$

This proves Theorem 1.2. $\square$

REFERENCES

[1] L. BABAI AND L. KUČERA, *Canonical labelling of graphs in linear average time*, 20th Annual IEEE Symp. on Foundations of Comp. Sci. (Puerto Rico) 1979, pp. 39–46.

[2] B. BOLLOBÁS, *Degree sequences of random graphs*, Aarhus University, 1978 preprint.

[3] P. ERDŐS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Akadémiai Kiadó, Budapest, 1974.

[4] P. ERDŐS AND R. J. WILSON, *On the chromatic index of almost all graphs*, J. Comb. Theory—B, 23 (1977), pp. 255–257.

[5] W. FELLER, *An Introduction to Probability Theory and its Applications*, Vol. 1, 3rd ed., John Wiley, New York, 1968.

[6] R. M. KARP, *The fast approximate solution of hard combinatorial problems*, Proc. 6th South-Eastern Conf. Combinatorics, Graph Theory and Computing (Florida Atlantic U. 1975), pp. 15–31.

[7] ———, *Probabilistic analysis of a canonical numbering algorithm for graphs*, Proc. Symposia in Pure Math. vol. 34, American Mathematical Society, Providence, RI, 1979, pp. 365–378.

[8] R. J. LIPTON, *The beacon set approach to graph isomorphism*, Yale University, 1978, preprint.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

SPACE LOWER BOUNDS FOR MAZE THREADABILITY ON RESTRICTED MACHINES*

STEPHEN A. COOK† AND CHARLES W. RACKOFF†

Abstract. A restricted model of a Turing machine called a JAG (Jumping Automaton for Graphs) is introduced for solving the maze threadability problem (determining whether there is a path joining two distinguished nodes in an input graph). A JAG accesses its input graph by moving pebbles from a limited supply along the edges of the graph under a finite state control, and detecting when two pebbles coincide. It can also cause one pebble to jump to another. We prove that for every N there is a JAG which can determine threadability of an arbitrary N node input graph in storage O $((\log N)^2)$, where the storage of a JAG with P pebbles and Q states is defined to be $P \log N + \log Q$. Further, we prove that any JAG which determines threadability requires storage $\Omega((\log N)^2/\log \log N)$. Finally, we prove that even when the inputs are restricted to undirected graphs (with no bound on the number of nodes), no single JAG can determine threadability.

Key words. space, space lower bounds, maze threadability, pebble automata

1. Introduction. The question of whether nondeterminism adds power to a space bounded Turing machine was apparently first posed by Kuroda [1] in 1964, who asked whether deterministic and non-deterministic linear bounded automata are equivalent. If we use the standard notation DSPACE $(L(n))$ (respectively NSPACE $(L(n))$ for the class of sets of strings accepted in deterministic (respectively nondeterministic) space $L(n)$ by a Turing machine, then Kuroda's question is equivalent to asking whether DSPACE (n) = NSPACE (n). In 1969, Savitch showed in his Ph.D. thesis (and later published in [2]) that NSPACE $(L(n)) \subseteq$ DSPACE $(L(n)^2)$ for "space constructible" functions $L(n) \geqq \log n$, and also he used a padding argument to show that if DSPACE $(\log n)$ = NSPACE $(\log n)$, then DSPACE $(L(n))$ = NSPACE $(L(n))$ for any space constructible $L(n)$ (including $L(n) = n$). Further, he proved that a certain interesting set, the "Threadable Mazes", is (using more recent terminology) log space complete for NSPACE $(\log n)$. This means that this set is in NSPACE $(\log n)$, and if it is also in DSPACE $(\log n)$, then DSPACE $(L(n))$ = NSPACE $(L(n))$ for all reasonable $L(n) \geqq \log n$.

The purpose of this paper is to give evidence that the set of Threadable Mazes is not in DSPACE $(\log n)$, by showing that a restricted model of a log space Turing machine cannot recognize the set.

DEFINITION 1.1. A *d-graph*, $d \geqq 2$, is a directed graph in which every node has outdegree at most d, and the edges leading out from each node have distinct labels from the set $\{1, 2, \cdots, d\}$. The d-graph is said to be *undirected* iff whenever there is an edge from a node x to a node y, there is also an edge from y to x. A d-maze $\mathcal{G}$ is a triple $\langle G, s, g\rangle$, where G is a d-graph, and s and g are nodes of G (the *start* and *goal nodes*, respectively). The maze $\mathcal{G}$ is *threadable* if there is a directed path from s to g.

In § 2, we introduce a restricted model of a Turing machine, called a d-JAG (Jumping Automaton for Graphs), for recognizing threadable mazes. A d-JAG consists of a deterministic control (which one can think of as a Turing machine control with space bounded work tapes, but actually it is more general than this, being an arbitrary finite state control) which controls a finite set of pebbles which move on an input d-maze. In one move, the machine can move a pebble along an edge, or jump it to

* Received by the editors March 19, 1978, and in revised form March 27, 1979.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7.

another pebble. We show (theorem 2.4) the model is powerful enough to execute an adaptation of Savitch's algorithm for recognizing threadable mazes in deterministic space $(\log n)^2$.

To make a d-JAG equivalent in maze threading ability to a deterministic log space Turing machine, it suffices to assume the input nodes are numbered in some arbitrary way from 1 to N (where N is the number of nodes), and to give the d-JAG the ability to move any one of its heads from node i to node $i+1$ (where node $N+1$ is node 1). In this case, jumps are not necessary. The result is essentially *Savitch's Maze Recognizing Automaton* [3]. Unfortunately, we are unable to prove that this more general machine cannot determine whether an arbitrary input d-maze is threadable.

A model similar to d-JAG's is described by Blum and Sakoda [4]. They show that some finite automaton with 4 pebbles can search any two dimensional maze (and hence could check whether such a maze is threadable), and they prove that no finite collection of finite automata is capable of searching every finite 3-dimensional maze. However, our d-JAG's are formally more powerful than the Blum-Sakoda finite automata with pebbles, because d-JAG's can cause pebbles to jump. This jumping ability substantially complicates our lower bound arguments.

Our d-JAG's are also like Tarjan's reference machines (see [5]) restricted so that they can search but not modify the data structure they are working on. It has been suggested that our lower bounds might have relevance to search algorithms for data structures.

One of the inspirations for this paper was a seminar talk [6] given by Michael Rabin in 1967, attended by the first author. Rabin's talk concerned a finite automaton moving along the edges of a graph. The automaton can drop a pebble on a node, as well as detect and pick up pebbles. Rabin outlined an argument that no such machine can completely search an arbitrary undirected regular graph of degree four. The informal argument given in this paper early in § 4 uses a different set of graphs than Rabin's to defeat the machine, and a somewhat different kind of automaton, but our argument certainly owes something to Rabin's talk. Our d-JAG (without jumps) differs from Rabin's automaton in that our automaton itself has no location on the graph, but each of its pebbles is allowed to move along edges under the direction of the finite state control. This difference seems slight, but it is much clearer how to generalize the argument for this machine to the case in which pebbles can jump (Theorem 4.13), which is the main result of § 4.

In § 3 we tackle the simpler problem of showing no d-JAG can determine whether an arbitrary (directed) d-maze is threadable. The fact that the edges are directed (so that a pebble cannot easily "back up") makes this result easier than the undirected case in § 4, but in fact we are able to prove a much better lower bound on the number of states and pebbles required to determine threadability (Theorem 3.1). Under a reasonable definition of storage (Definition 2.3) used by a d-JAG, Corollary 3.3 states that a d-JAG requires storage $c(\log N)^2/\log\log N$ to determine threadability of N-node d-mazes. This lower bound is close to Savitch's upper bound of $O(\log N)^2$ (which is also our upper bound for d-JAG's given in Theorem 2.4). It might seem that Corollary 3.3 comes close to establishing a space lower bound for Turing machines which recognize the threadable d-mazes since the d-JAG's in the corollary are allowed to use a Turing machine work tape with space $c(\log n)^2/\log\log n$. However, the restriction on the way a d-JAG (as opposed to a Turing machine) accesses its input d-maze seems crucial. In particular, a Turing machine has no trouble in backing a "pebble" against a directed edge. When this ability is allowed in § 4 (by restricting the input mazes to be undirected) the bound on the number of states and pebbles required to determine threadability is weakened to the point where it cannot be translated into an interesting general lower bound on work tape space for a Turing machine.

It is interesting to note that the kind of graphs used to defeat machines in § 3 is completely different from the kind in § 4. The graphs in § 3 are long, skinny, directed binary trees, with the start node at the root and the goal node hidden at one of the leaves. If the d-JAG were allowed to back up along edges, it is easy to see that a simple backtracking algorithm with two pebbles would suffice to search all binary trees. Hence no tree with undirected edges could defeat a d-JAG. The graphs in section 4 are completely homogeneous (transitive), and have a property which prevents a d-JAG from counting by moving pebbles. If a pebble is moved in any single direction repeatedly, it quickly winds up where it started from.

2. An Upper Bound for *d*-JAG's.

DEFINITION 2.1. A *d-Jumping Automaton for Graphs* (d-JAG) J is a system consisting of the following: A finite set of *states* with a distinguished start state q_0 and accepting state q_a, a positive integer P and a set of P objects called *pebbles*, which are numbered 1 through P, and a (deterministic) transition function δ which controls the behavior of J as described below. The input to J is a d-maze $\mathcal{G} = \langle G, s, g \rangle$. An *instantaneous description* (id) of J on input $\mathcal{G}$ is specified by a state q and an assignment of a node of G to each of the P pebbles. The next move of J for such an instantaneous description is specified by the transition function δ, and depends on (a) the state q, and (b) the coincidence partition $\mathcal{S}$ of the pebbles defined by two pebbles being in the same block of $\mathcal{S}$ if they lie on the same node. A move consists of assuming a new state after doing one of the following: (a) Move some specified pebble i along a specified edge $j \in \{1, 2, \cdots, d\}$. (If there is no edge labeled j leading out of the node on which pebble i lies, this move has no effect.) (b) Jump a specified pebble i to the node occupied by some specified pebble j, leaving pebble j alone. Any sequence (finite or infinite) of id's of J on an input $\mathcal{G}$ which results from consecutive legal moves of J is called a *computation* of J with input $\mathcal{G}$. The *initial* id of J with input $\mathcal{G}$ has state q_0, pebble 1 on the goal node g, and all other pebbles on the start node s. We say *J accepts $\mathcal{G}$* if some computation of J with input $\mathcal{G}$ which starts with the initial id ends in the accepting state q_a.

We could modify the above definition to allow the move to depend on which pebbles occupy node s and which occupy node g, and allow as possible moves "jump pebble i to node s" and "jump pebble i to node g". However, such a modified automaton with P pebbles can be obviously simulated by the present d-JAG with $P+2$ pebbles, where pebble $P+1$ sits permanently on node s and pebble $P+2$ is initially jumped to pebble 1 on node g, where it sits permanently.

Another possible modification is to allow the machine to know whether any pebble i is on a "leaf"; that is, a node with no edges leading out. Again, this more powerful machine can be simulated by a d-JAG with $2P$ pebbles. Each pebble of the leaf detecting machine is simulated by two pebbles which stay together. At each move, the two pebbles can determine if they are scanning a leaf by moving one along each out edge in turn, and checking whether the two still coincide. If for some edge they do not coincide the node is not a leaf, and the moved pebble is jumped back to its partner. (In fact, $P+1$ pebbles suffice for the simulation.)

DEFINITION 2.2. A d-JAG is *valid for N* if it accepts an arbitrary d-maze $\mathcal{G}$ of N or fewer nodes if and only if $\mathcal{G}$ is threadable.

It follows from results below that for $d \geqq 2$, no d-JAG J is valid for all N. To heuristically relate the number of pebbles and states of J to space on a Turing machine, we provide the following:

DEFINITION 2.3. The storage S used by a d-JAG with P pebbles and Q states on an input with N nodes is

$$S = P \log N + \log Q.$$

(All logarithms in this paper have base 2.)

Perhaps the best way to motivate this definition of storage is to introduce a new device, called a *Turing d*-JAG, which accesses its input graph like a d-JAG but does its computation like a Turing machine. A Turing d-JAG T has a finite state control which controls a read-write work tape like a Turing machine, and in addition T has a special *command tape* and a *query tape*. To move pebble i along edge j T writes $M * \bar{\imath} * \bar{\jmath}$ on its command tape, and to jump pebble i to pebble j T writes $J * \bar{\imath} * \bar{\jmath}$ on its command tape. (Here $\bar{\imath}$ and $\bar{\jmath}$ are the binary notations for i and j.) T can discover whether pebbles i and j coincide by writing $\bar{\imath} * \bar{\jmath}$ on its query tape, whereupon the next state will depend on whether the answer is yes or no. The storage $S_T(N)$ used by T is defined to be the maximum over all N-node input d-mazes $\mathcal{G}$ of $P \log N + L$, where P is the number of distinct pebbles queried during the computation with input $\mathcal{G}$, and L is the number of distinct work tape squares scanned during the computation. Then for each N, there is an ordinary d-JAG J_N whose storage according to Definition 2.3 is $O(S_T(N))$ which correctly simulates T on all N-node input graphs. (The states of J_N are pairs consisting of a state of T together with any possible tape configuration of T with an N-node input). On the other hand, an ordinary off-line Turing machine M can simulate T in storage $L(n) = S_T(n)$ simply by keeping track of the current node being visited by each pebble i, $1 \leqq i \leqq P$. (Each node can be identified by a string of length at most $\log n$.)

The reverse simulations cannot always be carried out. But the above argument does show that the storage lower bound of $c(\log N)^2/\log\log N$ given in Corollary 3.3. also applies to Turing d-JAG's, as well as to ordinary Turing machines which simulate Turing d-JAG algorithms in the manner indicated. Conversely, Theorem 2.4 below shows that d-JAG's can do as well at recognizing threadability as the best known Turing machine algorithm when storage is defined as in 2.3. In fact, the d-JAG algorithm given in the proof of the theorem is sufficiently uniform that it could be realized by a single Turing d-JAG with the same storage $(\log N)^2$, but we will omit this argument.

THEOREM 2.4. *For each N there is a d-JAG* $\hat{J}_N$ *valid for N with* O $(\log N)$ *pebbles and* $O(N^4)$ *states. Thus* $\hat{J}_N$ *has storage* $O((\log N)^2)$.

The proof is motivated by Savitch's algorithm [2] which simulates a nondeterministic $L(n)$ space bounded Turing machine by a deterministic $(L(n))^2$ space bounded machine. This algorithm shows how the threadable d-mazes can be recognized deterministically in space $(\log n)^2$. The idea is to define a recursive procedure Path (x, y, k) which determines whether there is a path of length k or less from node x to node y. This is done by calling, for each node z, Path $(x, z, k/2)$ and Path $(z, y, k/2)$ and returning yes if the both answers are yes for some z.

A d-JAG cannot cycle through all z as above, because it can only reach nodes accessible from either s or g. However, a d-JAG can succeed with a modified algorithm. Naively, the idea is to argue by induction on k that some d-JAG with $k+2$ pebbles can successively place pebble 1 on all nodes within a distance 2^k from the node s, no matter what the input d-maze. For $k = 0$ this can be done by the instructions: jump pebble 1 to pebble 2, move pebble 1 along edge 1, jump pebble 1 to pebble 2, move pebble 1 along edge 2, jump pebble 1 to pebble 2, $\cdots$, move pebble 1 along edge d. The induction step would be as follows: Assume J_k is a d-JAG with $k+2$ pebbles satisfying the induction hypothesis. To construct J_{k+1} it is necessary to have a new pebble $r = k+3$, and three slightly modified copies of J_k: call them J_k^1, J_k^2, and J_k^3. J_k^1 is the same as J_k, except each time pebble 1 scans a new node, all pebbles (including pebble r) except pebble 2, are jumped to pebble 1 and control is passed to the initial state of J_k^2. Machine J_k^2 acts like J_k, except it bases its operations at pebble r instead of pebble 2 at node s, and thus moves pebble 1 to all nodes within a distance 2^k from the node scanned by pebble r. After J_k^2 is finished, all pebbles except pebble r are jumped back to pebble 2 on node s, and control is passed to the initial state of J_k^3. Machine J_k^3 simulates J_k until pebbles 1 and r coincide (at which point all pebbles should be restored to where they were then J_k^2 took over), and control is now passed to J_k^1, which resumes pushing pebble 1 to new nodes. Hence pebble r eventually reaches all nodes within a distance 2^k from node s, and pebble 1 reaches all nodes within a distance 2^k from pebble r, and hence all nodes within a distance 2^{k+1} from s.

The above argument works when every node of the input d-maze has at most one path leading to it from s. However, if there were multiple paths to some node, then for some k, J_k^1 would cause pebble 1 (and hence pebble r) to visit that node twice, but J_k^3 would always restore the pebbles to the configuration of the first visit, and hence J_{k+1} would loop.

To overcome this difficulty, the induction hypothesis must be strengthened to assume J_k has a distinguished state q_d with the property that no node is ever visited twice by pebble 1 when J_k is in state q_d, and further every node at a distance 2^k from s is visited by pebble 1 exactly once with J_k in state q_d. We further assume J_k has $3k+3$ pebbles, that all nodes within a distance 2^k from s are visited by pebble 1 at least once, and that J_k enters a distinguished state q_f after completing its computation. More generally, we want the d-JAG J_k (i, q_d, q_f) to have all these properties except the start node s is replaced in the above specifications by the node scanned by some given pebble $i \geqq 3k+3$. We assume $J_k(i, q_d, q_f)$ starts with pebbles 1to $3k+2$ at pebble i, and pebble i is never moved during the computation.

The construction of $J_0(i, q_d, q_f)$ with pebbles 1, 2, and i is straightforward and will be left to the reader.

Assume $J_k(i, q_d, q_f)$ has been constructed to satisfy the induction hypothesis. We construct $J_{k+1}(i, q_d, q_f)$ from modified copies $J_k^1(p_i^1, q_d^1, q_f^1), \cdots, J_k^8(p_i^8, q_d^8, q_f^8)$ of J_k. Also, J_{k+1} has three new pebbles, r_1, r_2, and r_3 (which are numbered $3k+3$, $3k+4$, and $3k+5$). The idea is that control starts with J_k^1, and each time q_d^1 is entered, the pebbles (including r_1) are jumped to pebble 1, and control passes to J_k^2, which bases its operation at r_1. Each time q_d^2 is entered, r_2 is jumped to pebble 1 to save that position, and copies J_k^3 and J_k^4 start the whole process from scratch, checking to see if the node scanned by r_2 has been previously scanned by pebble 1 in state q_d. If not, then J_k^6 restores the pebbles for J_k^2, q_d is entered, and J_k^2 continues. If r_2 has been scanned before in state q_d, then J_k^7 restores the pebbles for J_k^2 and J_k^2 continues without assuming state q_d first.

Here is the detailed construction of $J_{k+1}(i, q_d, q_f)$. The instructions after each q_d^j and q_f^j tell what happens when these states are entered. Phrases in parentheses are comments. The phrase "jump all pebbles" means jump pebbles 1 to $3k+5$. That is, pebble i is never moved.

$J_k^1(i, q_d^1, q_f)$ *(Start J_k routine at pebble i).*

q_d^1: Jump all pebbles (including r_1) to pebble 1. Pass control to initial state of J_k^2.

q_f: Halt (final state of J_{k+1}).

$J_k^2(r_1, q_d^2, q_f^2)$ *(Base J_k routine at r_1).*

q_d^2: Jump pebble r_2 to pebble 1. Jump all pebbles except r_1 and r_2 to pebble i. Pass control to initial state of J_k^3.

q_f^2: Jump all pebbles except r_1 to pebble i. Pass control to initial state of J_k^8.

$J_k^3(i, q_d^3, q_f^3)$ (*Start checking whether r_2 scanned before*).

q_d^3: If pebbles 1 and r_1 coincide (check completed), jump all pebbles except r_2 to r_1 and pass control to the initial state of J_k^6. Otherwise, jump all pebbles except r_1 and r_2 to pebble 1 and pass control to initial state of J_k^4.

q_f^3: Halt (Error).

$J_k^4(r_3, q_d^4, q_f^4)$ (*Continue checking based at r_3*).

q_d^4: If pebbles 1 and r_2 coincide, jump all pebbles except r_2 to r_1 and pass control to initial state of J_k^7. Otherwise, continue with J_k^4.

q_f^4: Jump all pebbles except r_1, r_2, r_3 to pebble i. Pass control to initial state of J_k^5.

$J_k^5(i, q_d^5, q_f^5)$ (*Restore pebbles to continue with J_k^3*).

q_d^5: If pebbles 1 and r_3 coincide, continue with J_k^3 after q_d^3. Otherwise, continue with J_k^5.

q_f^5: Halt (Error).

$J_k^6(r_1, q_d^6, q_f^6)$ (*r_2 not scanned before: Restore pebbles to continue with J_k^2*).

q_d^6: If pebbles 1 and r_2 coincide, enter state q_d (of J_{k+1}) and then continue with J_k^2 after q_d^2. Otherwise, continue with J_k^6.

q_f^6: Halt (Error).

$J_k^7(r_1, q_d^7, q_f^7)$ (*r_2 scanned before: Restore pebbles to continue with J_k^2*).

q_d^7: If pebbles 1 and r_2 coincide, continue with J_k^2 after q_d^2. Otherwise continue with J_k^7.

q_f^7: Halt (Error).

$J_k^8(i, q_d^8, q_f^8)$ (*Restore pebbles to continue with J_k^1*).

q_d^8: If pebbles 1 and r_1 coincide, continue with J_k^1 after q_d^1. Otherwise continue with J_k^8.

q_f^8: Halt (Error).

Notice that if $S(k)$ is the number of states in J_k, then $S(k+1) \leqq 8S(k) + O(k)$, so $S(k) = O(9^k)$. Further, since $J_k(3k+3, q_d, q_f)$ visits all nodes within a distance 2^k from s, a slight modification of J_k can detect whether or not an arbitrary input maze of 2^k nodes or less is threadable. Since J_k has $3k+3$ pebbles, and J_k is valid for 2^k, we can take $\hat{J}_N = J_{\lceil \log N \rceil}$ to satisfy Theorem 2.4. □

Note that our d-JAG's in the above theorem can be made to enter a reject state when the input maze is not threadable, as well as entering the accept state q_a when the input maze is threadable. Our definition of *valid for N* did not require a reject state, since we wanted a weaker definition for stronger lower bound results.

3. A general lower bound for d-JAG's. The purpose of this section is to prove the following result.

THEOREM 3.1. *There is a constant c such that for every 2-JAG J with P pebbles and Q states there is a 2-graph G with fewer than $(cP^2(\log P + \log Q))^P$ nodes such that G is a directed binary tree with edges from each node to its sons, and when J starts with all pebbles on the root, there is some leaf which no pebble ever visits during the computation.*

COROLLARY 3.2. *There is a constant c_1 such that if some d-JAG with P pebbles and Q states is valid for N, then $N \leqq (c_1P^2(P + \log Q))^P$.*

COROLLARY. 3.3. *Each d-JAG valid for N uses storage $S \geqq c_2(\log N)^2/\log\log N$ for some constant $c_2 > 0$ and all $N \geqq 4$.*

Let us first prove 3.3. from 3.2. The inequality in 3.2 gives $\log N \leqq P \log(c_1P^2(P + \log Q))$, or

$$P \geqq \frac{\log N}{\log c_1 + 2\log P + \log(P + \log Q)}. \tag{3.4}$$

If $P \geqq \log N$ or $\log Q \geqq (\log N)^2$, then 3.3. follows directly from the Definition 2.3 of S, with $c_2 = 1$. If $P \leqq \log N$ and $\log Q \leqq (\log \mathrm{N})^2$, then 3.3 follows by substituting the right side of 3.4 for P in 2.3. □

The Corollary 3.2 is proved from Theorem 3.1 as follows. Given a d-JAG J with P pebbles and Q states, let J' be a 2-JAG which simulates J as follows. First, all instructions of the form "move pebble i along edge j" are deleted in J', for $j > 2$. Second, J', with all pebbles initially on one node, simulates J when all pebbles except pebble 1 are on one node, and pebble 1 is on an inaccessible leaf l (node with no outedges). As long as at least one pebble remains on the leaf l, pebbles may be jumped to and from leaf l, but, because the leaf l is inaccessible, no edge move will land a pebble on leaf l, and of course no edge move will remove a pebble from leaf l. The JAG J' keeps track of which set of pebbles would be on the leaf l in the computation of J, and makes the move J would make. So J' needs $2^P Q$ states and P pebbles.

Let G be the 2-graph for J' whose existence is asserted in the theorem, and let $\mathcal{G} = \langle G, s, g \rangle$ be the 2-maze whose start node s is the root of G and whose goal node g is the leaf which no pebble of J' ever visits. Then J, with pebble 1 initially on g and the other pebbles initially on s, makes a sequence of moves identical to the moves of J' with all pebbles initially on s. In particular, no pebble during the computation of J ever moves along the edge e from the father of g to g. Thus, if $\mathcal{G}'$ is the 2-maze which results from $\mathcal{G}$ by removing the edge e, then the computations of J on $\mathcal{G}$ and $\mathcal{G}'$ are identical, and yet $\mathcal{G}$ is threadable but $\mathcal{G}'$ is not. Therefore J is not N-valid, where N is the number of nodes in G. Corollary 3.2 follows. □

To prove Theorem 3.1, let J be any 2-JAG with P pebbles and Q states. To construct the 2-graph G, we construct successively a sequence $G_0, G_1, \cdots, G_{p-1} = G$ of 2 graphs, such that each G_k is a directed binary tree with root s_k and a distinguished leaf g_k. We define $\hat{G}_k$ to be the infinite binary tree constructed from copies of G_k as follows (see Fig. 1). The root of $\hat{G}_k$ is a node $\hat{s}_k$, and the sons of $\hat{s}_k$ are the s_k and s'_k of two copies of G_k. In general, the sons of each node of each copy of G_k in $\hat{G}_k$ are the same as in G_k, except each distinguished node g_k has as sons the roots of two new copies of G_k.

A *pruning* $\hat{\hat{G}}_k$ of $\hat{G}_k$ is a finite subtree of $\hat{G}_k$ obtained by selecting certain copies of g_k in $\hat{G}_k$ and deleting all proper descendants of these g_k.

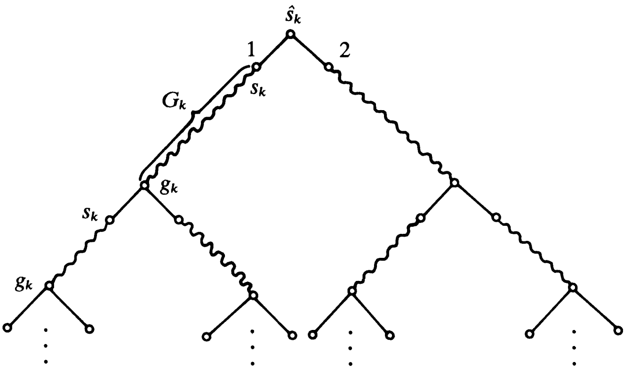


FIG. 1 $\hat{G}_k$

The graphs $G_0, G_1, \cdots$ are successively more complex, and are designed to cause successively more difficulty for the machine J to send pebbles from the root s_k to the leaf g_k of G_k. For example, G_0 has the property that if $\hat{\hat{G}}_0$ is any pruning of $\hat{G}_0$, then for any initial configuration of the P pebbles on distinct nodes of $\hat{\hat{G}}_0$ and any initial state, no pebble will ever move from s_0 to g_0 of a copy of G_0, as long as no two pebbles come together (either by an edge move or a jump). Each G_k has the same property (stated precisely in 3.5 below), except a certain degree of interaction among the pebbles is allowed. The amount of interaction (to be defined below as the number of empty "blocks") is bounded by the index k.

In order to formulate the property which G_k must satisfy we need to define notions concerning partitions of pebbles. (We will abuse the definition of partition slightly to allow some blocks to be empty.) A partition $\{B_1, \cdots, B_l\}$ of (pebbles) $\{1, \cdots, P\}$ into blocks $B_1, \cdots, B_l$ is *admissible* at an id if any two pebbles on the same node are in the same block. Let $C_0, C_1, \cdots$ be a computation of J and let $\{B_1 \cdots, B_l\}$ be a pebble partition admissible at C_0. The *continuation* of the blocks $B_1, \cdots, B_l$ at C_t with respect to the computation is defined by induction on t. For $t = 0$, the continuation of each B_i is B_i. In general, if the move from C_t to C_{t+1} causes some pebble u to coincide with some pebble v either by an edge move or a jump, then to obtain the continuation at C_{t+1} from the continuation at C_t, simply delete pebble u from its block and add it to the block of pebble v, and keep all other blocks the same. If no pebble is caused to hit another by this move, then the continuations at C_t and C_{t+1} are the same. Notice that at each step C_t of the computation the continuations of blocks $B_1, \cdots, B_l$ always form an admissible partition of $\{1, \cdots, P\}$, although some blocks may be empty. If a partition $\{B_1, \cdots, B_l\}$ has been specified at the beginning of a computation, and we refer to the block B_i at some point in the computation, we shall always mean the *continuation* of B_i at that point.

We shall construct G_k to have the following property.

3.5. *Induction hypothesis* $H(k)$. Let $\hat{\hat{G}}_k$ be any pruning of $\hat{G}_k$, and consider any initial configuration of the P pebbles on $\hat{\hat{G}}_k$, any admissible partition of these pebbles into blocks $B_1, \cdots, B_l$, $l \geqq P-k$, and any initial state. If all pebbles in some block B_i lie initially on or above the node s_k of some copy of G_k in $\hat{\hat{G}}_k$, then no pebble in (the continuation of) B_i will visit the g_k of that copy of G_k during the computation.with that initial state and pebble configuration, as long as at least $P-k$ of the blocks B_j remain nonempty.

Basis: $k = 0$. Let T be the complete infinite labeled directed binary tree with root r. For each pebble i and each state σ, consider the computation of J which starts in state σ with pebble i on the root r of T, and the other pebbles scattered on T in such a way that no two pebbles meet before a jump takes place. Let us terminate the computation as soon as any jump takes place. Then pebble i describes a path $\pi(i, \sigma)$ down from node r, which may be finite or infinite. Since there are P choices for i and Q choices for σ, there are at most PQ such paths. On the other hand, there are exactly 2^d binary paths of length d, so if $d > \log (PQ)$ there must be some path π of length d which differs from the initial segment of length d of each $\pi(i, \sigma)$ (although some $\pi(i, \sigma)$ could be a proper initial segment of π).

To construct G_0, let s_0 be the initial node (root) of π, let g_0 be the final node, and add a second labeled edge to every node of π except g_0, so that G_0 becomes a finite labeled directed binary tree with root s_0, such that every node has either two or zero edges leading out. Fig. 2 gives an example of π and the corresponding G_0.

To see that G_0 satisfies the induction hypothesis $H(0)$, note that the condition that $P-k$ blocks remain nonempty (with $k = 0$) ensures that no two pebbles will meet and no jumps will take place during the relevant part of the computation. The hypothesis $H(0)$ simply states that no pebble can find its way from s_0 to g_0, which is clear by our construction of G_0. Also note that if some pebble visits a leaf of $\hat{\hat{G}}_0$, it must remain trapped there. (Our definition of JAG does not allow J to detect when a pebble is on a leaf, but see the remark in the paragraph before Definition 2.2).

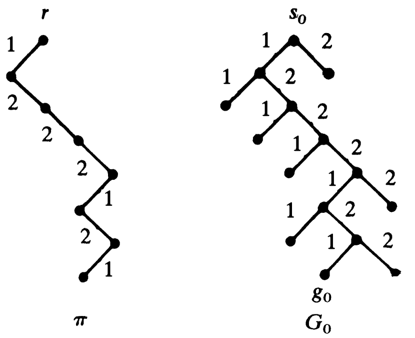


FIG. 2

If we take $d = \lceil \log P \rceil + \lceil \log Q \rceil + 1$, then

$$|G_0| = 2(\lceil \log P \rceil + \lceil \log Q \rceil) + 3, \tag{3.6}$$

where $|G_0|$ is the number of nodes in G_0.

Induction Step: $H(k) \Rightarrow H(k+1)$.

Suppose G_k has been constructed to satisfy $H(k)$.

LEMMA 3.7. *Suppose in the statement* 3.5 *of* $H(k)$ *we weaken the assumption that at least* $P-k$ *blocks remain nonempty to at least* $P-k-1$ *blocks remain nonempty. Then the following weaker conclusion holds. If all pebbles in some block* B_i *lie initially on or above the node* s_k *of some copy of* G_k *in* $\hat{G}_k$, *and if there is a first time* t_1 *at which some pebble j of (the continuation of)* B_i *visits the node* g_k *of that* G_k, *then all pebbles of* B_i *will lie in that* G_k *at time* t_1.

Proof. Assume all hypotheses in the lemma. We claim that for each node ν on the path from s_k to g_k there is a time t_ν such that if B_ν is the set of pebbles on ν at time t_ν, then j at time t_1 is in the continuation of B_ν. The claim is proved by induction on the distance δ of ν from g_k. If $\delta = 0$, then $\nu = g_k$ and $t_\nu = t_1$. If $\delta > 0$, let μ be the successor of ν which is a distance $\delta - 1$ from g_k, so t_μ exists by the induction hypothesis. Then t_ν is defined by the condition that $t_\nu + 1$ is the first time t that μ is continuously occupied between times t and t_μ. Thus the move executed at time t_ν is to advance some pebble along the edge from ν to μ. Also B_μ at time t_μ is a subset of the continuation of B_ν, and since pebble j at time t_1 is in the continuation of B_μ, it is also in the continuation of B_ν. This proves the claim.

Now let $t_0 = t_{s_k}$ and $B = B_{s_k}$. Note that $B \subseteq C$, where C is the continuation of block B_i at time t_0. If we split C into B and $B' = C - B$, then the continuation of B' at time t_1 must be empty, for otherwise there would be $P-k$ nonempty blocks at time t_1, and the hypothesis $H(k)$ applied to the computation from t_0 to t_1 under the split partition at time t_0 would preclude the possibility of pebble j reaching node g_k at time t_1. But the continuation of B_i at time t_1 is the union of the continuations of B and B' at time t_1, and so it is equal to the continuation of B at time t_1. It is clear that this continuation lies entirely on the indicated copy of G_k, since pebble j is the first pebble of B_i to reach g_k, and any jumps of pebbles from B_i would cause these pebbles to leave the block. □

By Lemma 3.7, a single block of pebbles describes a unique path down $\hat{G}_k$ for any single computation, as long as at least $P-k-1$ blocks remain nonempty. To construct G_{k+1}, we find a path in $\hat{G}_k$ distinct from all possible such paths, and turn it into a tree in a way similar to the construction of G_0. We think of $\hat{G}_k$ as the complete binary *super tree*, with *super edges* which are copies of G_k. Thus each block traces a *superpath* in $\hat{G}_k$, which includes precisely those super edges traversed completely from s_k to g_k by the block.

The *length* of the super path is the number of its super edges. We wish to get an upper bound on the cardinality of $F(d)$, the set of initial segments of length d of super paths described by any block of pebbles in any pruning $\hat{\hat{G}}_k$ of $\hat{G}_k$ in a computation with any initial state and pebble configuration in $\hat{\hat{G}}_k$, assuming first that at least $P-k-1$ blocks remain nonempty during the computation, and second that no block traces a super path of length exceeding $2d-1$ during the computation. Note that for this purpose, we can assume that the initial pebble partition is the coincidence partition, since the path of any block in a coarser partition will coincide with the path of some block in the coincidence partition (recall that a super edge does not count in the path of a block unless it is traced completely from s_k to g_k).

Note that for each super edge E in the path of a block B_i, pebbles of B_i might visit nodes in the four super edges adjacent to E (brother, father, and two sons) as well as nodes in E itself. In fact, if the path has length l, then the number of potential super edges which pebbles of B_i might visit during the computation is $2l+3$, counting the super edge containing the node on which B_i lies initially. (If B_i initially lies on the node s_k of a super edge, the number is $2l+2$). We call this set of super edges the *extended path* of B_i.

To specify a path in $F(d)$, it suffices to specify the following:

(1) The initial state (Q possibilities).

(2) For each pebble i, the node relative to a super edge (copy of G_k) on which pebble i lies ($|G_k|^P$ possibilities).

(3) For each i and j, whether block B_i lies initially on the (future) extended path of block B_j, and if so, the super edge on which B_i lies relative to the extended path (see below) ($(2(2d-1)+4)^{P2}=(4d+2)^{P2}$ possibilities at most).

(4) For each i, the length of the path traced by block B_i and whether some pebble of B_i visits a node g_k which is a leaf of $\hat{\hat{G}}_k((4d)^P$ possibilities, including length zero).

(5) The block which determines the path in question (P possibilities).

In (3), the information specified is an integer m between 0 and $4d+1$ where $m=0$ means B_i does not lie on the extended path, and $m>0$ means B_i lies on edge E_m of the extended path, where we label the edges of the path traced by B_j by $E_2, E_4, \cdots, E_{4d-2}$ in the order traced, and let E_1 be the edge initially visited by B_j (E_1 is the father of E_2, unless B_j is initially on s_k). Also E_{2l+1} is the brother of E_{2l}, $1 \leqq l \leqq 2d-1$, and E_{4d}, E_{4d+1} are respectively the left and right sons of E_{4d-2}.

To see that the above information is sufficient to determine the moves in the computation, and hence the path of the block specified in (5), one can verify by induction on t that at each time t in the computation the following information is known:

(a) The state.

(b) For each pebble i, the node relative to a super-edge (copy of G_k) on which pebble i lies.

(c) For each pair of pebbles i and j, whether i and j lie in the same super-edge.

(d) Which pebbles are in each block.

(e) The relative position of any two pebbles in the same block.

Notice (b) and (c) together specify the coincidence partition of the pebbles, and hence (a), (b), and (c) specify the next move. That (a) and (b) are determined initially follows from (1) and (2). To determine (c) initially, note that any two pebbles in the same block initially scan the same square by assumption, and for two pebbles in different blocks one can tell by (3) whether or not they lie initially on the same super-edge. Trivially (d) and (e) are determined initially because the initial partition is the coincidence partition.

In general, at time $t+1$, (a), (b), and (e) are easily updated because the move at time t is known (by the induction hypothesis for (a), (b), and (c)), except some pebble i may lie at time t on a node g_k which is a leaf of $\hat{G}_k$, in which case of course pebble i will not move. However, it can be determined by the specification (4) whether pebble i is on a leaf g_k, because the length of the path of the block of pebble i to date is known by the induction hypothesis applied to (a)–(d). The only difficulty in updating (c) at time $t+1$ comes when the move at time t causes a pebble i in a block B to move from the node g_k of some super-edge to a new super-edge E'. We know by (e) the existence and position of any pebbles of block B which already lie on E'. If there are pebbles in a different block B' on E', then initially B' lay on the extended path of B (since it does at time $t+1$, and there cannot be a time when the path of B' joined that of B). Therefore by specification (3), and the fact that we know the paths of all pebbles in B and B' to date, we can determine that certain pebbles of B' are in fact on super-edge E' at time $t+1$. Finally, (d) can be updated at time $t+1$ because (b) and (c) are known at time $t+1$.

Thus an upper bound on the number of super paths in $F(d)$ is the product of the bounds given in (1) to (5), namely $A=Q|G_k|^P(4d+2)^{P^2}(4d)^P P$, where $|G_k|$ is the number of nodes in G_k. To insure the existence of a super path Π of length d which differs from every member of $F(d)$, we need only assume $2^d > A$, or $d > \log_2 A$. Hence it suffices to assume

$$d > \log P + \log Q + P \log |G_k| + (P^2+P)\log(4d+2). \tag{3.8}$$

Suppose now d is chosen so that (3.8) is satisfied, and Π is a super path of length d as above. Then G_{k+1} is formed from Π in the same way that G_0 was formed from Π. Namely, the initial node s_k of Π becomes s_{k+1} of G_{k+1}, the final g_k becomes g_{k+1}, and to every other node g_k we attach an out edge labeled opposite to the label on the existing out edge, and attach a copy of G_k to this new edge (see Fig. 3).

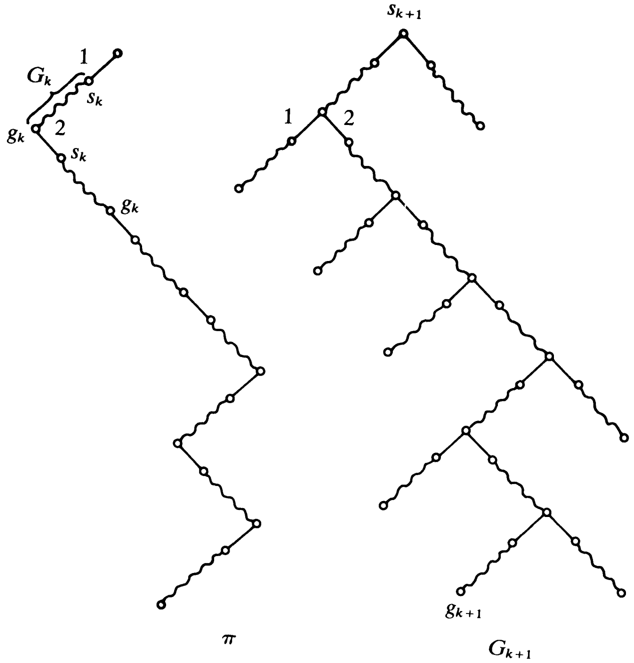


FIG. 3

Thus

$$|G_{k+1}| = 2d|G_k| + 1. \tag{3.9}$$

To see that G_{k+1} satisfies the hypothesis $H(k+1)$, note that any pruning $\hat{\hat{G}}_{k+1}$ of $\hat{G}_{k+1}$ is also a pruning of $\hat{G}_k$. We must show that as long as $P-k-1=P-(k+1)$ blocks remain nonempty, no block can trace a super edge in $\hat{\hat{G}}_{k+1}$. This is because right after the first such super edge is traced, no block has yet traced a path of $\hat{G}_k$ of length exceeding $2d-1$, so our assumption that Π is not in $F(d)$ would be violated.

It remains to estimate the number of nodes $|G_k|$ of G_k. We shall show by induction on k that for some constant c,

$$|G_k| \leqq (cP^2(\log P + \log Q))^{k+1}, \quad \text{for } 0 \leqq k < P, PQ \geqq 2. \tag{3.10}$$

For $k=0$, this follows directly from (3.6). In general, the inequality (3.8) is satisfied if we take

$$d = \lfloor aP^2(\log P + \log Q) \rfloor, \tag{3.11}$$

with $a = b \log c$, for some constant b independent of $c \geqq 2$, assuming (3.10) holds as an induction hypothesis. Thus (3.10) with $k+1$ substituted for k follows directly from (3.9), (3.10), and (3.11), provided we take $c \geqq 3a$.

Theorem 3.1 follows by setting $k = P-1$ in inequality (3.10). □

4. The undirected case. We are interested in showing that no 3-JAG is valid for the set of undirected 3-mazes. Actually, we shall first show that for any P there is a d such that no d-JAG with P pebbles accepts an arbitrary undirected d-maze iff it is threadable, and then show how d can in fact be chosen to be 3.

Our approach is as follows. For integers m and d, we define below a $2d$-graph $G_{m,d}$, and compute an upper bound on how many nodes a given $2d$-JAG J with P pebbles can visit. For m and d chosen appropriately, the pebbles of J cannot visit very many nodes of $G_{m,d}$, and using this result we can easily construct a $2d$-(undirected) maze on which J acts incorrectly.

DEFINITION 4.1 Let Z_m^d be the set of ordered d-tuples of integers from the set $Z_m = \{0, 1, \cdots, m-1\}$. When we use the operation $+$ (or $-$) between members of Z_m, we mean addition (or subtraction) componentwise, mode m. For $1 \leqq i \leqq d$, define $e_i \in Z_m^d$ to be that element having all 0's except for 1 in place i. Define the undirected $2d$-graph $G_{m,d}$ to have node set Z_m^d, and for every node x there is an arc labeled i from x to $x+e_i$ and an arc labeled $d+i$ from x to $x-e_i$, for $1 \leqq i \leqq d$.

Now let m and d be fixed positive integers (until stated otherwise) and let J be a fixed d-JAG with Q states and P pebbles. We wish to obtain a good upper bound on how many nodes the pebbles of J can visit when J is started (in an arbitrary way) on $G_{m,d}$.

Although technically J is defined as starting in a particular id (instantaneous description) on $G_{m,d}$, it is easy to generalize the notion of computation by allowing J to start in an arbitrary id $C_0 = (q, M)$ where q is a state and M (giving the position of the pebbles) maps $\{1, 2, \cdots, P\}$ into Z_m^d.

DEFINITION 4.2. We denote by COMP (q, M) the sequence $C_0, C_1, \cdots$ of id's that J goes through when it starts with C_0. Define DISPLAY (C_0) to be the sequence of state and pebble-partition pairs that the finite state control of J "sees" when J starts computing in id C_0. That is, if

$$\text{COMP }(C_0) - (q_0, M_0), (q_1, M_1), \cdots \quad \text{then}$$
$$\text{DISPLAY }(C_0) = (q_0, \mathscr{S}_0), (q_1, \mathscr{S}_1), \cdots$$

where $\mathcal{S}_i$ is the coincidence partition of $\{1, 2, \cdots, P\}$ determined by M_i; two pebbles are in the same block of $\mathcal{S}_i$ iff they are mapped onto the same point by M_i. [Note that we are using q_0 and $\mathcal{S}_0$ to represent an arbitrary state and partition here, not necessarily the initial ones.]

We now give an informal outline of how we compute an upper bound on the number of nodes visited by J. Along with the COMP and DISPLAY sequences, we will define the SUPERDISPLAY (henceforth abbreviated SD) sequence. At each step, the SD will give the state of the machine and a partition of the pebbles, such that two pebbles are in the same block of the partition if the information in the DISPLAY sequence so far allows one to deduce their relative displacement (their difference as members of Z_m^d). This partition will therefore be as coarse as or coarser than the coincidence partition in the display (where one puts two pebbles in the same block only if one can deduce that their relative displacement is $\bar{0}$). In addition, the SD will contain the relative displacement of every pair of pebbles in each of its blocks.

Whenever a pebble from a block B_1 in the SD at a certain time moves along an edge and encounters a pebble from another block B_2, the two blocks will be merged in the SD for the next instant, since we now know the relative displacements of all the pebbles in $B_1 \cup B_2$; we call this a *coalition*. If a pebble from B_1 jumps to a member of B_2, that pebble leaves B_1 and becomes part of B_2 in the next SD; in the case that this leaves B_1 empty, we also call this a *coalition*. In brief, a coalition occurs whenever the number of blocks in the SD partition decreases. The upper bound will proceed by induction on the number of coalitions. By definition, the number of coalitions occurring by time 0 is $P-n$, where n is the number of blocks in the initial coincidence partition. To gain an intuition behind this induction, note that as long as *no* coalition occurs (i.e. the pebbles don't interact at all) it is easy to compute an upper bound on the number of nodes covered. (Note that the notion of coalition implicit in the definition of "continuation" in § 3 is different from our definition here.)

Let C_0 be any id and let $\mathrm{COAL}_n\ (C_0)$ be the first time at which there are $P-n$ or fewer blocks in the SD for the computation starting with C_0; $\mathrm{COAL}_n\ (C_0) = \infty$ if there is no such time. That is, $\mathrm{COAL}_n\,(C_0)$ is the time of the n-th coalition. If C and C' are arbitrary id's, we say $C \equiv_n C'$ iff $\mathrm{COAL}_n\ (C) = \mathrm{COAL}_n\ (C') = t$ and DISPLAY (C) and DISPLAY (C') are identical up to and including time t. If $C \equiv_n C'$, then their DISPLAY sequences will in fact be equal up to (but excluding) the time when one of them has an $n+1$ coalition. It is easy to see that $\equiv_n$ is an equivalence relation.

Let R_n be the number of $\equiv_n$ equivalence classes. Let A_n be the maximum over all id's C of the number of distinct id's occurring in COMP (C) up to (but excluding) the id at the time $\mathrm{COAL}_{n+1}\ (C)$ (if it exists). We wish to compute an upper bound on A_{p-1}. More generally, we will get upper bounds on A_n and R_n by induction on n:

$$R_0 \leqq Q \cdot P^P \tag{1}$$

since there are at most P^P partitions on $\{1, 2, \cdots, P\}$.

$$A_n \leqq (1+P+mP!)R_n \quad \text{for } 0 \leqq n \leqq P. \tag{2}$$

Consider an initial piece of a computation with no $n+1$ coalition. After at most R_n steps the $\equiv_n$ class of the id repeats with periodicity at most R_n. This means that after at most R_n steps the pebble movements repeat with a periodicity of at most R_n. If there were no pebble jumps, the cyclic nature of G_m^d would yield that after at most R_n steps the id repeats with a periodicity of at most mR_n. Even with pebble jumps, however, we can conclude a periodicity in the id of at most $mP!\,R_n$, beginning after at most $(1+P)R_n$ steps.

$$R_{n+1} \leqq R_n(A_n \cdot P^P + 1) \cdot (QP^P) \quad \text{for } 0 \leqq n < P. \tag{3}$$

In order for $C \equiv_{n+1} C'$ it is sufficient that $C \equiv_n C'$ (R_n possibilities), that the displays at the moment (if it exists) of the $n+1$ coalition are the same (QP^P) possibilities, and that the moments of the $n+1$ coalitions are the same (we will see that there are A_nP^P+1 possibilities).

From this we can conclude that J visits at most $(mQ)^{c^P}$ nodes of $G_{m,d}$ for some constant c. Hence, if $d > c^P$ and m is large enough, J fails to visit all the nodes of $G_{m,d}$.

We leave as an interesting exercise to the reader to prove the fact that the above result is the best possible in the following sense: If $d = 2^{P-1}$ then there is a d-JAG J with no jumps with P pebbles such that for all m, if J starts on $G_{m,d}$ with all pebbles on one node, pebble 1 will eventually visit all nodes in the graph.

We now proceed with the formal development. Many details and proofs are omitted, however, and the reader is urged to consult [7] for a more complete presentation.

We first define the SUPERDISPLAY (SD) sequence carefully. Although the definition is long, the intuition (as described above) is straightforward.

DEFINITION 4.3. Let $C_0 = (q_0, M_0)$ be an id; let

$$\text{COMP}\,(C_0) = (q_0, M_0),\ (q_1, M_1), \cdots$$

and

$$\text{DISPLAY}\,(C_0) = (q_0, \mathcal{S}_0),\ (q_1, \mathcal{S}_1), \cdots.$$

We define a sequence $\text{SD}\,(C_0) = (q_0, (\tau_0, f_0)), (q_1, (\tau_1, f_1)), \cdots$. Each τ_i is a partition of the pebbles, and for each j_1, j_2 in a block of τ_i, $f_i(j_1, j_2)$ is defined and is a member of Z_m^d. A consequence of our definition will be that $f_i(j_1, j_2) = M_i(j_2) - M_i(j_1)$.

Let $\tau_0 = \mathcal{S}_0$ and f_0 (where defined) takes on the value $\bar{0}$ (the 0-tuple in Z_m^d).

(τ_{i+1}, f_{i+1}) will be defined from $(q_i, (\tau_i, f_i))$ and $\mathcal{S}_{i+1}$. From (τ_i, f_i) we can deduce $\mathcal{S}_i$, and therefore determine from $(q_i, (\tau_i, f_i))$ the move J will make at step $i+1$.

Case 1. J jumps pebble j_1 to pebble j_2 (assume $j_1 \neq j_2$). Then form τ_{i+1} by taking j_1 out of the block of τ_i it's in, and putting it in the block containing j_2 (which may be the same block). f_{i+1} has the same value as f_i (if any) on any pair of pebbles neither of which is j_1. For an arbitrary pebble j in the same block as j_2 and j_1, define $f_{i+1}(j_1, j) = -f_{i+1}(j, j_1) = f_i(j_2, j)$.

Case 2. J moves pebble j_1 by distance $e \in \{e_1, e_2, \cdots, e_d, -e_1, \cdots, -e_d\} \subseteq Z_m^d$. Firstly, define τ'_{i+1} to be the same as τ_i, and define f'_{i+1} to have the same value (if any) as f_i on any pair of pebbles neither of which is j_1; for arbitrary pebble j in the same block as j_1, define $f'_{i+1}(j, j_1) = -f'_{i+1}(j_1, j) = f_i(j, j_1) + e$. There are two subcases to consider to define τ_{i+1} and f_{i+1} from τ'_{i+1} and f'_{i+1}.

Case 2a. Pebble j_1 has "by surprise" hit another pebble j_2. That is, j_1 and j_2 are not in the same block of τ_i, but are in the same block of $\mathcal{S}_{i+1}$ (for some pebble j_2). Then form τ_{i+1} from τ_i by merging the blocks of τ_i containing j_1 and j_2. Extend f'_{i+1} to f_{i+1} as follows: if x_1 and x_2 are, respectively, in the same blocks of τ_i as j_1 and j_2, then define $f_{i+1}(x_1, x_2) = -f_{i+1}(x_2, x_1) = f'_{i+1}(x_1, j_1) + f'_{i+1}(j_2, x_2)$.

Case 2b. Otherwise. Then define $\tau_{i+1} = \tau'_{i+1} = \tau_i$ and $f_{i+1} = f'_{i+1}$.

NOTATION 4.4. Say that $\text{COMP}\,(q, M) = (q_0, M_0), (q_1, M_1) \cdots$, $\text{DISPLAY}\,(q, M) = (q_0, \mathcal{S}_0), (q_1, \mathcal{S}_1), \cdots$, and $\text{SD}\,(q, M) = (q_0, (\tau_0, f_0)), (q_1, (\tau_1, f_1)), \cdots$. Then for $i \geqq 0$, we define $\text{COMP}_i\,(q, M) = (q_i, M_i)$, $\text{DISPLAY}_i\,(q, M) = (q_i, \mathcal{S}_i)$, $\text{SD}_i\,(q, M) = (q_i, (\tau_i, f_i))$.

For $n \geqq 0$, let COAL_n, $\equiv_n$, A_n, and R_n be defined as in the informal outline.

The next lemma says that if two id's are $\equiv_n$, then they have the same display sequence up to but excluding the moment when one of them has an $n+1$ coalition.

LEMMA 4.5. *Say that* $(q, m) \equiv_n (q', M')$. Let $j = \mathrm{COAL}_n(q, M) = \mathrm{COAL}_n(q', M')$. *Say that* $k = \mathrm{COAL}_{n+1}(q, M) \leqq \mathrm{COAL}_{n+1}(q', M')$. *Then for all* i, $0 \leqq i < k$, *we have* $\mathrm{DISPLAY}_i(q, M) = \mathrm{DISPLAY}_i(q', M')$.

Proof. Certainly $\mathrm{DISPLAY}_i(q, M) = \mathrm{DISPLAY}_i(q', M')$ for all i, $0 \leqq i \leqq j$, and hence $\mathrm{SD}_i(q, M) = \mathrm{SD}_i(q', M')$ for all i, $0 \leqq i \leqq j$. Since for every i, $j < i < k$, neither of the SD's experience a coalition, SD_i is determined by SD_{i-1}. So $\mathrm{SD}_i(q, M) = \mathrm{SD}_i(q', M')$ and hence $\mathrm{DISPLAY}_i(q, M) = \mathrm{DISPLAY}_i(q', M')$) for all i, $j < i < k$. □

The next lemma shows that a periodicity in pebble movements implies a periodicity in the pebble positions.

LEMMA 4.6. *Let* α *be a finite sequence of pebble moves of* J. *Say that in a given computation, the sequence* α *is repeated consecutively* t *times, and the mapping of the pebbles after the* s-*th execution of* α *is given by* M_s *for* $0 \leqq s \leqq t$. *Then for every pebble* i *there exists an* $x_i \in Z_m^d$ *such that for all* s, $P \leqq s \leqq t - P!$, *we have* $M_{s+P!}(i) = M_s(i) + x_i$. *Hence,* $M_{P+mP!}(i) = M_P(i)$ *if* $P + mP! \leqq t$.

Proof outline. Let $i \in \{1, 2, \cdots, P\}$ be a pebble. One first shows that there exists a pebble j, numbers k_1, k_2, and displacements $y_1, y_2 \in Z_m$ such that for all $s \geqq 0$

(1) $M_{s+k_1}(i) = M_s(j) + y_1$;

(2) $M_{s+k_2}(i) = M_s(j) + y_2$;

(3) $0 \leqq k_1 < k_2 \leqq P$

(assuming that $s + k_1$ and $s + k_2$ are both $\leqq t$.)

Hence, $M_{s+k_2}(i) = M_{s+k_1}(i) + (y_2 - y_1)$. So we can say that whenever $P \leqq s < s + k_2 - k_1 \leqq t$, $M_{s+(k_2-k_1)}(i) = M_s(i) + (y_2 - y_1)$. So $M_{s+P!}(i) = M_s(i) + (y_2 - y_1) \cdot P!/(k_2 - k_1)$ if $P \leqq s \leqq s + P! \leqq t$. It suffices to let $x_i = (y_2 - y_1) \cdot P!/(k_2 - k_1)$. $M_{P+mP!}(i) = M_P(i) + m \cdot x_i = M_P(i)$. □

LEMMA 4.7.

$$A_n \leqq (1 + P + mP!)R_n; \quad \textit{for } 0 \leqq n < P.$$

Proof outline. $C_0, C_1, \cdots, C_l$ is a computation (or an initial part of a computation) not containing an $n+1$ coalition. Assume $l \geqq (1 + P + mP!)R_n$. For some $0 \leqq i < j \leqq R_n$, $C_i \equiv_n C_j$. A subtle point which must be checked here is that neither the computation $C_i, C_{i+1}, \cdots, C_l$ nor the computation $C_j, C_{j+1}, \cdots, C_l$ contains an $n+1$ coalition. Lemma 4.5 tells us therefore that $\mathrm{DISPLAY}_{i+k}(C_0) = \mathrm{DISPLAY}_{j+k}(C_0)$ for all $k \geqq 0$, $j + k \leqq l$. So if α is the sequence of pebble moves made from time i to time j, then α is repeated over and over until the end of the computation. Lemma 4.6 tells us that after at most $P + mP!$ occurrences of α, the pebble position starts repeating; the periodicity of DISPLAY implies that the state sequence repeats with each α, so the id starts repeating after at most $P + mP!$ occurrences of α. The number of id's from the beginning up to the end of an occurrence of α is $\leqq j - i \leqq R_n$. The number of id's before time i is $\leqq i \leqq R_n$. So the number of different id's in the computation is $\leqq R_n + (P + mP!)R_n = (1 + P + mP!)R_n$. □

LEMMA 4.8.

$$R_{n+1} \leqq R_n \cdot (A_n \cdot P^P + 1) \cdot (QP^P) \quad \textit{for } 0 \leqq n < P.$$

Proof. In order that $(q, M) \equiv_{n+1} (q', M')$, it is sufficient, by Lemma 4.5, that

(1) $(q, M) \equiv_n (q', M')$;

(2) $\mathrm{COAL}_{n+1}(q, M) = \mathrm{COAL}_{n+1}(q', M') = j$;

(3) if $j < \infty$, then

$$\mathrm{DISPLAY}_j(q, M) = \mathrm{DISPLAY}_j(q', M').$$

Therefore $R_{n+1} \leqq R_n \cdot (x+1) \cdot (QP^P)$ where x is the number of possible finite values for j. If an initial part of a computation doesn't contain an $n+1$ coalition, then the id starts repeating after at most A_n steps, and so the id-SD pair starts repeating after at most $A_n P^P$ steps. So an $n+1$ coalition cannot *ever* occur afterwards if the sequence has length more than $A_n P^P$. □

THEOREM 4.9. *There is a constant c such that for all P, m, d, $Q \in N$, if J is a d-JAG with P pebbles and Q states operating on $G_{m,d}$ (with its pebbles in an arbitrary starting configuration) then at most $(mQ)^{c^P}$ nodes of $G_{m,d}$ are visited by pebbles of J.*

Proof. In the informal outline it is argued that $R_0 \leqq QP^P$. This and Lemmas 4.7 and 4.8 imply that $A_{p-1} \leqq (mQ)^{e^P}$ for some constant e. This is the largest number of distinct id's in a computation, so the largest number of nodes visited by pebbles $\leqq P(mQ)^{e^P} \leqq (mQ)^{c^P}$ for some constant c. □

DEFINITION 4.10. For integers d, m, define the d-graph $G'_{m,d}$ to be the graph consisting of *two* unconnected copies of $G_{m,d}$.

COROLLARY 4.11. *There is a constant c such that for all P, m, d, $Q \in N$, if J is a d-JAG with P pebbles and Q states operating on $G'_{m,d}$ (with its pebbles in an arbitrary starting configuration), then at most $(mQ)^{c^P}$ nodes of $G'_{m,d}$ are visited by pebbles of J.*

Proof. This is not actually a corollary, but the proof of this statement is virtually identical to that of Theorem 4.9. □

THEOREM 4.12. *For every P there is a d such that no d-JAG with P pebbles accepts an arbitrary undirected d-maze $\mathscr{G}$ iff $\mathscr{G}$ is threadable.*

Proof. Let P be fixed; choose d such that $d > c^P$ where c is the constant of Corollary 4.11. Let J be a $2d$-JAG with P pebbles and Q states. Choose m larger than Q^{c^P}, so that $(mQ)^{c^P} < m^d$.

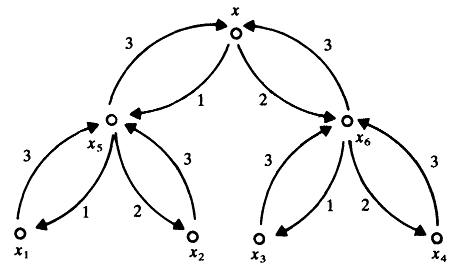
Now consider J on the undirected $2d$-maze $\mathscr{G}' = (G'_{m,d}, s, g)$ where s and g are in different components of $G'_{m,d}$. Assume J doesn't accept $\mathscr{G}'$. Corollary 4.11 tells us that when J runs on $\mathscr{G}'$, there must be at least one node of each copy of $G_{m,d}$ which is never visited by a pebble. It is easy to see that by redirecting some arcs that J never uses from one copy to the other, one can obtain a *threadable*, undirected $2d$-maze $\mathscr{G}''$ that J fails to accept. □

THEOREM 4.13. *There is no 3-JAG J such that if J is given an arbitrary undirected 3-maze $\mathscr{G}$, J accepts $\mathscr{G}$ if and only if $\mathscr{G}$ is threadable.*

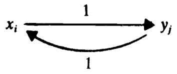
Proof outline. Let J be a 3-JAG with P pebbles, and assume that J accepts an arbitrary undirected 3-maze iff it is threadable. We can now show that for every d, there is a d-JAG J_d with P pebbles which accepts an arbitrary undirected d-maze iff it is threadable, contradicting Theorem 4.12. For simplicity, we shall only consider the case $d = 4$, but it should be clear how to generalize this.

Our 4-JAG J_4 will run on a 4-maze $\mathscr{G}$ by simulating the action of J on a 3-maze $\mathscr{G}'$ which is a "homomorphicish" version of $\mathscr{G}$. More particularly, if $\mathscr{G} = (G, s, g)$, let $\mathscr{G}' = (G', s, g)$, where G' is defined as follows.

For every node x of g, G' has nodes x, x_1, x_2, x_3, x_4, x_5, x_6. G' has arcs:



In addition, if G has an arc labeled i from x to y, and an arc labeled j from y to x, then G' has arcs



$\mathcal{G}'$ is threadable $\Leftrightarrow$ $\mathcal{G}$ is threadable, and it is easy to see how J_4 on $\mathcal{G}$ simulates J on $\mathcal{G}'$. □

We now state without proof a quantitative version of Theorem 4.13; the proof can be deduced in a straightforward way by an examination of the steps in the proof of Theorem 4.13.

THEOREM 4.14. *For some constant c the following is true. Let J be a* 3*-JAG with P pebbles and Q states. Then there is an undirected* 3*-maze* $\mathcal{G}$ *with* Q^{c^P} *nodes such that J accepts* $\mathcal{G}$ $\Leftrightarrow$ $\mathcal{G}$ *is not threadable.*

Theorem 4.14 does not yield a nice quantitative lower bound on storage as does Corollary 3.3 for the directed case. The following Fact shows that this is not possible.

Fact. For every $d > 0$ there is a constant c such that for every N there is a d-JAG J_N such that

(1) J_N accepts an undirected N-node d-maze iff it is threadable, and

(2) J_N uses storage $\leqq c \log N$.

In fact, J_N has only *two* pebbles and performs no jumps.

The proof involves showing the existence of a string $S_N \in \{1, 2, \cdots, d\}^*$ which (when viewed as a sequence of edges to follow) covers every connected undirected N-node d-graph, and which only has length polynomial in N (see [8]).

REFERENCES

[1] S. Y. KURODA, *Classes of languages and linear bounded automata*, Information and Control, 7 (1964), pp. 207–223.

[2] W. J. SAVITCH, *Relationships Between Nondeterministic and Deterministic Tape Complexities*, J. Comput. System Sci. 4 (1970), pp. 177–192.

[3] ———, *Maze Recognizing Automata and Nondeterministic Tape Complexities*, Ibid., 7 (1973), pp. 389–403.

[4] M. BLUM AND W. J. SAKODA, *On the Capability of Finite Automata in 2 and 3 Dimensional Space, Proc.* 18*th Annual Symposium on Foundations of Computer Science*, 1977, pp. 147–161.

[5] R. E. TARJAN, *Reference machines require non-linear time to maintain disjoint sets*, Proc. of the Ninth Annual ACM Symposium on Theory of Computing, May 1977, 18–29.

[6] M. O. RABIN, *Maze threading automata*, Seminar Talk presented at the University of California at Berkeley, Oct. 1967.

[7] S. A. COOK AND C. RACKOFF, *Space lower bounds for maze threadability on restricted machines*, University of Toronto, Dept. of Computer Science Technical Report 117, March 1978.

[8] R. ALELIUNAS, R. M. KARP, R. J. LIPTON, L. LOVÁSZ AND C. W. RACKOFF, *Random walks, universal traversal sequences, and the complexity of maze problems*, Proc. 20th Annual Symposium on Foundations of Computer Science, 1979, pp. 218–223.

SIAM J. COMPUT.
Vol. 9, No. 3, August 1980

PREDICTORS OF CONTEXT-FREE GRAMMARS*

KUO-CHUNG TAI†

Abstract. A *predictor* of a context-free grammar G is a substring of a sentence in $L(G)$ which determines unambiguously the contents of the parse stack immediately before (in top-down parsing) or after (in bottom-up parsing) symbols of the predictor are processed. Two types of predictors are defined, one for bottom-up parsers and the other for top-down parsers. Algorithms for finding predictors are given and the possible applications of predictors are discussed.

Key words. Context-free grammars, parsers, error recovery

1. Introduction. For a context-free grammar G, a *predictor* x of G is a terminal string which can predict, during the bottom-up (top-down) parsing of any string $\cdots x \cdots$ in $L(G)$, what the contents of the parse stack should be immediately before (after) symbols in x are parsed.

In this paper, two types of predictors of context-free grammars are defined, prefix-predictors for bottom-up parsers and suffix-predictors for top-down parsers. Properties of predictors are explored and algorithms for finding predictors are given.

The notion of predictors of context-free grammars can be applied to many problems including

(1) parallel compilation,
(2) complexity of context-free grammars, and
(3) error recovery in syntax analysis.

For syntactic error recovery, the concept of predictors formalizes the heuristic ideas of "important symbols" given in previous error recovery techniques. Moreover, the theory of predictors provides a basis for the evaluation of previous error recovery techniques and for the development of new techniques.

2. Terminology. A *context-free* (CF) *grammar* G is a quadruple (V_N, V_T, P, S), where V_N is a finite nonempty set of symbols called *nonterminals*, V_T is a finite set of symbols distinct from those in V_N called *terminals*, P is a finite set of pairs called *productions*, and S is a distinguished symbol in V_N called the *start symbol*. Each production is written $A \to \alpha$ and has a left part A in V_N and a right part α in V^*, where $V = V_N \cup V_T$. V^* denotes the set of all strings composed of symbols in V, including the empty string ε. An *ε-production* is a production of the form $A \to \varepsilon$. G is said to be *ε-free* if either there is no ε-production in P, or there is exactly one ε-production $S \to \varepsilon$ and S does not appear in the right part of any production in P.

The capital letters $A, B, \cdots, F, S$ denote nonterminals; X, Y, Z denote nonterminals or terminals; lowercase letters $a, b, c, \cdots, h$ denote terminals; $u, v, \cdots, z$ denote strings in V_T^*, and lowercase Greek letters $\alpha, \beta, \gamma, \cdots, \sigma$ denote strings in V^*. $|u|$ denotes the length of (number of symbols in) the string u; therefore $|\varepsilon| = 0$. $\varnothing$ denotes the empty set.

If $A \to \alpha$ is a production and $\beta A \gamma$ is a string in V^*, then $\beta A \gamma \overset{G}{\Rightarrow} \beta \alpha \gamma$ or $\beta A \gamma \Rightarrow \beta \alpha \gamma$ if G is clear. The transitive closure of $\Rightarrow$ is denoted by $\Rightarrow+$, and the reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow*$. A *derivation* of length n is a sequence of strings $\alpha_0, \alpha_1, \cdots, \alpha_n$, where $n \geqq 0$, such that $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n$; it is written $\alpha_0 \Rightarrow* \alpha_n$. A *rightmost derivation* is a derivation in which the rightmost nonterminal of each string is replaced to form the next; it is written $\alpha_0 \underset{rm}{\Rightarrow}* \alpha_n$. Similarly, a *leftmost*

* Received by the editors March 20, 1979 and in revised form September 12, 1979.

† Department of Computer Science, North Carolina State University, Raleigh, NC 27650. This research was supported in part by the National Science Foundation under Grant MCS77-24582.

derivation is written $\alpha_0 \underset{lm}{\Rightarrow}{}^* \alpha_n$. A string α is called a *sentential form* of G if $S \Rightarrow^* \alpha$. A sentential form α is called a *left-sentential* (*right-sentential*) *form* if $S \underset{lm}{\Rightarrow}{}^*$ $(\underset{rm}{\Rightarrow}{}^*)$ α. A *sentence* is a sentential form containing only terminal symbols. The *language* $L(G)$ generated by G is the set of sentences, i.e., $L(G) = \{w \text{ in } V_T^* \mid S \Rightarrow + w\}$.

A CF grammar G is said to be *cycle-free* if there is no derivation of the form $A \Rightarrow + A$ for any A in V_N. A nonterminal A is said to be *recursive* if $A \Rightarrow + \beta A \gamma$ for some β and γ in V^*. If $\beta = \varepsilon$, then A is said to be *left-recursive.* If $\gamma = \varepsilon$, then A is *right-recursive.* If $\beta \neq \varepsilon \neq \gamma$, then A is *self-embedding.* A symbol X in V $(X \neq S)$ is said to be *useless* if there does not exist a derivation of the form $S \Rightarrow * uXw \Rightarrow * uvw$, where u, v and w are in V_T^*. G is said to be *proper* if it is cycle-free, is ε-free, and has no useless symbols. A string α is said to be a *phrase* of a sentential form $\beta\alpha\gamma$ if $S \Rightarrow * \beta A \gamma \Rightarrow \beta \alpha \gamma$. A string α is said to be *the last phrase* of a derivation $\beta \Rightarrow + \gamma$ if the derivation is $\beta \Rightarrow * \delta A \sigma \Rightarrow \delta \alpha \sigma = \gamma$. A string σ is called a *viable prefix* if $S \underset{rm}{\Rightarrow} * \beta A w \underset{rm}{\Rightarrow} \beta \alpha w$ and σ is a prefix of $\beta\alpha$; thus, σ does not contain any symbol to the right of the last phrase of $S \underset{rm}{\Rightarrow} + \beta \alpha w$.

A *configuration* of a shift-reduce parser for a CFG is pair (α, x), where α represents the pushdown stack (or parse stack) whose top is on the right, and x represents the unparsed portion of the input. For LR parsers, α is a sequence of states (not symbols in V), but each sequence of states corresponds to a unique string in V^*. Without loss of generality, it is assumed that α is a string in V^*. A *move* by a shift-reduce parser from (α, x) to (α', x') is denoted by $(\alpha, x) \vdash (\alpha', x')$. In shift-reduce parsing there are two types of moves $(\alpha, ax) \vdash (\alpha a, x)$ (called a *shift*) and $(\alpha\beta, x) \vdash (\alpha B, x)$ (called a *reduce*) where $B \to \beta$ is a production. The transitive closure of $\vdash$ is denoted by $\vdash +$, and the reflexive transitive closure of $\vdash$ is denoted by $\vdash *$.

All grammars considered in this paper are assumed to be context-free with no useless symbols.

3. Prefix-predictors for bottom-up parsers.

DEFINITION. Let x be a string in V_T^+ and α be a string in V^*. α is called a *canonical prefix of G for x* if there exists w in V_T^* such that $S \underset{rm}{\Rightarrow} + \alpha x w$ with α not containing the last phrase of the rightmost derivation.

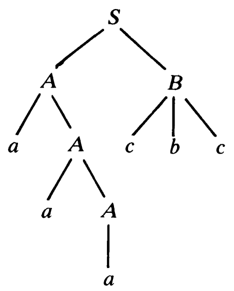
Consider the grammar $G1$ with productions

$$S \to AB,$$

$$A \to aA \mid a,$$

$$B \to cbc.$$

Then $L(G1) = \{a^n cbc \mid n > 0\}$. The parse tree of $aaacbc$ in $L(G1)$ is



Ac is a canonical prefix of $G1$ for b, but $aaac$, $aaAc$ and aAc are not. Both A and Acb are canonical prefixes of $G1$ for c.

DEFINITION. The set of canonical prefixes of G for x in V_T^+ is defined by $CP_G(x) = \{\alpha | \alpha$ is a canonical prefix of G for $x\}$.

DEFINITION. Let x be a string in V_T^+. x is called a *prefix-predictor* of G if $|CP_G(x)| = 1$, i.e., there exists exactly one canonical prefix of G for x.

In $G1$ b is a prefix-predictor because $CP_{G1}(b) = \{Ac\}$. Both a and c are not prefix-predictors of $G1$ since $CP_{G1}(a) = \{a^n | n \geqq 0\}$ and $CP_{G1}(c) = \{A, Acb\}$. However, cb is a prefix-predictor of $G1$ because $CP_{G1}(cb) = \{A\}$.

Note that for any y in V_T^+ such that $S \Rightarrow^+ \cdots y \cdots$, $|CP_G(y)| \geqq 1$. Let x be a prefix-predictor of G with $CP_G(x) = \{\alpha\}$. Then for any uxw in $L(G)$, $S \underset{rm}{\Rightarrow}^+ \alpha xw \Rightarrow^* uxw$ with α not containing the last phrase of the rightmost derivation from S to αxw. Thus during a deterministic, no-backtrack bottom-up parsing of uxw, immediately before symbols in x are parsed, the parse stack must be α or a sequence of states corresponding to α (for LR parsers).

How to find prefix-predictors will be discussed in the next section. In the remainder of this section, some properties of prefix-predictors are explored.

LEMMA 3.1. *Let x be a prefix-predictor of G. If $S \Rightarrow^+ \cdots xy \cdots$ for some y in V_T^+, then xy is also a prefix-predictor of G.*

Proof. Suppose that xy is not a prefix-predictor. Then there exist two or more canonical prefixes for xy. Since every canonical prefix for xy is also a canonical prefix for x, there exist two or more canonical prefixes for x. It follows that x is not a prefix-predictor. □

LEMMA 3.2. *Let x be a prefix-predictor of G and $CP_G(x) = \{\alpha\}$. Then x may appear more than once in some sentence of G if and only if $\alpha \underset{rm}{\Rightarrow}^+ \alpha xw$ for some w in V_T^*.*

Proof. If: trivial.

Only if: Let $uxwxv$ be a sentence of G. Since x is a prefix-predictor, $S \underset{rm}{\Rightarrow}^+ \alpha xwxv \underset{rm}{\Rightarrow}^* uxwxv$. For $\alpha xwxv$, there exists a string β such that $S \underset{rm}{\Rightarrow}^+ \beta xv \underset{rm}{\Rightarrow}^* \alpha xwxv$ with β not containing the last phrase of the rightmost derivation from S to βxv. Since x has a unique canonical prefix, $\beta = \alpha$. Therefore, $\alpha \underset{rm}{\Rightarrow}^+ \alpha xw$. □

Thus if a prefix-predictor appears more than once in some sentence of G, then G has at least one left-recursive nonterminal. However, as shown in the next lemma, it could happen that no prefix-predictors appear in any terminal string derived from a recursive nonterminal.

LEMMA 3.3. *Assume that $A \underset{rm}{\Rightarrow}^+ \beta Av$ with $\beta \neq \varepsilon$ and v in V_T^*. For any y in V_T^+ such that $A \Rightarrow^+ \cdots y \cdots$, y is not a prefix-predictor.*

Proof. Since G has no useless symbols, there exist strings α, γ in V^* and u, w in V_T^* such that $S \underset{rm}{\Rightarrow}^* \alpha Aw \underset{rm}{\Rightarrow}^+ \alpha\gamma yuw$ with $\alpha\gamma$ not containing the last phrase of the rightmost derivation from S to $\alpha\gamma yuw$. Then $\alpha\gamma$ is a canonical prefix for y. Since $A \underset{rm}{\Rightarrow}^+ \beta Av$, $S \underset{rm}{\Rightarrow}^* \alpha Aw \underset{rm}{\Rightarrow}^+ \alpha\beta Avw \underset{rm}{\Rightarrow}^+ \alpha\beta\gamma yuvw$. Thus $\alpha\beta\gamma$ is also a canonical prefix for y and $\alpha\beta\gamma \neq \alpha\gamma$. Therefore, y is not a prefix-predictor. □

LEMMA 3.4. *Assume that G is ε-free. If every terminal in V_T is a prefix-predictor, then $L(G)$ is a regular language.*

Proof. If $L(G) = \{\varepsilon\}$, then it is a regular language. If $L(G) \neq \{\varepsilon\}$, then for every nonterminal B, $B \Rightarrow^+ \cdots b \cdots$ for some b in V_T. Since b is a prefix-predictor, by Lemma 3.3 B is neither right-recursive nor self-embedding. Because all nonterminals are not self-embedding, $L(G)$ is regular. □

4. Finding prefix-predictors. This section first shows how to determine, for given strings α in V^* and x in V_T^+, whether α is a canonical prefix for x. Section 4.1

presents an algorithm for finding prefix-predictors of length one. Section 4.2 discusses how to find prefix-predictors of length greater than one.

The following two simple lemmas show the relations between "canonical prefixes" and "viable prefixes."

LEMMA 4.1. *Let αx be a viable prefix of G, where x is in V_T^+. Then α is a canonical prefix of G for x.*

Proof. Since αx is a viable prefix, there exists a string w in V_T^* such that $S \underset{rm}{\Rightarrow}{}^+ \alpha x w$ and αx does not contain any symbol to the right of the last phrase of the rightmost derivation. Because $|x|>0$, α does not contain the last phrase of the rightmost derivation. Thus α is a canonical prefix for x. □

LEMMA 4.2. *Suppose that $x = x_1 x_2 \cdots x_m$ is in V_T^m, $m>0$, and that α is a canonical prefix of G for x. Then αx_1 is a viable prefix, but $\alpha x_1 x_2 \cdots x_n$, $2 \leqq n \leqq m$, might not be a viable prefix.*

Proof. By assumption, there exists a string w in V_T^* such that $S \underset{rm}{\Rightarrow}{}^+ \alpha x w$ with α not containing the last phrase of the rightmost derivation. Since αx_1 certainly does not contain any symbol to the right of the last phrase, αx_1 is a viable prefix. However, the last phrase might be a suffix of αx_1 and thus $\alpha x_1 x_2 \cdots x_n$, $2 \leqq n \leqq m$, might contain symbols to the right of the last phrase. Hence, $\alpha x_1 x_2 \cdots x_n$, $2 \leqq n \leqq m$, might not be a viable prefix. For example, in $G1$ *aa* is a canonical prefix for *ac*. *aaa* is a viable prefix of $G1$, but not *aaac*. □

Thus "canonical prefixes" and "viable prefixes" are related but not equivalent. The above two lemmas lead to the following theorem:

THEOREM 4.3. *Let b be a terminal. α is a canonical prefix of G for b if and only if αb is a viable prefix of G.*

Proof. It follows from Lemmas 4.1 and 4.2. □

Note that the above theorem does not hold for a terminal string of length greater than one.

THEOREM 4.4. *Let α be a string in V^* and x a string in V_T^+. α is a canonical prefix for x if and only if there exists a viable prefix β such that either $\beta = \alpha x$ or $\beta \underset{rm}{\Rightarrow}{}^+ \alpha x$ with α not containing the last phrase of the rightmost derivation.*

Proof. If: If $\beta = \alpha x$, then αx is a viable prefix and thus α is a canonical prefix for x. If $\beta \underset{rm}{\Rightarrow}{}^+ \alpha x$, then there exists a string w in V_T^* such that $S \underset{rm}{\Rightarrow}{}^* \beta w \underset{rm}{\Rightarrow}{}^+ \alpha x w$ *with* α not containing the last phrase of the rightmost derivation from S to $\alpha x w$. Therefore, α is a canonical prefix for x.

Only if: Similar to the "if" part. □

To determine whether there exists a rightmost derivation from a viable prefix β to αx with α not containing the last phrase of the rightmost derivation is very complicated. An alternative is to perform reductions on αx based on the reverse of rightmost derivations.

To make reductions based on the reverse of rightmost derivations, we use the finite state machine corresponding to the LR(0) parser for G, called the *characteristic finite state machine* (CFSM) of G [2]. State s_0 in the CFSM is designated as the *start state*. Each state s in the CFSM, $s \neq s_0$, has a unique accessing symbol $A(s)$. A *path p* in the CFSM is a sequence of state $s_1, s_2, \cdots, s_m$ such that for each i, $1 \leqq i \leqq m-1$, s_i has a transition to s_{i+1} under $A(s_{i+1})$. An alternate notation for $p = s_1, s_2, \cdots, s_m$ is $[s_1 : \alpha]$, where $\alpha = A(s_2) \cdots A(s_m)$. For a path $p = s_1, s_2, \cdots, s_m$, we define $\mathrm{TOP}(p) = s_m$ and say that p *accesses* s_m and *passes* $s_1, s_2, \cdots$, and s_m. The *concatenation* of two paths $[s:\alpha]$ and $[s':\beta]$, where $\mathrm{TOP}([s:\alpha]) = s'$, is written $[s:\alpha][s':\beta]$ and denotes $[s:\alpha\beta]$. A *cycle* in a path $p = s_1, s_2, \cdots, s_m$ is a subsequence of p, say $s_i, s_{i+1}, \cdots, s_j$, such that $1 \leqq i < j \leqq m$, $s_i = s_j$ and $s_i \neq s_k$ for $i<k<j$.

LEMMA 4.5. *β is a viable prefix of G if and only if $[s_0:\beta]$ is a path in the CFSM of G.*

Proof. Trivial. □

If G is LR(0), then the CFSM of G is deterministic. Otherwise, the CFSM of G is nondeterministic because it may take two or more different actions in some states. The CFSM of G can be used to nondeterministically parse sentences in $L(G)$ from left to right, based on LR(0) (or canonical) parsing [2]. A canonical parse from configuration (α, x) to configuration (β, y) using the CFSM is denoted by $(\alpha, x) \vdash_{ca}^{*} (\beta, y)$.

LEMMA 4.6. *Let α be a viable prefix and x be a string in V_T^*. If $(\alpha, x) \vdash_{ca}^{*} (\beta, \varepsilon)$, then β is a viable prefix such that either $\beta = \alpha x$ or $\beta \underset{rm}{\Rightarrow}^{+} \alpha x$ with α not containing any symbol to the right of the last phrase of the rightmost derivation.*

Proof. This is due to the construction of the CFSM. □

THEOREM 4.7. *Let α be a string in V^* and $x = x_1x_2 \cdots x_m$ be a string in V_T^m, $m > 0$. α is a canonical prefix for x if and only if αx_1 is a viable prefix and $(\alpha x_1, x_2 \cdots x_m) \vdash_{ca}^{*} (\beta, \varepsilon)$ for some β in V^+.*

Proof. If: By Lemma 4.6, β is a viable prefix such that either $\beta = \alpha x$ or $\beta \underset{rm}{\Rightarrow}^{+} \alpha x$ with α not containing the last phrase of the rightmost derivation. Then by Theorem 4.4, α is a canonical prefix for x.

Only if: Since α is a canonical prefix for x, by Lemma 4.2 αx_1 is a viable prefix. By Theorem 4.4, there exists a viable prefix β such that either $\beta = \alpha x$ or $\beta \underset{rm}{\Rightarrow}^{+} \alpha x$ with α not containing the last phrase of the rightmost derivation. Thus $(\alpha x_1, x_2 \cdots x_m) \vdash_{ca}^{*} (\beta, \varepsilon)$. □

Thus whether α is a canonical prefix for $x = x_1x_2 \cdots x_m$, $m > 0$, can be determined as follows:

(1) If $[s_0:\alpha x_1]$ is not a path in the CFSM of G, then α is not a canonical prefix for x.

(2) If $[s_0:\alpha x_1]$ is a path in the CFSM of G:

(2.1) if $m = 1$, then α is a canonical prefix for x;

(2.2) if $m > 1$, then derive canonical parses of $(\alpha x_1, x_2 \cdots x_m)$ based on the CFSM of G. If $(\alpha x_1, x_2 \cdots x_m)$ can be reduced to a (β, ε) by some canonical parse, then α is a canonical prefix for x. Otherwise, α is not a canonical prefix for x. (This step takes a finite amount of time if G is proper.)

4.1. Finding prefix-predictors of length one. Let b be a terminal. By Theorem 4.3, α is a canonical prefix of G for b if and only if $[s_0:\alpha b]$ is a path in the CFSM of G. Thus, b is a prefix-predictor of G if and only if there exists a unique path from s_0, say $[s_0:\beta]$, such that β ends with b. The following theorem provides the basis for finding prefix-predictors of length one.

THEOREM 4.1.1. *Let b be a terminal. b is a prefix-predictor of G if and only if there exists a state s in the CFSM of G such that* (1) *s is the only state with $A(s) = b$, and* (2) *there exists exactly one path from s_0 to s.*

Proof. It follows from the above discussion. □

The remainder of this section shows how to construct an algorithm for finding prefix-predictors of length one.

DEFINITION. A state s in the CFSM of G is called a *singular state* if there exists exactly one path from s_0 to s.

LEMMA 4.1.2. *If s is a singluar state, then each state passed by the path from s_0 to s is a singular state.*

Proof. Trivial. □

LEMMA 4.1.3. *If s is not singular and there exists a path from s to s', then s' is not singular.*

Proof. Trivial. □

Figure 1 shows the CFSM of $G2$. States in the CFSM, except s_1, s_6, s_9, s_{12} and s_{15}, are singular states. Note that s_6 is not singular because there is a path from s_6 to itself. Since there exist paths from s_6 to s_{12} and s_{15}, respectively, both s_{12} and s_{15} are not singular.

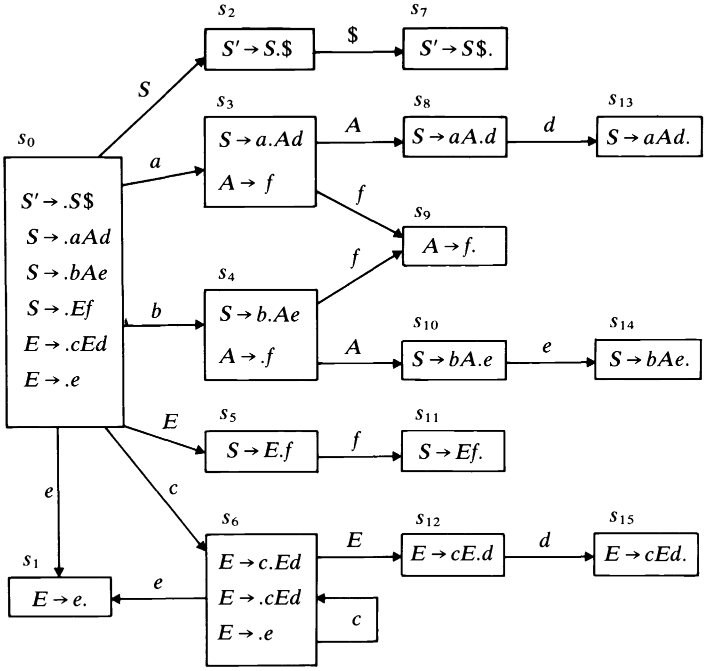


FIG. 1. *The CFSM of the grammar $G2$ with productions*

$S' \to S\$$, $A \to f$,

$S \to aAd$, $E \to cEd$,

$S \to bAe$, $E \to e$.

$S \to Ef$,

LEMMA 4.1.4. *Let b be a terminal. b is a prefix-predictor of G if and only if there exists a singular state s in the CFSM of G such that s is the only state with $A(s) = b$.*

Proof. It follows directly from Theorem 4.1.1. □

By the above lemma, $G2$ has three prefix-predictors of length one, a, b, and \$.

ALGORITHM 4.1.5. Find prefix-predictors of length one of G.

(1) Construct the set M of singular states in the CFSM of G as follows.

(1.1) Initially, let $M = \{s_0\}$ with s_0 "unmarked".

(1.2) For each unmarked state s in M, mark it by performing the following step: For each immediate successor s' of s, if s is the only immediate predecessor of s', then add s' to M as an unmarked state.

(2) Construct the set N of prefix-predictors of length one of G as follows.

(2.1) Initially, let $N = \varnothing$.

(2.2) For each state s in M, if the accessing symbol of s, say b, is not the accessing symbol of any other state in the CFSM of G, then add b to N.

4.2. Finding prefix-predictors of length greater than one. Let $x = x_1x_2 \cdots x_m$ be a prefix-predictor of G, $m > 1$, and $\mathrm{CP}_G(x) = \{\alpha\}$. x has several properties different from those of prefix-predictors of length one. First, αx_1 is a viable prefix, while αx might not be a viable prefix. Second, $[s_0 : \alpha x_1]$ may pass nonsingular states. Third, $[s_0 : \alpha x_1]$ may have cycles.

Consider the CFSM of $G2$ in Figure 1. Since $\mathrm{CP}_{G2}(f) = \{E, a, b\}$, f is not a prefix-predictor of $G2$. But fd and fe are prefix-predictors of $G2$ because $\mathrm{CP}_{G2}(fd) = \{a\}$ and $\mathrm{CP}_{G2}(fe) = \{b\}$. Note that afd is not a viable prefix and that $[s_0 : af]$ accesses state s_9 which is not singular.

In $G2$, the nonterminal E derives c^ned^n,for any $n \geqq 1$. Since E is not left-recursive, by Lemma 3.3 no substring of c^ned^n, $n \geqq 1$, is a prefix-predictor of $G2$. However, d^nf is a prefix-predictor because $\mathrm{CP}_{G2}(d^nf) = \{c^nE\}$. Note that $[s_0 : c^nE]$ has cycles if $n > 1$.

To find prefix-predictors of length m of G, let us consider the set $K_{G,m}$ defined by

$$K_{G,m} = \{(\beta, y) | y \text{ is in } V_T^m \text{ and } \beta \text{ is a canonical prefix of } G \text{ for } y\}.$$

Then x is a prefix-predictor of length m of G if and only if there exists exactly one (β, y) in $K_{G,m}$ with $y = x$. However, if the CFSM of G has cycles, the length of β in $K_{G,m}$ may be unbounded and thus $K_{G,m}$ could be an infinite set. A simple solution is to find a value called BOUND(G, m) satisfying the following property:

PROPERTY 4.2.1. For any y in V_T^m, $m > 0$, such that $S \Rightarrow + \cdots y \cdots$,

(1) if y is a prefix-predictor of G and $\mathrm{CP}_G(y) = \{\alpha\}$, then $|\alpha| \leqq \mathrm{BOUND}(G, m)$;

(2) if y is not a prefix-predictor of G, then there exist at least two distinct canonical prefixes of G for y with their lengths less than or equal to BOUND(G, m).

By defining

$$L_{G,m} = \{(\beta, y) | y \text{ is in } V_T^m, \beta \text{ is a canonical prefix of } G \text{ for } y \text{ and } |\beta| \leqq \mathrm{BOUND}(G, m)\},$$

then x is a prefix-predictor of length m of G if and only if $L_{G,m}$ contains exactly one (β, y) with $y = x$. Thus the problem now is to find a value of BOUND(G, m) which satisfies Property 4.2.1.

Let MAXPATH(G) be the maximum length of cycle-less paths in the CFSM of G, MAXRP(G) be the maximum length of right parts of productions in G, and #NT(G) be the number of nonterminals in G.

THEOREM 4.2.2. *Let G be a proper CF grammer. By letting*

$$\mathrm{BOUND}(G, m) = 2 * \mathrm{MAXPATH}(G) + (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP(G)} - 1),$$

Property 4.2.1 *is satisfied.*

Proof. Let β be a canonical prefix of G for y in $V_T^m, m > 0$. We show in the Appendix that if $|\beta| > \mathrm{BOUND}(G, m)$, there exist two distinct canonical prefixes of G for y with their lengths less than or equal to BOUND(G, m). Therefore, Property 4.2.1 is satisfied. □

Thus for $m > 1$, prefix-predictors of length m of a proper context-free grammar can be found as follows.

(1) Let $Q_1 = \{(\beta, b) | [s_0 : \beta b]$ is a path in the CFSM of G, b is in V_T, and $|\beta| \leqq \mathrm{BOUND}(G, m)\}$.

(2) For $i = 2, 3, \cdots$, and m, let $Q_i = \{(\beta, ub) | (\beta, u)$ is in Q_{i-1}, b is in V_T, and there exists a canonical parse $(\beta, ub) \underset{ca}{\vdash^*} (\gamma, b) \underset{ca}{\vdash} (\gamma b, \varepsilon)\}$.

(3) Let $N_m = \{x | Q_m$ has exactly one (β, y) with $y = x\}$. Then N_m is the set of prefix-predictors of length m of G.

5. Suffix-predictors for top-down parsers.

DEFINITION. Let y be a string in V_T^+ and let β be a string in V^*. β is called a *canonical suffix of G for y* if there exists a string u in V_T^* such that $S \underset{lm}{\Rightarrow}+ uy\beta$ with β not containing the last phrase of the leftmost derivation.

DEFINITION. The set of canonical suffixes of G for y in V_T^+ is defined by

$$\mathrm{CS}_G(y) = \{\beta \mid \beta \text{ is a canonical suffix of } G \text{ for } y\}.$$

DEFINITION. Let y be a string in V_T^+. y is called a *suffix-predictor of G* if $|\mathrm{CS}_G(y)| = 1$, i.e., there exists exactly one canonical suffix of G for y.

Let y be a suffix-predictor of G with $\mathrm{CS}_G(y) = \{\beta\}$. Then for any uyw in $L(G)$, $S \underset{lm}{\Rightarrow}+ uy\beta \Rightarrow* uyw$ with β not containing the last phrase of the leftmost derivation from S to $uy\beta$. Thus during a deterministic, no-backtrack top-down parsing of uyw, immediately after symbols in y are parsed, the parse stack must be β^R.

Let the *reflection of a string* $y = y_1 \cdots y_m$ be $y^R = y_m \cdots y_1$ and let the *reflection of a grammar* $G = (V_N, V_T, P, S)$ be $G^R = (V_N, V_T, P^R, S)$, where P^R is P with all right parts reversed. The following theorem says that suffix-predictors of G can be found by using algorithms for finding prefix-predictors.

THEOREM 5.1. *y is a suffix-predictor of G if and only if y^R* is a prefix-predictor of G^R.

Proof. It is sufficient to show that β is a canonical suffix of G for y if and only if β^R is a canonical prefix of G^R for y^R.

If: Since β^R is a canonical prefix of G^R for y^R, $S \overset{G^R}{\underset{rm}{\Rightarrow}}+ \beta^R y^R w$ with β^R not containing the last phrase of the rightmost derivation. Then $S \overset{G}{\underset{lm}{\Rightarrow}}+ w^R y\beta$ with β not containing the last phrase of the leftmost derivation. Thus β is a canonical suffix of G for y.

Only if. Similar to the "if" part. □

Some of the properties of prefix-predictors discussed in § 3 can be transformed for suffix-predictors.

LEMMA 5.2. *Let y be a suffix-predictor of G. If $S \Rightarrow+ \cdots xy \cdots$, then xy is also a suffix-predictor.*

Proof. Similar to the proof of Lemma 3.1. □

LEMMA 5.3. *Let y be a suffix-predictor of G and* $\mathrm{CS}_G(y) = \{\beta\}$. Then y may appear more than once in some sentence of G if and only if $\beta \underset{lm}{\Rightarrow}+ uy\beta$ for some u in V_T^*.

Proof. Similar to the proof of Lemma 3.2. □

LEMMA 5.4. *Assume that $A \underset{lm}{\Rightarrow}+ uA\gamma$ with $\gamma \neq \varepsilon$ and u in V_T^*. For any y in V_T^+ such that $A \Rightarrow+ \cdots y \cdots$, y is not a suffix-predictor.*

Proof. Similar to the proof of Lemma 3.3. □

LEMMA 5.5. *Assume that G is ε-free. If every terminal in V_T is a suffix-predictor, then $L(G)$ is a regular language.*

Proof. Similar to the proof of Lemma 3.4. □

Pai and Kieburtz [9] defined the *strong phrase level uniqueness* property for syntactic error recovery in top-down parsing.

DEFINITION. A symbol Z in V has *strong phrase level uniqueness* (SPLU) if

(1) there is at most one production, say $A \to \alpha$, which contains Z in its right part,

(2) Z occurs only once in the right part α,

(3) A does not have left or embedded recursion in G, i.e., there is no derivation of the form $A \Rightarrow* \beta A\gamma$, where $\gamma \neq \varepsilon$,

(4) A has SPLU in $G(A)$, where $G(A)$ is the grammar obtained from G by deleting all productions for A.

THEOREM 5.6. *If a terminal b has SPLU in G, then b is a suffix-predictor of G.*

Proof. Since b has SPLU in G, there exists a unique sequence of productions

$$\begin{aligned} &A^{(1)} \to \alpha^{(1)} A^{(2)} \beta^{(1)}, \\ &A^{(2)} \to \alpha^{(2)} A^{(3)} \beta^{(2)}, \\ &\quad\vdots \\ &A^{(n)} \to \alpha^{(n)} b \beta^{(n)} \end{aligned}$$

where $A^{(1)} = S$, $n > 0$, and $A^{(1)}, A^{(2)}, \cdots$, and $A^{(n)}$ are distinct. Furthermore, for each $A^{(i)}$, $1 \leqq i \leqq n$, there is no derivation of the form $A^{(i)} \Rightarrow + \alpha A^{(i)} \gamma$ with $\gamma \neq \varepsilon$. Thus $\beta^{(n)} \beta^{(n-1)} \cdots \beta^{(1)}$ is the only canonical suffix of G for b. It follows that b is a suffix-predictor of G. □

However, the reverse of the above theorem is not true. That is, there may exist suffix-predictors of length one of G which do not have SPLU in G. Consider the grammar $G3$ with production

$$S \to abc \mid dbc.$$

b does not have SPLU in $G3$ because it appears in the right parts of two productions. But b is a suffix-predictor of G because $\mathrm{CS}_{G3}(b) = \{c\}$.

6. Applications of predictors. A predictor generator for context-free grammars is being implemented. Research is also underway on the applications of predictors to the following problems.

Parallel compilation. Let x be a prefix-predictor with $\mathrm{CP}_G(x) = \{\alpha\}$ and let uxw be a string in $L(G)$. The bottom-up parsing of uxw can be performed by two processes in parallel. One is to parse u with its parse stack empty, while the other is to parse xw with α as its parse stack. Furthermore, the interface between these two processes for other phases of compilation (e.g., semantic processing and code generation) can be formally defined in terms of the attributes [6] of symbols in α. Thus if a string in $L(G)$ has n separated predictors, it can be divided into $n+1$ segments such that these segments are compiled in parallel or separately.

Complexity of context-free grammars. Recently the complexity problems of both context-free grammar forms and context-free grammars have received much attention [3], [4], [12]. The concept of predictors can be applied to define complexity measures for context-free grammars.

Syntactic error recovery. Prefix-predictors (suffix-predictors) can be used for error recovery in bottom-up (top-down) parsing as follows. After an error is detected, symbols in the remaining input are discarded until a prefix-predictor (suffix-predictor), say x, is found. Assume that $\mathrm{CP}_G(x)$ $(\mathrm{CS}_G(x)) = \{\alpha\}$. The current parse stack is replaced with α (α^R) before parsing is resumed. For top-down parsing, however, x must be discarded from the input before parsing is resumed.

Although a number of error recovery techniques have been proposed, most of them are ad hoc and heuristic. The theory of predictors provides a basis for comparing and evaluating previous error recovery techniques [1], [5], [7], [8], [9], [10], [11] because these techniques can be described in terms of different approximations to predictors. Moreover, the notion of predictors can be extended to define "local" predictors which determine how to recover from syntax errors by examining both the parse stack and the remaining input. Several types of local predictors have been defined and are being applied for practical error recovery.

Appendix. Let β be a canonical prefix of G for y in V_T^m, $m>0$, with $|\beta|>$ BOUND$(G,m) = 2 * \text{MAXPATH}(G) + (m+1) * \#\text{NT}(G) * (\text{MAXRP}(G)-1)$. Below we prove that there exist two distinct canonical prefixes of G for y with their lengths less than or equal to BOUND(G, m). The following lemma is needed for the proof.

LEMMA A.1. *Let α be a viable prefix of G. Suppose that the path $[s_0:\alpha]$ in the CFSM of G has cycles, say $[s_0:\alpha]=[s_0:\gamma][s:\delta][s:\sigma]$, where $\delta \neq \varepsilon$ and $\alpha = \gamma\delta\sigma$. Then $[s_0:\gamma][s:\sigma]$ is a path in the CSFM of G, $\gamma\sigma$ is a viable prefix of G, and* TOP$([s_0:\alpha])$ = TOP$([s_0:\gamma\sigma])$.

Proof. Trivial. □

Since β is a canonical prefix for y, by Theorem 4.4 there exists a viable prefix γ such that $\gamma=\beta y$ or $\gamma \underset{rm}{\Rightarrow}^{+} \beta y$ with β not containing the last phrase of the rightmost derivation. Consider the following two cases:

(1) $\gamma=\beta y$. Since $|\beta|>2*\text{MAXPATH}(G)$, $[s_0:\beta]$ has at least two cycles. By Lemma A.1, β can be shortened to produce two viable prefixes β' and β'' such that $[s_0:\beta']$ has no cycles, $[s_0:\beta'']$ has exactly one cycle, and TOP$([s_0:\beta])$ = TOP$([s_0:\beta'])$ = TOP$([s_0:\beta''])$. Then $\beta' \neq \beta''$ and both $|\beta'|$ and $|\beta''|$ are less than or equal to $2*\text{MAXPATH}(G)$. Since both $\beta' y$ and $\beta'' y$ are viable prefixes, β' and β'' are canonical prefixes for y.

(2) $\gamma \underset{rm}{\Rightarrow}^{+} \beta y$ with β not containing the last phrase of the rightmost derivation. Let δ be the longest prefix of γ not replaced in the rightmost derivation. Thus $\gamma=\delta\sigma$ and $\beta=\delta\eta$ for some σ in V^+ and η in V^*. It follows that $\sigma \underset{rm}{\Rightarrow}^{+} \eta y$ with σ_1 deriving a prefix of y in the rightmost derivation. Since $|\beta|=|\delta\eta|>$ $2*\text{MAXPATH}(G)+(m+1)*\#\text{NT}(G)*(\text{MAXRP}(G)-1)$, either $|\delta|>$ $2*\text{MAXPATH}(G)$ or $|\eta|>(m+1)*\#\text{NT}(G)*(\text{MAXRP}(G)-1)$ (or both). Consider the following cases.

Case A.1. $|\delta|>2*\text{MAXPATH}(G)$. By Lemma A.1, δ can be shortened to produce two viable prefixes δ' and δ'' such that $[s_0:\delta']$ has no cycles, $[s_0:\delta'']$ has exactly one cycle, and TOP $([s_0:\delta])$ = TOP $([s_0:\delta'])$ = TOP $([s_0:\delta''])$. Then $\delta' \neq \delta''$ and $|\delta'|$ and $|\delta''|$ are less than or equal to $2*\text{MAXPATH}(G)$.

Case A.2. $|\eta|>(m+1)*\#\text{NT}(G)*(\text{MAXRP}(G)-1)$. Since $\sigma \underset{rm}{\Rightarrow}^{+} \eta y$ with σ_1 deriving a prefix of y in the rightmost derivation, $\sigma=\sigma_1\sigma' \underset{rm}{\Rightarrow}^{*} \sigma_1 y'' \underset{rm}{\Rightarrow}^{+} \eta y' y'' = \eta y$, where $y' \neq \varepsilon$. We show below that the derivation $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta y'$ can be shortened to produce two derivations $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta' y'$ and $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta'' y'$ such that $\eta' \neq \eta''$ and both $|\eta'|$ and $|\eta''|$ are less than or equal to $(m+1)*\#\text{NT}(G)*(\text{MAXRP}(G)-1)$.

The sequence of productions applied in the derivation $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta y'$ is as follows:

$$\begin{aligned} A^{(0)} &\to \eta^{(1)}A^{(1)}y^{(1)}\\ A^{(1)} &\to \eta^{(2)}A^{(2)}y^{(2)}\\ &\vdots\\ A^{(p-2)} &\to \eta^{(p-1)}A^{(p-1)}y^{(p-1)}\\ A^{(p-1)} &\to \eta^{(p)}y^{(p)}, \end{aligned}\tag{A.1}$$

where $A^{(0)}=\sigma_1$, $\eta^{(1)}\cdots\eta^{(p)}=\eta$, $y^{(p)}\cdots y^{(1)}=y'$ and $y^{(p)}\neq\varepsilon$ (because $\delta\eta$ is a canonical prefix for y'). Since $|y'|\leqq m$ and $y^{(p)}\neq\varepsilon$, at most $m-1$ of $y^{(1)},\cdots,y^{(p-1)}$ are nonempty. Thus the sequence of productions in (1) can be divided into subsequences

$Q_1, Q_2, \cdots, Q_n$, where $n \leqq m$ and each subsequence is of the following form

$$
\begin{aligned}
&A^{(q)} \to \eta^{(q+1)} A^{(q+1)} \\
&A^{(q+1)} \to \eta^{(q+2)} A^{(q+2)} \\
\text{(A.2)} \qquad &\vdots \\
&A^{(r-2)} \to \eta^{(r-1)} A^{(r-1)} \\
&A^{(r-1)} \to \eta^{(r)} A^{(r)} y^{(r)},
\end{aligned}
$$

where $0 \leqq q < r \leqq p$, $y^{(r)} \neq \varepsilon$, and $y^{(q)} \neq \varepsilon$ if $q > 0$.

The subsequence shown in (A.2) can be applied to produce a derivation

$$A^{(q)} \underset{rm}{\Rightarrow} \eta^{(q+1)} A^{(q+1)} \underset{rm}{\Rightarrow} \eta^{(q+1)} \eta^{(q+2)} A^{(q+2)} \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \eta^{(q+1)} \cdots \eta^{(r)} A^{(r)} y^{(r)}.$$

Assume that in this subsequence a nonterminal appears twice in the left parts of productions, say $A^{(s)} = A^{(t)}$ where $q \leqq s < t < r$. Then the subsequence shown in (A.2) can be shortened by deleting the following sequence of productions

$$
\begin{aligned}
&A^{(s)} \to \eta^{(s+1)} A^{(s+1)} \\
&A^{(s+1)} \to \eta^{(s+2)} A^{(s+2)} \\
&\vdots \\
&A^{(t-1)} \to \eta^{(t)} A^{(t)}.
\end{aligned}
$$

The shortened subsequence can be applied to produce a derivation $A^{(q)} \underset{rm}{\Rightarrow}^{*} \eta^{(q+1)} \cdots \eta^{(s)} A^{(s)} \underset{rm}{\Rightarrow} \eta^{(q+1)} \cdots \eta^{(s)} \eta^{(t+1)} A^{(t+1)} y^{(t+1)} \underset{rm}{\Rightarrow}^{*} \eta^{(q+1)} \cdots \eta^{(s)} \eta^{(t+1)} \cdots \eta^{(r)} A^{(r)} y^{(r)}$. Since G is cycle-free, $\eta^{(s+1)} \cdots \eta^{(t)} \neq \varepsilon$ and therefore $|\eta^{(q+1)} \cdots \eta^{(s)} \eta^{(t+1)} \cdots \eta^{(r)}| < |\eta^{(q+1)} \cdots \eta^{(r)}|$. If the shortened subsequence still has a nonterminal appearing twice in the left parts of productions, the same process can be repeated. When the resulting subsequence has no nonterminal appearing more than once in the left parts of productions, it can be applied to produce a derivation $A^{(q)} \underset{rm}{\Rightarrow}^{+} \theta A^{(r)} y^{(r)}$ with $|\theta| < \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$.

Since $\eta = \eta^{(1)} \cdots \eta^{(p)}$, $|\eta| > (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$ and $|\eta^{(i)}| \leqq \mathrm{MAXRP}(G) - 1$ for $1 \leqq i \leqq p$, so $p > (m+1) * \#\mathrm{NT}(G)$. As mentioned earlier, the sequence of p productions shown in (A.1) is divided into n subsequences, $n \leqq m$. Then at least one of subsequences $Q_1, Q_2, \cdots Q_n$, say Q_t, consists of more than $\#\mathrm{NT}(G)$ productions. Thus in Q_t at least one nonterminal appears more than once in the left parts of productions. Suppose Q_t is of the form shown in (A.2). Q_t can be shortened to generate two subsequences Q_t' and Q_t'' (one of them may be equal to Q_t) such that Q_t' and Q_t'' produce derivations $A^{(q)} \underset{rm}{\Rightarrow}^{+} \theta' A^{(r)} y^{(r)}$ and $A^{(q)} \underset{rm}{\Rightarrow}^{+} \theta'' A^{(r)} y^{(r)}$ with $|\theta'| < |\theta''| < 2 * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$.

Some of the subsequences $Q_1, \cdots, Q_{t-1}, Q_{t+1}, \cdots, Q_n$ may also have nonterminals appearing more than once in the left parts of productions. Let Q_i' be the shortened Q_i, $1 \leqq i \leqq n$ and $i \neq t$, such that Q_i' has no nonterminals appearing more than once in the left parts of productions (possibly $Q_i' = Q_i$). Then each Q_i', $1 \leqq i \leqq n$ and $i \neq t$, is of the form shown in (A.2) and produces a derivation of the form

$A^{(q)} \underset{rm}{\Rightarrow}^{+} \theta A^{(r)} y'^{(r)}$ with $|\theta| < \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$. Therefore, the two sequences $Q'_i, \cdots, Q'_{t-1}, Q'_t, Q'_{t+1}, \cdots, Q'_n$ and $Q'_1, \cdots Q'_{t-1}, Q''_t, Q'_{t+1}, \cdots, Q'_n$ produce derivations $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta' y'$ and $\sigma_1 \underset{rm}{\Rightarrow}^{+} \eta'' y'$ respectively with $|\eta'| < |\eta''| < (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$. In other words, there exist two distinct strings η' and η'' such that both $|\eta'|$ and $|\eta''|$ are less than $(m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$ and $\sigma \underset{rm}{\Rightarrow}^{+} \eta' y$ and $\sigma \underset{rm}{\Rightarrow}^{+} \eta'' y$ with σ_1 deriving a prefix of y in both derivations.

From Cases (A.1) and (A.2), let γ', γ'', β', and β'' be defined as follows:

(a) $\gamma' = \delta'\sigma$, $\gamma'' = \delta''\sigma$, $\beta' = \delta'\eta$, and $\beta'' = \delta''\eta$
if $|\delta| > 2 * \mathrm{MAXPATH}(G)$
and $|\eta| \leqq (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$,

or

(b) $\gamma' = \gamma'' = \delta\sigma$, $\beta' = \delta\eta'$, and $\beta'' = \delta\eta''$
if $|\delta| \leqq 2 * \mathrm{MAXPATH}(G)$
and $|\eta| > (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$,

or

(c) $\gamma' = \gamma'' = \delta'\sigma$, $\beta' = \delta'\eta'$, and $\beta'' = \delta'\eta''$
if $|\delta| > 2 * \mathrm{MAXPATH}(G)$
and $|\eta| > (m+1) * \#\mathrm{NT}(G) * (\mathrm{MAXRP}(G) - 1)$.

Then γ' and γ'' are viable prefixes of G and $\gamma' \underset{rm}{\Rightarrow}^{+} \beta' y$ ($\gamma'' \underset{rm}{\Rightarrow}^{+} \beta'' y$) with β' (β'') not containing the last phrase of the rightmost derivation. Therefore, β' and β'' are distinct canonical prefixes of G for y with both $|\beta'|$ and $|\beta''|$ less than or equal to $\mathrm{BOUND}(G, m)$. □

REFERENCES

[1] J. CIESINGER, *Generating error recovery in a compiler generating system*, GI-4 Fachtagung über Programmiersprachen, Springer-Verlag, New York, 1976, pp. 185–193.

[2] F. L. DEREMER, *Simple LR(k) grammars*, Comm. ACM, 14 (1971), pp. 453–460.

[3] S. GINSBURG AND N. LYNCH, *Size complexity in context-free grammar forms*, J. Assoc. Comput. Mach., 23 (1976), pp. 582–598.

[4] ———, *Derivation complexity in context-free grammar forms*, this Journal, 6 (1977), pp. 123–138.

[5] S. L. GRAHAM AND S. P. RHODES, *Practical syntactic error recovery*, Comm. ACM, 18 (1975), pp. 639–650.

[6] D. E. KNUTH, *Semantics of context-free languages*, Math. Systems Theory, 2 (1968), pp. 127–145.

[7] J. P. LEVY, *Automatic correction of syntax errors in programming languages*, Acta Informat. 4 (1975), pp. 271–292.

[8] M. D. MICKUNAS AND J. A. MODRY, *Automatic error recovery for LR parsers*, Comm. ACM, 21 (1978), pp. 459–465.

[9] A. PAI AND R. B. KIEBURTZ, *Global context recovery: a new strategy for parser recovery from syntax errors*, SIGPLAN Notices 14 (1979), pp. 158–167.

[10] T. J. PENNELLO AND F. DEREMER, *A forward move for LR error recovery*, Conf. Rec. Fifth ACM Symp. Principles of Programming Languages, 1978, pp. 241–254.

[11] K. C. TAI, *Syntactic error correction in programming languages*, IEEE Trans. Software Engineering, 4 (1978), pp. 414–425.

[12] D. WORKMAN, *A measure of structural complexity for context-free grammars*. Technical Report CSD-TR-129, Computer Science Dept., Purdue Univ., 1975.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0001 $01.00/0

COMPLETENESS WITH FINITE SYSTEMS OF INTERMEDIATE ASSERTIONS FOR RECURSIVE PROGRAM SCHEMES*

KRZYSZTOF R. APT† AND LAMBERT G. L. T. MEERTENS‡

Abstract. It is proved that in the general case of arbitrary context-free schemes a program is (partially) correct with respect to given initial and final assertions if and only if a suitable *finite* system of intermediate assertions can be found. Assertions are allowed from the extended state space $\mathscr{V}\times\mathscr{V}$. This result contrasts with the results of [2], where it is proved that if assertions are taken from the original state space $\mathscr{V}$, then in the general case an *infinite* system of intermediate assertions is needed. The extension of the state space allows a unification in the relational framework of [2], of the (essence of the) results of [2], and of [4], [5] and [6], and provides a semantic counterpart of the use of auxiliary variables.

Key words. partial correctness, intermediate assertions, relational framework, extended state space, recursive program schemes

1. Introduction.

De Bakker and Meertens proved in [2] that an *infinite* system of intermediate assertions is needed to prove the completeness of the inductive assertion method in the case of an arbitrary system of (mutually) recursive parameterless procedures. On the other hand, Gorelick in [5] extended the results of [3] and obtained a completeness result for a Hoare-like axiomatic system (see [7]) for a fragment of ALGOL 60 in which (deterministic) systems of recursive procedures are allowed. Thus any true asserted statement is provable. (Observe, however, that the axiomatic system uses an oracle determining the truth of formulas from the underlying assertion language.) From the proof we can extract all intermediate assertions about atomic substatements of the original program. Since proofs are finite, we obtain a *finite* system of intermediate assertions, thus apparently contradicting the result of [2]. Also [4] and [6] avoid the necessity of an infinite number of assertions by using an extension of the inductive assertion method.

The purpose of this paper is to investigate this issue in the relational framework of [2] and to obtain, within that framework, a unification of the (essence of the) results of [2] and of [4], [5] and [6]. The solution of the apparent contradiction lies in the fact that in [4], [5] and [6] auxiliary variables are used (to store the initial values of variables). These auxiliary variables have no semantic counterpart in the relational framework of [2]. Semantically, the use of auxiliary variables corresponds to the use of states which have an additional coordinate (from a space $\mathscr{W}$) inaccessible to a program. We shall call the domain $\mathscr{V}\times\mathscr{W}$ of such states an extended state space.

We prove that if one allows intermediate assertions from the extended state space $\mathscr{V}\times\mathscr{V}$, then one can always find a *finite* system of intermediate assertions. More precisely, a program is partially correct with respect to given initial and final assertions if and only if a suitable finite system of assertions from the extended state space can be found. Thus for the space $\mathscr{W}$ one can take the original state space $\mathscr{V}$. Theorem 4.4 of [2] shows that for $\mathscr{W}$ one could also take the set of all so-called index-triple sequences, so that these two completeness results differ only in the choice of the extended state space. Our choice is both more economical and easier to use in the concrete proofs.

* Received by the editors December 27, 1978, and in revised form October 19, 1979. This publication is a revised form of the Mathematical Centre Report IW 84/77.

† Faculty of Economics, Erasmus University Rotterdam, P.O. Box 1738, Rotterdam, The Netherlands.
‡ Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

In [2] it is proved that in the case of regular declaration schemes (corresponding to flow-chart programs) one can always find a finite system of intermediate assertions taken from the original state space. In more syntactical terms this can be interpreted as a statement that auxiliary variables are not needed for correctness proofs in the case of flow-chart programs. They are needed in the general case of arbitrary systems of (parameterless) procedure declarations.

In the relational framework any subset of the state space can be taken as an assertion. This is not the case with a more syntactical approach in which assertions are formulas from an assertion language. These two different approaches lead to different types of completeness results. Thus one should be cautious in translating results from one framework into the other because there can exist subsets of the state space which do not correspond to (are not defined by) any formula from the assertion language. This problem within the relational framework could be resolved by defining a language over the state space in which assertions could be expressed. However, a natural question then arises as to which formulas (subsets) should be accepted as assertions. This problem has been studied in [1].

2. Preliminaries. As in [2] we shall use binary relations over the state space to provide an interpretation for systems of mutually recursive procedures. More precisely, given a set $\mathcal{P}=\{P_1, \cdots, P_n\}$ of procedure symbols, we define a language of "statements" $\mathcal{S}(\mathcal{P})$ as follows: let $\mathcal{A}=\{I, A_1, A_2, \cdots\}$ be a set of "elementary action" symbols, $\mathcal{B}=\{t_1, t_2, \cdots\}$ a set of "Boolean expressions." $\mathcal{S}(\mathcal{P})$ is then the least set containing $\mathcal{A}\cup\mathcal{B}\cup\mathcal{P}$ that is closed under the operations ";" (sequencing) and "$\cup$" (nondeterministic choice).

By a declaration scheme we mean a set $\mathcal{D}=\{P_1\Leftarrow S_1, \cdots, P_n\Leftarrow S_n\}$, where for $i=1, \cdots, n$, $P_i\in\mathcal{P}$, $S_i\in\mathcal{S}(\mathcal{P})$.

In [2] a theory of partial correctness and inductive assertions has been worked out in a relational framework. The meaning of a program is viewed as a *binary relation* over the state space, i.e., a set of pairs of initial and final states, whereas an assertion is viewed as a *subset* of the state space, i.e., the set of states satisfying the assertion. We recall some definitions from [2] which are used below.

Let $\mathcal{V}$ be the domain of states. Letters $R, R_1, \cdots$ denote binary relations over $\mathcal{V}$; p, q, r subsets of $\mathcal{V}$; x, y, z elements of $\mathcal{V}$.

$R_1; R_2=\{(x, y)\colon \exists z[xR_1z \wedge zR_2y]\}$,

$p_+=\{(x, x)\colon x\in p\}$,

$p\circ R=\{y\colon \exists x[x\in p\wedge xRy]\}$,

$\check{R}=\{(x, y)\colon yRx\}$,

Ω denotes the empty set.

Throughout the paper we use the convention from [2] that in any expression involving programs and assertions built up by using ;, $\cup$ or $\subseteq$ we suppress the subscript "$_+$".

So, for example, if we write $p; R\subseteq R; q$ we actually mean $p_+; R\subseteq R; q_+$, i.e., $\forall x, y[(x\in p\wedge xRy)\rightarrow y\in q]$, or (informally speaking) that the program R is partially correct with respect to p and q. We shall need the following results proved in [2].

LEMMA 1.

(i) $(R_1; R_2); R_3=R_1; (R_2; R_3)$ $(=R_1; R_2; R_3$, from now on),

(ii) $R_1; (R_2\cup R_3)=R_1; R_2\cup R_1; R_3$,

(iii) $(R_1\cup R_2); R_3=R_1; R_3\cup R_2; R_3$,

(iv) $p\circ(R_1; R_2)=(p\circ R_1)\circ R_2$.

If $X_1, \cdots, X_n, Y_1, \cdots, Y_n$ are subsets of $\mathcal{V}\times\mathcal{V}$, then by definition $(X_1, \cdots, X_n)\leqq(Y_1, \cdots, Y_n)$ iff $X_i\subseteq Y_i$, for $i=1, \cdots, n$ ($\leqq$ is a partial ordering).

Let $\mathscr{D}=\{P_1\Leftarrow S_1,\cdots,P_n\Leftarrow S_n\}$ be a declaration scheme. By an *interpretation* $i_{\mathscr{D}}$ into a state space $\mathscr{V}$ we mean a mapping from $\mathscr{S}(\mathscr{P})$ into relations over $\mathscr{V}$ such that:

(a) for each $A\in\mathscr{A}$, $i_{\mathscr{D}}(A)$ is a binary relation over $\mathscr{V}$;

(b) $i_{\mathscr{D}}(I)=\{(x,x): x\in\mathscr{V}\}$;

(c) for each $t\in\mathscr{B}$, $i_{\mathscr{D}}(t)$ is a subset of $\mathscr{V}$;

(d) for each $P\in\mathscr{P}$, $i_{\mathscr{D}}(P)$ is a binary relation over $\mathscr{V}$;

(e) $i_{\mathscr{D}}(S_1;S_2)=i_{\mathscr{D}}(S_1);i_{\mathscr{D}}(S_2)$;

(f) $i_{\mathscr{D}}(S_1\cup S_2)=i_{\mathscr{D}}(S_1)\cup i_{\mathscr{D}}(S_2)$;

(g) $(i_{\mathscr{D}}(P_1),\cdots,i_{\mathscr{D}}(P_n))$ is the $\leqq$-least n-tuple such that
$(i_{\mathscr{D}}(P_1),\cdots,i_{\mathscr{D}}(P_n))=(i_{\mathscr{D}}(S_1),\cdots,i_{\mathscr{D}}(S_n))$ holds.

The above definition is the usual denotational semantics of recursive program schemes. Its justification and equivalence with operational semantics is an immediate consequence of the results proved in [2].

Observe, for example, that if $\mathscr{D}=\{P\Leftarrow t_1;t_2\}$, then due to the convention mentioned above $i_{\mathscr{D}}(P)=i_{\mathscr{D}}(t_1)_+;i_{\mathscr{D}}(t_2)_+$.

In the sequel we shall always consider programs with respect to a given declaration scheme. We shall freely identify statements and their interpretations, hoping that no confusion will result from this.

3. Extending the state space. We now want to use the assertions from the extended space $\mathscr{V}\times\mathscr{V}$. In order to do this we have to extend (in an obvious way) several operations from $\mathscr{V}$ into $\mathscr{V}\times\mathscr{V}$. Let a, b denote subsets of $\mathscr{V}\times\mathscr{V}$ used as assertions and σ, τ elements of $\mathscr{V}$ used as a second coordinate of the extended state space. Let $R^{\uparrow}=\{((x,\sigma),(y,\sigma)): xRy\wedge\sigma\in\mathscr{V}\}$ be the extension of a program R to the space $\mathscr{V}\times\mathscr{V}$. The operations ; and $_+$ mentioned above retain their meaning when applied to subsets of $(\mathscr{V}\times\mathscr{V})\times(\mathscr{V}\times\mathscr{V})$ and $\mathscr{V}\times\mathscr{V}$ respectively, so obviously Lemma 1 holds in the case of the extended state space $\mathscr{V}\times\mathscr{V}$. We shall use in the sequel "mixed" expressions involving assertions from $\mathscr{V}\times\mathscr{V}$ and programs from $\mathscr{V}\times\mathscr{V}$. While doing so we shall *always* mean their "extensions" to $(\mathscr{V}\times\mathscr{V})\times(\mathscr{V}\times\mathscr{V})$, which can be obtained by attaching the subscript $_+$ to assertions and the superscript $^{\uparrow}$ to programs. For example, if we write $R_1;a;R_2$, we actually mean $R_1^{\uparrow};a_+;R_2^{\uparrow}$. The reader should convince himself that the convention of omitting brackets (as indicated in Lemma 1) does not lead now to any ambiguities, since $(R_1;R_2)^{\uparrow}=R_1^{\uparrow};R_2^{\uparrow}$.

Observe that $a;R\subseteq R;b$ means that $a_+;R^{\uparrow}\subseteq R^{\uparrow};b_+$, i.e., that

$$\forall x,y,\sigma[((x,\sigma)\in a\wedge xRy)\rightarrow(y,\sigma)\in b],$$

or that the program R is partially correct with respect to a and b.

We shall need the following definition:

$$a(R)=\{(x,\sigma): \exists\tau[\sigma R\tau\wedge(x,\tau)\in a]\}.$$

In the proofs below we shall use Scott induction to prove inclusions between relations on $\mathscr{V}\times\mathscr{V}$.

Scott induction. Let $\mathscr{D}=\{P_1\Leftarrow S_1(P_1,\cdots,P_n),\cdots,P_n\Leftarrow S_n(P_1,\cdots,P_n)\}$ be a declaration scheme. Let $\mathscr{E}_l(X_1,\cdots,X_n)$ and $\mathscr{E}_r(X_1,\cdots,X_n)$ be two expressions built up from assertions from $\mathscr{V}\times\mathscr{V}$ and programs from $\mathscr{V}\times\mathscr{V}$ and formal (place-holding) variables $X_1,\cdots,X_n$ using ; and $\cup$ and let the following two conditions be satisfied:

(i) $\mathscr{E}_l(\Omega,\cdots,\Omega)\subseteq\mathscr{E}_r(\Omega,\cdots,\Omega)$, and

(ii) for each $R_1,\cdots,R_n\subseteq\mathscr{V}\times\mathscr{V}$,
if $\mathscr{E}_l(R_1,\cdots,R_n)\subseteq\mathscr{E}_r(R_1,\cdots,R_n)$
then $\mathscr{E}_l(S_1(R_1,\cdots,R_n),\cdots,S_n(R_1,\cdots,R_n))$
$\subseteq\mathscr{E}_r(S_1(R_1,\cdots,R_n),\cdots,S_n(R_1,\cdots,R_n))$.

Then $\mathscr{E}_l(P_1, \cdots, P_n) \subseteq \mathscr{E}_r(P_1, \cdots, P_n)$.
The proof is analogous to the proof of the version formulated in [2].

4. Completeness result. The general context-free declaration scheme is

$$\{P_i \Leftarrow S_{i,1} \cup S_{i,2} \cup \cdots \cup S_{i,M_i}\}_{i=1}^{n}, \tag{1}$$

with M_i some integer $\geqq 1$, and each $S_{i,j}$, $j = 1, \cdots, M_i$, of the form

$$S_{i,j} = A(i,j,0); P(i,j,1); \cdots; A(i,j,K_{i,j}-1); P(i,j,K_{i,j}); A(i,j,K_{i,j}),$$

where $A(i,j,k) \in \mathscr{A} \cup \mathscr{B}$, $P(i,j,k) \in \{P_1, \cdots, P_n\}$, and $K_{i,j}$ is an integer $\geqq 0$ (if $K_{i,j} = 0$, then $S_{i,j}$ is simply $A(i,j,0)$).

In the above declaration scheme each $P(i,j,k)$ is some element of $\{P_1, \cdots, P_n\}$. Define a function h by: $h(i,j,k) = l$ iff $P(i,j,k) = P_l$.

The general inductive assertion method calls for suitable intermediate assertions preceding and succeeding each statement in the program. The theorem presented below states soundness and completeness of a particular version of the method in which intermediate assertions from the extended state space are used. The theorem shows that the global correctness property $p; P_1 \subseteq P_1; q$ can always be established by finding intermediate assertions of the special form a^i, $a(i,j,k)$, b^i and $b(i,j,k)$.

THEOREM. *Assume the declaration scheme* (1). *For any two assertions $p, q \subseteq \mathscr{V}$,*

$$p; P_1 \subseteq P_1; q$$

iff there exist assertions $a^i, b^i \subseteq \mathscr{V} \times \mathscr{V}$ ($i \in \{1, \cdots, n\}$) and relations $R_{i,j,k} \subseteq \mathscr{V} \times \mathscr{V}$ ($i \in \{1, \cdots, n\}$, $j \in \{1, \cdots, M_i\}$ and $k \in \{1, \cdots, K_{i,j}\}$) such that for all $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, M_i\}$,

$$\begin{array}{lr} a^i; A(i,j,0) \subseteq A(i,j,0); b^i & \textit{if } K_{i,j} = 0, \\ \left.\begin{array}{l} a^i; A(i,j,0) \subseteq A(i,j,0); a(i,j,1), \\ b(i,j,k); A(i,j,k) \subseteq A(i,j,k); a(i,j,k+1), \quad k = 1, \cdots, K_{i,j}-1 \\ b(i,j,K_{i,j}); A(i,j,K_{i,j}) \subseteq A(i,j,K_{i,j}); b^i, \end{array}\right\} & \textit{if } K_{i,j} > 0, \end{array} \tag{2}$$

and

$$\begin{aligned} &I \cap (p \times p) \subseteq a^1, \\ &b^1 \cap (\mathscr{V} \times p) \subseteq q \times p. \end{aligned} \tag{3}$$

Here by definition $a(i,j,k) = a^{h(i,j,k)}(R_{i,j,k})$ and $b(i,j,k) = b^{h(i,j,k)}(R_{i,j,k})$.

Proof. To make the argument more readable we shall prove the theorem in the case of the declaration $P \Leftarrow A_1; P; A_2; P; A_3 \cup A_4$. The proof for the case of the general context-free declaration scheme is analogous and we leave it to the reader. We thus prove the following.

Assume the declaration $P \Leftarrow A_1; P; A_2; P; A_3 \cup A_4$. For any two assertions $p, q \subseteq \mathscr{V}$,

$$p; P \subseteq P; q,$$

iff there exist assertions $a, b \subseteq \mathscr{V} \times \mathscr{V}$ and relations $R_1, R_2 \subseteq \mathscr{V} \times \mathscr{V}$ such that

$$\begin{aligned} &a; A_1 \subseteq A_1; a(R_1), \\ &b(R_1); A_2 \subseteq A_2; a(R_2), \\ &b(R_2); A_3 \subseteq A_3; b, \\ &a; A_4 \subseteq A_4; b, \end{aligned} \tag{4}$$

and

(5)
$$I\cap(p\times p)\subseteq a,$$
$$b\cap(\mathscr{V}\times p)\subseteq q\times p.$$

If part. We first prove by Scott induction that

(6)
$$a\,;P\subseteq P;\,b.$$

Assume that $a\,;X\subseteq X;\,b$ for some $X\subseteq\mathscr{V}\times\mathscr{V}$, i.e., that

$$\forall x,\,y,\,\sigma[((x,\sigma)\in a\wedge xXy)\rightarrow(y,\sigma)\in b].$$

Thus for any relation R,

$$\forall x,\,y,\,\sigma,\,\tau[\sigma R\tau\wedge(x,\tau)\in a\wedge xXy\rightarrow(y,\tau)\in b],$$

i.e., according to our notation,

(7)
$$a(R);X\subseteq X;\,b(R).$$

Now, due to the assumptions, Lemma 1 and (7),

$$\begin{aligned}a\,;(A_1;X;A_2;X;A_3)&=(a\,;A_1);X;A_2;X;A_3\subseteq A_1;a(R_1);X;A_2;X;A_3\\&\subseteq A_1;X;b(R_1);A_2;X;A_3\subseteq A_1;X;A_2;a(R_2);X;A_3\subseteq A_1;X;b(R_2);A_3\\&\subseteq(A_1;X;A_2;X;A_3);\,b.\end{aligned}$$

Hence, by Lemma 1 and the assumptions,

$$a\,;(A_1;X;A_2;X;A_3\cup A_4)\subseteq(A_1;X;A_2;X;A_3\cup A_4);\,b.$$

Since obviously $a\,;\Omega\subseteq\Omega;\,b$, by Scott induction, (6) holds.

We are now ready to prove $p;\ P\subseteq P;\ q$. Suppose that $x\in p$ and xPy for some $x,\,y\in\mathscr{V}$. We have to show: $y\in q$. By the assumptions $(x,x)\in a$. By (6), $(y,x)\in b$. Since $x\in p$, by the assumptions $(y,x)\in q\times p$, so $y\in q$.

Only if part. Put $a=I$, $b=\check{P}$ and let $R_1=A_1$ and $R_2=A_1;P;A_2$. We are to prove that (4) and (5) hold.

Let $x,\,y,\,\sigma$ be arbitrary elements of $\mathscr{V}$.

(i) We have to show: $a\,;A_1\subseteq A_1;\,a(R_1)$, i.e., $(x,\sigma)\in a$ and xA_1y implies $(y,\sigma)\in a(R_1)$, which is equivalent to $\exists\tau[\sigma R_1\tau\wedge(y,\tau)\in a]$. Suppose $(x,\sigma)\in a$ and xA_1y. By the definition of a, $\sigma=x$, so by the definition of R_1, σR_1y. Hence, since $(y,y)\in a$, we get $\exists\tau[\sigma R_1\tau\wedge(y,\tau)\in a]$ by putting $\tau=y$.

(ii) We have to show: $b(R_1);A_2\subseteq A_2;\,a(R_2)$, i.e., $\exists\tau[\sigma R_1\tau\wedge(x,\tau)\in b]$ and xA_2y implies $\exists\tau_1[\sigma R_2\tau_1\wedge(y,\tau_1)\in a]$. Suppose that for some $\tau,\sigma R_1\tau$, $(x,\tau)\in b$ and xA_2y. By the definition of R_1 and $b,\sigma A_1\tau$ and τPx, so $\sigma(A_1;\,P;\,A_2)y$. By the definition of $R_2,\sigma R_2y$, so, since $(y,y)\in a$, we get $\exists\tau_1[\sigma R_2\tau_1\wedge(y,\tau_1)\in a]$ by putting $\tau_1=y$.

(iii) We have to show: $b(R_2);A_3\subseteq A_3;\,b$, i.e., $\exists\tau[\sigma R_2\tau\wedge(x,\tau)\in b]$ and xA_3y implies $(y,\sigma)\in b$. Suppose that for some $\tau,\sigma R_2\tau,(x,\tau)\in b$ and xA_3y. By the definition of R_2 and b, $\sigma(A_1;\,P;\,A_2)\tau$ and τPx, so $\sigma(A_1;\,P;\,A_2;\,P;\,A_3)y$. Thus, σPy, which means $(y,\sigma)\in b$.

(iv) We have to show: $a;\ A_4\subseteq A_4;\ b$, i.e., $(x,\sigma)\in a$ and xA_4y implies $(y,\sigma)\in b$. Suppose $(x,\sigma)\in a$ and xA_4y. Then $\sigma=x$ and xPy, *i.e.*, $(y,\sigma)\in b$.

(v) Obviously $I\cap(p\times p)\subseteq a$.

(vi) We have to show: $b\cap(\mathscr{V}\times p)\subseteq q\times p$, i.e., $(x,y)\in b$ and $y\in p$ implies $x\in q$. Suppose $(x,y)\in b$ and $y\in p$. Then yPx, and since $p;P\subseteq P;\ q$, we find $x\in q$. This concludes the proof.

The above proof is an analogue of the corresponding completeness proofs in [5] and [6]. However, the relational approach sheds some light on the role of the auxiliary variables used in [5] to obtain so-called "most general formulas" and in [6], analogously, to "freeze" the global variables upon entering a procedure call. It is clear from the above proof that completeness is obtained by using the *meaning* of a procedure as an assertion.

The proof also suggests an alternative, equivalent point of view at the way of introducing the extended state space. Namely, the same result can be obtained by proving first partial correctness of the program $\sigma := x; P$ using assertions from its state space. The condition $I \cap (p \times p) \subseteq a$ is then replaced by the equivalent requirement $p \times \mathcal{V}; \sigma := x \subseteq \sigma := x; a$.

In such a way the extension of the state space is caused by a change in the original program P. The desired global correctness property is then derived by deleting the assignment $\sigma := x$ to the "auxiliary variable" σ using the corresponding proof rule from [8].

5. An application. Having obtained a specific form of the completeness result we shall illustrate its usefulness by the following example.

Let the state space $\mathcal{V}$ be the set of natural numbers $\mathcal{N}$. Consider the following declaration:

$$P \Leftarrow [n \leqq 100]; [n := n + 11]; P; P \cup [n > 100]; [n := n - 10], \tag{8}$$

where, of course, $[n \leqq 100] = \{x: x \leqq 100\}$, $[n := n + 11] = \{(x, y): y = x + 11\}$ and so on. P is of course McCarthy's well-known 91 function defined in a relational framework. We want to prove that

$$[n \leqq 100]; P \subseteq P; [n = 91]. \tag{9}$$

Observe that the above declaration is of the form $P \Leftarrow A_1; P; A_2; P; A_3 \cup A_4$, where

$$\begin{aligned} A_1 &= [n \leqq 100]; [n := n + 11], \\ A_2 &= I, \\ A_3 &= I, \\ A_4 &= [n > 100]; [n := n - 10]. \end{aligned}$$

We can now use the theorem to prove (9). The easiest way to proceed is to define the required relations and functions as in the proof of the theorem, taking for P $[n \leqq 100]$; $[n := 91] \cup [n > 100]; [n := n - 10]$, and to check that (4) and (5) hold.

Thus we define

$$\begin{aligned} a &= \{(x, x): x \in \mathcal{N}\}, \\ b &= \{(x, y): (x = 91 \wedge y \leqq 100) \vee (x = y - 10 \wedge y > 100)\}, \\ R_1 &= \{(x, y): x \leqq 100 \wedge y = x + 11\}, \\ R_2 &= [n \leqq 100]; [n := n + 11]; ([n \leqq 100]; [n := 91] \cup [n > 100]; [n := n - 10]) \\ &= \{(x, y): (90 \leqq x \leqq 100 \wedge y = x + 1) \vee (x < 90 \wedge y = 91)\}. \end{aligned}$$

We leave the task of checking that (4) and (5) indeed hold to the reader. Now, by the theorem, (9) holds.

The above program together with the corresponding assertions can be represented by the flow-chart (Fig. 1).

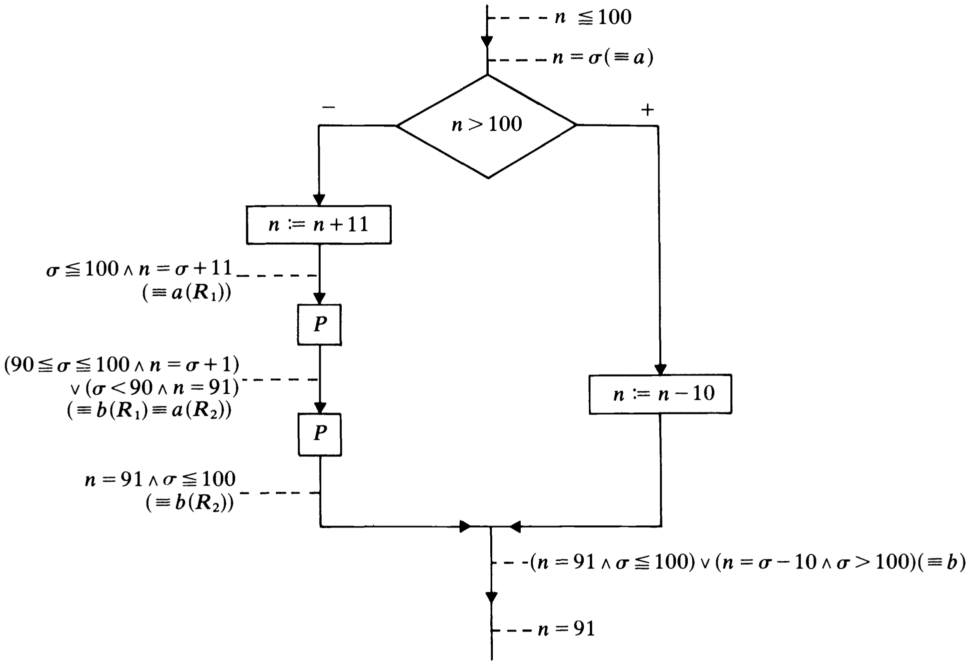

$n \leqq 100$
$n = \sigma (\equiv a)$
$n > 100$
$-$
$+$
$n := n + 11$
$\sigma \leqq 100 \wedge n = \sigma + 11$
$(\equiv a(R_1))$
P
$(90 \leqq \sigma \leqq 100 \wedge n = \sigma + 1)$
$\vee (\sigma < 90 \wedge n = 91)$
$(\equiv b(R_1) \equiv a(R_2))$
P
$n := n - 10$
$n = 91 \wedge \sigma \leqq 100$
$(\equiv b(R_2))$
$(n = 91 \wedge \sigma \leqq 100) \vee (n = \sigma - 10 \wedge \sigma > 100)(\equiv b)$
$n = 91$

FIG. 1

Acknowledgments. We are grateful to Prof. A. Pnueli who refereed the paper and suggested various improvements, in particular the present, stronger version of the completeness result. We also thank Prof. J. W. de Bakker for critical comments on an earlier version.

REFERENCES

[1] K. R. APT, J. A. BERGSTRA AND L. G. L. T. MEERTENS, *Recursive assertions are not enough—or are they?*, Theor. Comput. Sci., 8 (1979), pp. 73–87.

[2] J. W. DE BAKKER AND L. G. L. T. MEERTENS, *On the completeness of the inductive assertion method*, J. Comput. System Sci., 11 (1975), pp. 323–357.

[3] S. A. COOK, *Soundness and completeness of an axiom system for program verification*, this Journal, 7 (1978), pp. 70–90.

[4] J. H. GALLIER, *Semantics and correctness of nondeterministic flowchart programs with recursive procedures*, Proc. 5th Coll. Automata, Languages and Programming, Lecture Notes in Computer Science, No. 62, Springer-Verlag, 1978, pp. 251–267.

[5] G. A. GORELICK, *A complete axiomatic system for proving assertions about recursive and nonrecursive programs*, Technical Report No. 75, University of Toronto (1975).

[6] D. HAREL, A. PNUELI AND J. STAVI, *Completeness issues for inductive assertions and Hoare's method*, Technical Report, Tel-Aviv University, 1976.

[7] C. A. R. HOARE, *An axiomatic basis for programming language constructs*, Comm. ACM, 12 (1969), pp. 576–580.

[8] S. OWICKI AND D. GRIES, *An axiomatic proof technique for parallel programs I*, Acta Informatica, 6 (1976), pp. 319–340.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0002 $01.00/0

A NEW PROOF OF THE LINEARITY OF THE BOYER-MOORE STRING SEARCHING ALGORITHM*

LEO J. GUIBAS† AND ANDREW M. ODLYZKO‡

Abstract. The Boyer-Moore algorithm searches for all occurrences of a specified string, the *pattern*, in another string, the *text*. We study the combinatorial structure of periodic strings and use these results to derive a new proof of the linearity of the Boyer-Moore algorithm in the worst case. Our proof reduces the previously best known bound of $7n$ to $4n$, where n is the length of the text.

Key words. String searching, pattern matching, period, algorithmic analysis

1. Introduction. In this paper we analyze the worst case performance of the Boyer-Moore string searching algorithm [2]. We work within the paradigm of algorithmic analysis initiated by Knuth, that is, we develop a mathematical theory dealing with the behavior of a *specific* (and practical) algorithm. Other algorithms whose analysis has resulted in significant mathematical developments are the Union-Find algorithm [10], QuickSort [9], Double Hashing [3], and others. For a more general treatment of algorithmic analysis of combinatorial problems see [1], [5], [6].

The Boyer-Moore algorithm is the best currently known algorithm for finding an (all) occurrence(s) of a *pattern* (a string) in a *text* (another string), or deciding that none exist. We measure the cost of a string searching algorithm by the number of comparisons performed between characters of the pattern and characters of the text. In typical cases this algorithm exhibits sublinear performance, i.e., its cost equals only a fraction of the characters of the text, as evinced by empirical data, as well as theoretical analysis [2]. In fact, on the average, the algorithm performs better the longer the pattern gets.

In a worst case analysis, however, the situation is different. We will see in the next section that between successive attempts to match the pattern against the text, the Boyer-Moore algorithm completely forgets any information it may have gathered. Thus compared to its closest competitor, the Knuth-Morris-Pratt (KMP) algorithm [7], Boyer-Moore stands in a peculiar relationship. While Boyer-Moore is clearly superior on the average, the simple argument that proves that the cost of KMP is bounded by $2N$ in the worst case, where N is the length of the text, no longer works. Since Boyer-Moore "forgets", a proof of its linearity is nontrivial. It turns out that in order to prove linearity we must restrict ourselves to the case where the pattern does not appear in the text (see the discussion in [7]; Galil [11] has recently shown how to modify the algorithm so as to remove this restriction). For that case, Knuth [7] showed by a complicated argument that the number of comparisons performed by the algorithm is never more than $7N$. The complexity and terseness of Knuth's argument, however, has made his proof difficult to understand. He conjectured that a simpler proof could be found leading to a better bound.

The main result of this paper is a new proof of the linearity of the Boyer-Moore algorithm. We have paid close attention to details and to clarity of presentation. We have also improved the worst case bound from $7N$ to $4N$. In the process we have developed considerable combinatorial machinery dealing with the occurrence of periods in strings, much of it of interest in its own right. Undoubtedly the same or similar

* Received by the editors November 30, 1977, and in revised form September 24, 1979.

† Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

‡ Bell Telephone Laboratories, Murray Hill, New Jersey 07974.

machinery will come in handy in the analysis of other questions concerning pattern matching. We consider it probable that the true worst case bound for the algorithm is $2N$.

2. The Boyer-Moore string searching algorithm. The string searching problem is the following. Given an array *text*$[1{:}N]$ representing the input text, and an array *pattern*$[1{:}M]$ representing the pattern being sought, find the *first* (e.g., leftmost) occurrence of the pattern in the text, if one exists.

The Boyer-Moore algorithm solves this problem by repeatedly positioning the pattern over the text and attempting to match it. For each positioning that arises, the algorithm starts matching the pattern against the text from the *right* end of the pattern. If no mismatch occurs, then the pattern has been found. Otherwise the algorithm computes a *shift*, that is, an amount by which the pattern will be moved to the right before a new match attempt is undertaken.

In the program below we keep two pointers, one (J) to the current character of the *pattern* being examined, and the other (K) to the corresponding character of the *text*. Thus at that instant characters $J+1$ through M of the pattern are aligned with positions $K+1$ through $K-J+M$ of the text.

```
    begin "Boyer-Moore"
    $K \leftarrow M$;
    while $K \leqq N$ do
    begin comment position pattern pointer at right end;
    $J \leftarrow M$;
    while $J > 0$ and $text[K] = pattern[J]$ do
         begin comment move left as long as they match;
         $J \leftarrow J-1$;  $K \leftarrow K-1$;
         end;
    if $J = 0$ then
         begin
         $match\_found\_at\ (K)$;
         done "Boyer-Moore"
         end
    else comment mismatch! now shift and repeat;
         $K \leftarrow K+M-J+s(text[K], J)$;
    end;
$match\_not\_found$;
end "Boyer-Moore".
```

The crux of the algorithm lies in how the shift $s(text[K], J)$ is computed. In order to come as close as possible to maximizing the shift s, the algorithm uses two heuristics. The *match* heuristic is based on the idea that when the pattern is moved right it has to (1) match over all the characters previously matched, and (2) bring a *different* character over the character of the text that caused the mismatch. Thus the *match shift* is defined by

$$s.match(J) = \min\{T \mid T \geqq 1 \text{ and } (T \geqq J \text{ or } pattern[J-T] \neq pattern[J])$$
$$\text{and } ((T \geqq I \text{ or } pattern[I-T] = pattern[I]) \text{ for } J < I \leqq M)\}.$$

Note that *s.match* is only a function of the *pattern* and J, the location of the current mismatch. Secondly, the *occurrence* heuristic uses the fact that we must shift far enough to the right to bring over $CH = text[K]$ (the character that caused the mismatch), the first character of the pattern that will match it. Thus the *occurrence shift* is defined by

$$s.occ(CH) = \min\{T - M + J \mid T = M \text{ or } (0 \leqq T < M \text{ and } pattern[M - T] = CH)\}.$$

Note that *s.occ* depends only on the pattern and the mismatching character $text[K]$. Thus both shifts can be obtained from *precomputed* tables, based solely on the pattern and the alphabet used. The match heuristic requires a table of length equal to the pattern length, while the occurrence heuristic requires a table of size equal to the alphabet size. Given these two shifts, the Boyer-Moore algorithm chooses the largest one. Thus s is defined by

$$s(text[K]\, J) = \max\{s.match(J), s.occ(text[K])\}.$$

(Note that *s.occ* can be negative, but *s.match* is always at least 1.)

The key difference between the Boyer-Moore and KMP algorithms is that the former matches the pattern in the reverse direction from the direction in which the pattern is shifted. This typically allows larger shifts, as the information derived from a partial match is further to the right. Otherwise the definition of *s.match* is exactly analogous to that in KMP. The occurrence heuristic is most valuable when the alphabet size is large. In fact there is a subtle interplay between the two heuristics that considerably complicates the analysis of the algorithm. Although the choice of the maximum of the two shifts is locally optimal, there are examples in which it can cause worse performance in the long run than when consistently using one of the two heuristics alone [7].

In §3 we introduce most of our notation and prove a few basic lemmas about the occurrence of periods in strings. In § 4 we develop some fundamental properties of the algorithm. § 5 contains the key idea of the proof, that is the notion of making the shift of some later match "responsible" for the current one. § 6 discusses consequences of our definition of responsibility, and finally § 7 completes the accounting for the cost of the algorithm.

3. Matches and periods. The study of the Boyer-Moore algorithm requires that we devote some attention to the occurrence of periods in strings. Given a string x, we will say that suffix y of x is a *period* of x, if $x = \tilde{y}(y)^k$, where $\tilde{y}$ is a suffix of y. Equivalently, suffix y is a period of x, if a second copy of x placed under the original x, but shifted left $|y|$ places, matches the original. (We have chosen to align periods with the right end of x, since the algorithm matches the pattern from the right; our strings *begin* to the *right* and *end* to the *left*). For instance, "aaba" is a period of "baaabaaaba." For convenience of notation, in what follows we will use small letters x, y, m, r, to stand either for strings or for their lengths, as the context may require. If period (string) p is an exact multiple of period (string) q, then q will be called a *refinement* of p.

Given a string x, we will denote by $p(x)$ the shortest period of x, also called the *basic period* of x. The combinatorial structure of the periods of strings has a rich theory, which is investigated in detail in [4]. A corollary of this theory, that has been independently known for quite some time [7], [8], is the following useful lemma.

LEMMA 3.1 (GCD Rule). *If p and q are periods of x, and $p + q \leqq x + \gcd(p, q)$, then $\gcd(p, q)$ is also a period of x.* □

An easy consequence of the GCD Rule is the following result.

LEMMA 3.2 (Common Refinement). *If $x = yz$, where in addition y is a suffix of x and z is a prefix of x, then there exists a smallest common refinement a of x, y, z, such that for some nonnegative integers i, j we have $y = a^i$, $z = a^j$, $x = a^{i+j}$.* □

Proof. Both z and y are periods of x. Since $z+y=x \leqq x+\gcd(z, y)$, it follows from the GCD rule that $\alpha=\gcd(z, y)$ is the desired refinement of x, y, z. □

We now define some terminology relevant to the matches that arise during the algorithm. The algorithm proceeds in *stages*. At any stage the pattern is matched from the right until a *mismatch* is encountered. Let m denote the suffix of the pattern examined during a given stage. Such an m will be called a *match*. We will really think of the match m as an event. Associated with m are various attributes, such as the suffix m (by abuse of language), and others discussed below. The leftmost character of m caused the mismatch with the text. We use m' to denote m with the leftmost character removed. By the *right half* of m we will mean the suffix consisting of the rightmost $\lceil m/2 \rceil$ characters of m. At any stage the algorithm performs a *shift*, denoted from now on by $s(m)$, as described in the previous section. In writing $s(m)$ we indicate that the shift is another attribute of the event m. In a similar vein we will write *s.match*(m) and *s.occ*(m).

DEFINITION 3.1 (Major Match). A match m will be called a *major* match if

(1) $p(m)>p(m')$ (e.g , the mismatch of m destroys the basic period of m'),

and

(2) $m>2p(m')$. □

A match which is not major will be called *minor*. The following lemma asserts that a major match has a long basic period.

LEMMA 3.3 (Two Periods). *If m is a major match, then*

(1) $p(m)>\frac{1}{2}m>p(m')$,

(2) $p(m)+p(m')\geqq m$. □

Proof. Note that $p(m)$ is also a period of m'. Since the mismatch of m destroys the period of m', we cannot have $p(m')|p(m)$. Therefore by the GCD rule

$$p(m)+p(m')>m-1+\gcd(p(m), p(m'))\geqq m.$$

Now $p(m)>p(m')$ implies

$$p(m)>\tfrac{1}{2}m>p(m'). \quad □$$

In thinking of m as an event, we can also let it denote a consecutive set of positions over the text, that is the set of positions where the match m occurred during the execution of the algorithm. Similarly, $p(m)$ will also denote the set of positions over the text occupied by the rightmost occurrence of the period of match m. And $s(m)$ will denote the set of $s(m)$ positions over the text, immediately to the right of m. For example, in the next section we make the following assertion for matches m, r (r occurring later than m): "$p(r')$ does not end under $s(m)$." The detailed meaning of this is "the leftmost character of the rightmost occurrence of the basic period of r', as it occurs in r', cannot be in any of the $s(m)$ positions immediately to the right of match m."

We will use pictures to illustrate some of the situations that can arise. In these pictures a heavy black line will always indicate a match. Certain characters will be used with special meaning: "]" will denote the beginning of a match m; "(" will denote the leftmost character of $p(m')$, and "|" the midpoint of m; "X" will denote the mismatch of m; and "⟩" will denote the rightmost position of $s(m)$, i.e., the first character of the successor match to m. Fig. S3 illustrates how we imagine the execution of the algorithm. Note that a match r, occurring later than match m, is drawn below m in the picture. Language influenced by this graphical representation was implicit in the use of the word "under," in the phrase discussed in the above paragraph. At the very bottom of the picture we imagine the text, against which the pattern is matched.

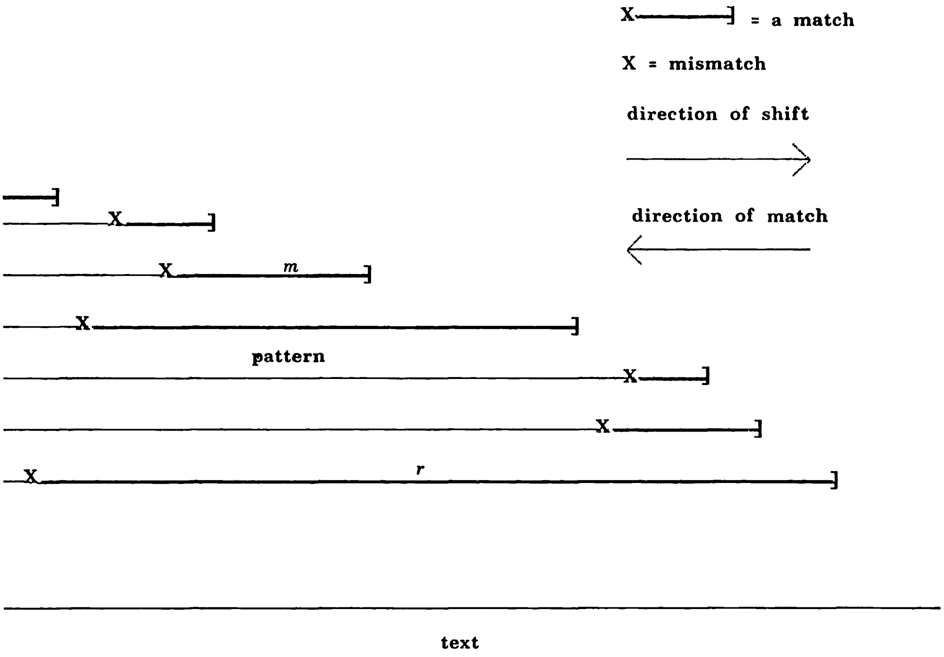


FIG. S3. *A window over the text depicting the history of the algorithm.*

4. Properties of the algorithm. The following two lemmas are trivial consequences of the definitions of the shifts considered by Boyer-Moore. They are listed here mostly for reference.

LEMMA 4.1 (Shift ≧ Period). *For any match m we have*

$$s(m) \geqq s.match(m) \geqq p(m). \quad \square$$

LEMMA 4.2 (Match Successor). *Let r denote the successor match to m. If r was chosen via the match heuristic, then r does not mismatch under m'.* □ (See Fig. L4.2)

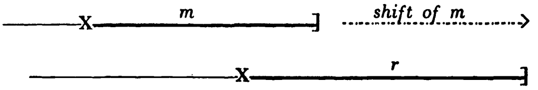


FIG. L4.2. *The above situation cannot arise.*

Lemmas 4.3 and 4.4 are more interesting. The first one indicates a significant constraint on matches that completely overlap.

LEMMA 4.3 (Period under Shift). *If match r occurs later than match m during the execution of the algorithm, and furthermore r goes strictly further left than m, then p(r') does not end under s(m).* □ (See Fig. L4.3)

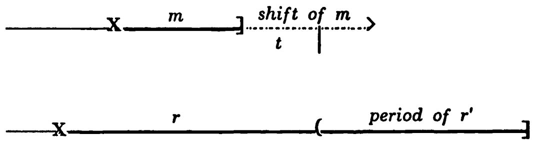


FIG. L4.3. *The above situation cannot arise.*

Proof. After the mismatch at m, consider the shift that would position the rightmost character of the pattern immediately to the left of $p(r')$. Call this shift t. We will obtain a contradiction by showing that $t, t<s(m)$, is a shift the algorithm could not have rejected. A shift of t satisfies the match heuristic since $p(r')$ is a period of r', a suffix of the pattern, which persists under m. Furthermore, since r goes strictly further left than m, the character brought under the mismatch of m by shift t matches the text, and thus it is different from the mismatching character of m. Therefore shift t is also acceptable to the occurrence heuristic. This completes the argument. □

If we compelled the string searching algorithm to always use the match heuristic, then a simplified version of the argument of this paper can be used to prove the same $4N$ worst case bound. The presence of the occurrence heuristic complicates matters and gives rise to the "lock up" phenomenon discussed in the next section. Lemma 4.4 is our main tool in dealing with the complications introduced by the occurrence heuristic.

LEMMA 4.4 (*s.occ* > *s.match*). *If for some match m, s.occ*(m) > *s.match*(m), *then the character in the text that causes m to mismatch does not appear anywhere in m.* □

The proof of the above lemma under our definition of the occurrence heuristic is entirely trivial. Note that the assertion of the lemma remains true under a strengthened form of the occurrence heuristic, in which we demand a shift that will bring over the mismatching text character the first matching character of the pattern *to the left of the current mismatch.*

Proof. Call Z the mismatching character of the text. It suffices to show that Z does not appear in $p(m)$. Consider $t = \{$the leftmost $p(m)$ characters of $m\}$. Then t is a cyclic shift of $p(m)$ and the leftmost character of t is not Z. Since *s.match*$(m) \geqq p(m)$, a copy of the rightmost $p(m)-1$ characters of t must occur in the pattern, immediately to the left of m. If Z was among them, then *s.match*(m) > *s.occ*(m), a contradiction. (An appropriate interpretation of this argument must be given when the above copy of t has partially fallen off the end of the pattern.)

5. The definition of responsibility. Note that the sum of all shifts made by the algorithm is bounded by the length of the text. Thus if we could prove that the length of each match is bounded by some constant multiple of the corresponding shift, we would have proved the linearity of the algorithm. Unfortunately it is easy to see that such a constant does not exist. Some form of global accounting is essential for a linearity proof. Any local argument must face the difficulty that individual characters of the text may be matched as many as $\Omega(\log M)$ times [7].

The idea of our proof is to let *future* shifts pay for the current match. The following two definitions identify those future matches r, whose shift will be eligible to bear the cost of the current match m.

DEFINITION 5.1 (Cover up). Match r, occurring later than match m, is said to *cover* m, if r goes strictly further to the left than m. □ (See Fig. D5.1)

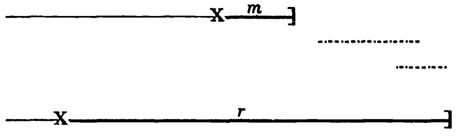


FIG. D5.1. *Cover up.*

DEFINITION 5.2 (Lock up). Match r, occurring later than match m, is said to *lock* with m, if the following four conditions hold:

(1) r does not extend to the left of m,

(2) m extends over the right half of r, and there exists a smallest refinement period p of $s.match(m)$ such that
(3) r covers p, and
(4) the part of r to the right of m is composed of an integral number of repetitions of p. □ (See Fig. D5.2)

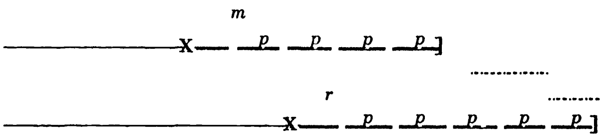


FIG. D5.2. *Lock up.*

[Observe that requirements (1) and (2) imply that $s.match(m) \leqq m$.]

Cover up and lock up are mutually exclusive. Part of the cost of a match m will be borne by its own shift $s(m)$, and part will be charged to the shift of some future match r. The details of how this is to be done are given in the two definitions below.

DEFINITION 5.3 (Charge). If m is a match, the *charge* $c(m)$ is defined to be the rightmost two occurrences of $p(m')$ in m if m is a major match, and all of m otherwise. We write $d(m)$ for what is left after $c(m)$ is taken out of m. Thus $c(m)+d(m)=m$. □

DEFINITION 5.4 (Allocation of Responsibility). The cost of a match m will be borne partially by the shift of m itself, and partially by the shift of some future match r. The match r is defined to be the first match following m which either covers m or locks with m. If no such r exists, then we will say that m goes on welfare. (We can usefully think of welfare as referring to a fictitious match at the very end of the algorithm, whose shift is the entire text).

We will say that m is *charged* to, or *assigned* to match r defined above, or that r is *responsible for* m, and denote this relation by $m \to r$. The cost of m will then be allocated as follows: $d(m)$ will be borne by $s(m)$, and $c(m)$ will be borne by $s(r)$. □

We must next prove that the various shifts are sufficiently long to pay for the portions of the various matches charged to them.

6. Analysis of responsibility. In this section we gather together a set of useful lemmas regarding the notion of responsibility introduced in the previous section. Using these lemmas, a complete accounting for the performance of the algorithm is presented in the following section.

LEMMA 6.1 (First Responsible via Cover up). *If, among all matches that have been charged to r via cover up, m is the earliest one (in the history of the algorithm), then either $p(r')$ covers the right half of m, or m is a major match and $p(r')$ covers $p(m')$* □ (See Fig. L6.1)

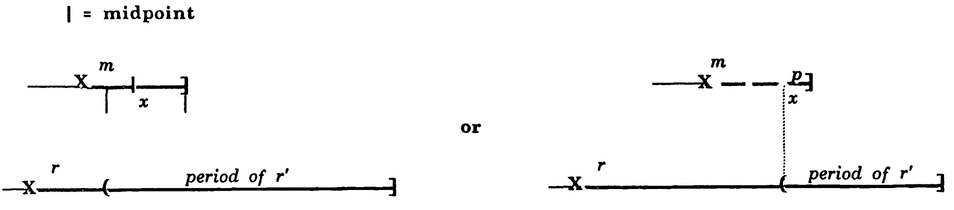


FIG. L6.1. *First responsible via cover up.*

Proof. Suppose that $p(r')$ does not cover the right half of m. Let x denote the portion of m covered by $p(r')$. Then we have $x < m/2$. Since r extends further left than m and $p(r')$ is a period of r', it follows that x is a period of m'. Thus we have $2p(m') \leqq 2x < m$. Further, the leftmost character of m which mismatches must obviously destroy period $p(m')$ and thus $p(m) > p(m')$ and m is major. Finally $p(r')$ obviously covers $p(m')$. □

The next lemma is a very important one. It states that matches charged to the same match cannot overlap too much.

LEMMA 6.2. (Right Halves Disjoint). *If m and t are two matches such that* (1) *t occurs after m and,* (2) *m is charged to a match after t, then the right halves of m and t are disjoint.* □ (See Fig. L6.2)

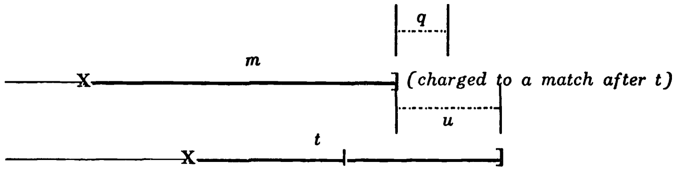


FIG. L6.2. *The above situation cannot arise.*

Proof. Suppose that m overlaps the right half of t. We will assume also that match t does not cover up match m. From this we will conclude that m locks up with t, thus obtaining a contradiction. Requirement (2) of the lock up definition is satisfied by assumption. Since we assume no cover up, requirement (1) is also met. Let $q = s.match(m)$. Let u denote the shift required to go from m to t. Since m overlaps the right half of t, we must have

$$1 \leqq q \leqq u < m.$$

As a portion u of t is to the right of m, t must extend by at least q under m.

Recall that q is a period of m and, by the definition of the match heuristic, q persists through suffix v of the pattern which is either the entire pattern, or satisfies $v \geqq m + q - 1$. We can easily check that in either case we have $v \geqq u + q$. It follows that if we start at the right end of t moving left, then q will persist at least through the first q characters of t under m. But the rightmost q characters of m form q itself so, by the transitivity of matching, the common refinement lemma applies. Thus there exists a smallest refinement period p which exactly divides u. This is the period we need for the lock up definition. Clearly t covers the rightmost occurrence of p in m, and the part of t to the right of m consists of an integral number of repetitions of p. □

As the reader might have expected from the definition in the previous section, a lock up situation is very highly constrained. The following two lemmas shed further light on how it can arise.

LEMMA 6.3 (Single Lock up). *Given a match r, at most one match m can be charged to it via lock up. If m was so charged to r, then m must be the earliest match charged to r during the execution of the algorithm.* □ (See Fig. L6.3)

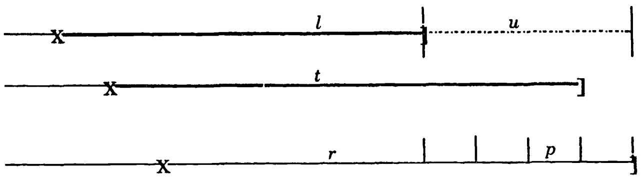


FIG. L6.3. *At most one match charged via lock up.*

Proof. Assume we have at least one match assigned to r via lock up, else the lemma is vacuously true. Let l be the earliest match assigned to r via lock up. It is clear that no match occurring earlier than l can be assigned to r via cover up. This is so since such a match is also covered by l, and l occurs before r. Now suppose there was a second match t assigned to r via lock up. Let p be the period of l referred to in the lock up definition. Let u be the part of r to the right of l.

Since r covers the rightmost p in l, and t locks up with r, it follows that $t \geqq p$. Moreover, we have $r \geqq 2p$, and u is composed of an integral number of p's. Since p is minimal, it follows from the common refinement lemma that t must start lined up with one of the occurrences of p in u.

Note that

(1) t does not extend to the left of l, or else l would have been assigned to t via cover up;

(2) the right half of t extends further left than that of r (if the endpoints on an interval are moved left, the midpoint is too) and so l extends over the right half of t;

(3) t covers the rightmost occurrence of p in l, since t extends to the left of r; and

(4) the part of t to the left of l is composed of an integral number of repetitions of p.

Thus l locks up with t, a contradiction.

This proves that among all matches assigned to r none can precede l, and those following l were assigned via cover up. □

LEMMA 6.4 (Lock up Analysis). *Suppose $m \rightarrow r$ via lock up. Then*

(1) *$r \geqq m + p$, with p as in the lock up definition,*

(2) *r is a major match, and*

(3) *either r is the successor of m, or $s(r) \geqq r$.* □ (See Fig. L6.4)

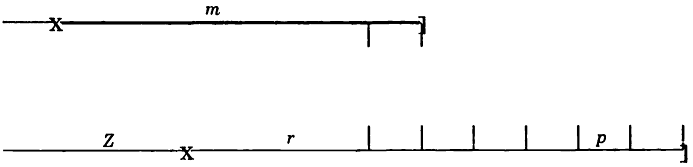


FIG. L6.4. *Lock up analysis; r not successor of m.*

Proof. Period p of m persists for at least $p-1$ characters to the left of m, or else the pattern matches the text at r, a contradiction. It follows that $r \geqq m + p$ and (1) is proved.

Note that $p = p(r')$ since by the lock up definition $r' \geqq 2p$, and p cannot be refined. Consider the successor match n of m. If $s.occ(m) \leqq s.match(m)$, then m locks with n and therefore we must have $r = n$. By the definition of the match heuristic we then conclude that r' has period p, which is destroyed at the leftmost character of r. Thus $p(r) > p = p(r')$. Furthermore, $r > r' \geqq 2p(r')$, and so r is a major match.

Otherwise we must have $s.occ(m) > s.match(m)$, and therefore a new character Z was brought by $s(m)$ under the mismatch of m. By Lemma 4.4 we know that Z does not appear in p (or m). We claim that r mismatches under m'. If $r = n$ this follows. (If n mismatches where m does, then we had $s.occ(m) = s.match(m)$.) If $r \neq n$, the Z was shifted further right by the time we reached r, and r must mismatch upon or before reaching Z. Thus it again follows that the mismatch of r destroys the period p of r', and so $p(r) > p(r')$. As above $r > 2p = 2p(r')$ and r is major. This proves (2).

Assume now that $r \neq n$. Note that since r is major,

$$p(r) + p \geqq r \geqq m + p,$$

or

$$s(r) \geqq p(r) \geqq m.$$

Thus the shift of r will bring the Z discussed above to the right of m. But Z does not appear in p (and therefore r'). The match heuristic must therefore recommend a shift long enough to move the Z completely past the right end of r. The relation $s(r) \geqq r$ follows. □

7. The final accounting. We are now ready to prove that each shift made by the algorithm can pay for the matches that have been charged to it. Lemma 7.1 below is an immediate consequence of Lemmas 4.3, 6.1, and 6.2. It states that for a match m, twice the period of m' can account for all matches charged to m via cover up.

LEMMA 7.1. (Long Period). *Let $K(m)$ denote the set of matches charged to m via cover up. Then we have*

$$\sum_{t \in K(m)} c(t) \leqq 2p(m'). \quad \square$$

Proof. Let t be the first match assigned to m via cover up. From Lemma 4.3 we know that $p(m')$ comes at least as far left as the rightmost character of t. If $p(m')$ covers the right half of t, then, since then right halves of all matches assigned to m are disjoint by Lemma 6.2, the desired conclusion follows.

If, however, $p(m')$ does not cover the right half of t, then by Lemma 6.1, t is a major match and $p(m')$ covers $p(t)$. Thus $c(t) = 2p(t)$ and the conclusion follows as above. □

Using this result and Lemmas 6.3 and 6.4 dealing with lock up, we obtain the following bounds.

THEOREM 7.1 (Detailed Accounting). *Let N denote the length of the text. Then for every match m we have*

$$\sum_{t \to m} c(t) \leqq 2s(m) - d(m),$$

while for those matches on welfare we have

$$\sum_{t \to \text{welfare}} c(t) \leqq 2N. \quad \square$$

Proof. Let l be the first match assigned to m. If l was assigned via cover up, then by Lemma 6.3, all matches assigned to m were done so via cover up. If m is minor, then $d(m) = 0$, and the conclusion of the theorem follows from Lemma 7.1 and the fact that $s(m) \geqq p(m) \geqq p(m')$. If m is major, however, then $d(m) = m - 2p(m')$. Further $p(m) + p(m') \geqq m$ by Lemma 3.3, and so

$$2s(m) - d(m) \geqq 2p(m) - m + 2p(m') \geqq m \geqq 2p(m'),$$

and again the conclusion follows from Theorem 7.1.

Assume now that l was assigned to m via lock up. Let p be the period of the lock up definition. From Lemma 6.4 we know that $m \geqq l$, and m is major. Therefore $d(m) = m - 2p(m') = m - 2p$. If m is the successor of l, then

$$d(m) + l = m - 2p + l \leqq 2(m - p) \leqq 2p(m) \leqq 2s(m),$$

and the conclusion is proved.

Suppose next that m is not the successor of l. We know that all other matches besides l were assigned to m via cover up and, by Lemma 7.1, their charges does not exceed $2p(m')=2p$. As above

$$d(m)+l+2p=m+l\leqq 2m.$$

But since m is not the successor of l, from Lemma 6.4 we conclude that $s(m)\geqq m$, and we are done.

To complete the proof we must consider matches on welfare. Lemma 6.2 applies to these, and therefore their right halves are disjoint. From this the second conclusion of the theorem follows. □

We move $d(m)$ to the left hand side in the inequality of Theorem 7.1, then sum over all matches m, including those on welfare. For each match m, the left hand side will contain $c(m)$ and $d(m)$ exactly once. On the right hand side we will have twice the sum of all shifts made by the algorithm, plus $2N$. Thus:

THEOREM 7.2 (Boyer-Moore Linearity). *The total number of character comparisons made during an unsuccessful search by the Boyer-Moore algorithm over a text of length N is bounded by* $4N$. *Formally,*

$$\sum_{m \text{ a match}} m\leqq 4N. \quad \square$$

The analysis of the case where there are S occurrences of the pattern in the text is straightforward [7]. We end by noting that the above argument remains valid for a number of suggested improvements to the algorithm [2]. In one improvement *s.occ* is modified to stand for the shift necessary to bring over the mismatching text character a matching pattern character *which is to the left of the current mismatch* (as already discussed in Section 4). In another the two heuristics are merged into one; that is we shift so that all previously matched characters match again, *and* the new character over the mismatch position does match. Both of these approaches require precomputed tables of size equal to the product of the length of the pattern and the alphabet size.

REFERENCES

[1] ALFRED V. AHO, JOHN E. HOPCROFT AND JEFFREY D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

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

[3] LEO J. GUIBAS, *The analysis of hashing algorithms*, Xerox PARC Technical Report, CSL-76-3, July 1976.

[4] LEO J. GUIBAS AND ANDREW M. ODLYZKO, *The occurrence of periods in strings*, J. Combinatorial Theor., Series A, to appear.

[5] DONALD E. KNUTH, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, Mass., 1968; 2nd edition 1973.

[6] ———, *Sorting and Searching, The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, Mass., 1972.

[7] DONALD E. KNUTH, JAMES H. MORRIS, JR. AND VAUGHN B. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

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

[9] ROBERT SEDGEWICK, *Quicksort*, Stanford University, Computer Science Ph.D. Thesis, May 1975; also report STAN-CS-75-492.

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

[11] ZVI GALIL, *On improving the worst case running time of the Boyer-Moore string matching algorithm*, unpublished manuscript, Computer Science Division, Tel-Aviv University, 1978.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0003 $01.00/0

COMPATIBLE ORDERINGS ON THE METRIC THEORY OF TREES*

STEPHEN L. BLOOM† AND RALPH TINDELL‡

Abstract. In many studies of computation which make use of rooted labeled trees a partial ordering is usually imposed on the trees in the following way. A particular label, say $\perp_0$, is distinguished and identified with the atomic tree whose only vertex is a leaf labeled $\perp_0$. A tree f is then defined to be less than a tree g if g can be obtained from f by attaching some new trees to leaves of f labeled $\perp_0$.

This paper answers the following questions. What is the significance of the tree $\perp_0$ in this ordering? Can other nonatomic and perhaps infinite trees $\perp$ be used to define a partial ordering on the trees in the same way? If so, what if anything distinguishes the partial ordering defined via the atomic tree $\perp_0$?

Key words. partial orderings on trees, ordered and metric algebraic theories

1. Introduction. Rooted trees, both finite and infinite, have been used frequently in studies of the theory of computation (for example, see [1], [2], [6], [7], [10], [12], [13], [14]). In the trees described below each vertex has a finite outdegree, and each vertex of outdegree $k > 0$ is labeled by an element of a set Γ_k; (The leaves of the tree are labeled either by positive integers or by elements of Γ_0.) The elements of Γ_k are sometimes thought of as names of either operations with k possible outcomes or functions of k arguments. We will call the collection of these trees $\Gamma\,\mathrm{Tr}$. A partial ordering is usually imposed on $\Gamma\,\mathrm{Tr}$ in the following way. A particular element of Γ_0, say "$\perp_0$," is distinguished and identified with the atomic tree consisting of only a root labeled $\perp_0$. Then a tree f is defined to be less than the tree g, in symbols $f \sqsubseteq_{\perp_0} g$, if g can be obtained from f by attaching some new trees to leaves of f labeled $\perp_0$. (Usually, $\perp_0$ is read "undefined" and $f \sqsubseteq_{\perp_0} g$ is read: "f is less defined than g.") Thus $\perp_0$ is the least tree in this ordering.

The following questions, which arose while one of the authors was working on [5], led to the present paper. What is the significance of the tree $\perp_0$ in this partial ordering? Can other nonatomic and perhaps infinite trees $\perp$ be used to define a partial ordering $\sqsubseteq_\perp$ on the trees in the same way? If so, what if anything distinguishes the partial ordering defined via the atomic tree $\perp_0$?

These questions and others are answered below. It is shown that the trees $\perp$ which can be used to define a partial ordering having $\perp$ as least element are precisely the "homogeneous" trees. (A tree f is homogeneous if for each vertex v of f, the tree of descendants of v is isomorphic to f. Thus a finite homogeneous tree has only one vertex, and in any infinite homogeneous tree, all vertices have the same label.) This fact has some significance in view of the main result of [5], discussed in § 5.

Furthermore, we will compare one aspect of the partial-order structure on the trees with the complete metric space structure on $\Gamma\,\mathrm{Tr}$ ([3], [5], [13], [16]). It is shown that for each partial ordering $\sqsubseteq_\perp$ obtained from a homogeneous tree $\perp$, any increasing ω-chain is also a Cauchy sequence. The limit of the Cauchy sequence is the least upper bound of the ω-chain, so that all of the orderings are "ω-complete."

* Received by the editors March 2, 1979, and in revised form September 17, 1979.

† Department of Pure and Applied Mathematics, Stevens Institute of Technology, Hoboken, New Jersey 07030. The research of this author was partially supported by the National Science Foundation under grant MCS 78-00882.

‡ Department of Pure and Applied Mathematics, Stevens Institute of Technology, Hoboken, New Jersey 07030.

An operation of *composition* is imposed on the trees in Γ Tr turning this structure into an "algebraic theory" ([11], [8]). It is shown that composition preserves least upper bounds of ω-chains in each of the partial orderings $\sqsubseteq_\perp$, so that the theory (Γ Tr, $\sqsubseteq_\perp$) is, in the terminology of [1], an ω-continuous theory.

(Although some facts about algebraic theories are proved below, the reader need not be familiar with this notion. An attempt will be made to keep the paper self-contained.)

Lastly, if $\perp$ and $\perp'$ are homogeneous trees whose vertices have different out-degrees, then the structures (Γ Tr, $\sqsubseteq_\perp$) and (Γ Tr, $\sqsubseteq_{\perp'}$) are not isomorphic. Thus it follows from [1] that the property which distinguishes the ordering $\sqsubseteq_{\perp_0}$ is that only with this ordering is Γ Tr "freely generated" in the class of ω-continuous theories.

Here is a short summary of the remaining sections. In § 2, we identify the Γ-trees as an algebraic theory, and show how these trees form a complete metric space. In § 3, we discuss partial orderings on the trees which are compatible with the algebraic theory operations. We prove that if $\perp$ is least in some partial ordering compatible with composition (as above) then $\perp$ is homogeneous, and conversely, for any homogeneous tree $\perp$ one may define a partial ordering (by substitution) which is compatible with the theory operations and in which $\perp$ is least. In § 4, we prove that the orderings $\sqsubseteq_\perp$ are ω-complete and ω-continuous, and identify least upper bounds as metric limits. In the last section, we briefly review these facts in light of the main result of [5].

2. Γ Tr is a metric algebraic theory. We will first define precisely the notion of a "Γ-tree $1 \to p$." Here, Γ is the disjoint union of the sets Γ_k, $k \geqq 0$, and Γ is assumed disjoint from $[\omega]$, the set of positive integers. For a nonnegative integer p, $[p]$ denotes the set $\{1, 2, \cdots, p\}$; $[0]$ is the empty set. Lastly, $[\omega]^*$ is the set of all words on $[\omega]$.

DEFINITION 2.1 (See [1], [2], [10]). A *Γ-tree* $f: 1 \to p$ is a partial function $f: [\omega]^* \to \Gamma \cup [p]$ such that:

(2.2) (i) the domain of f is a nonempty, prefix-closed subset of $[\omega]^*$ (so f is defined at least on the empty word λ);

(ii) for $u \in [\omega]^*$, $i \in [\omega]$, if $uf \in \Gamma_k$ and $k > 0$, then uif is defined iff $i \in [k]$;

(iii) if $uf \in \Gamma_0 \cup [p]$ then uif is undefined for all $i \in [\omega]$.

The function f of Definition 2.1 "represents" in an obvious way the Γ-tree whose vertices are those words in $[\omega]^*$ in domain of f; if u is in the domain of f, i.e., u is a vertex of the tree, then the *label* of u is the element $uf \in \Gamma \cup [p]$. The "leaves" of f, i.e., those vertices with no successors, are those words whose labels are in $\Gamma_0 \cup [p]$.

From now on, we will let $\mathbf{v}(f)$ denote the domain of the Γ-tree f.

We let $\Gamma\,\mathrm{Tr}_{1,p}$ denote the collection of all Γ-trees $1 \to p$, and for $n \geqq 0$, let $\Gamma\,\mathrm{Tr}_{n,p}$ be the set of all n-tuples $(f_1, \ldots, f_n)$ with f_i in $\mathrm{Tr}_{1,p}$. In particular when $n = 0$, there is a unique element 0_p in $\Gamma\,\mathrm{Tr}_{o,p}$. We will sometimes say "f: $n \to p$ is a Γ-tree" instead of $f \in \Gamma\,\mathrm{Tr}_{n,p}$.

The integers in $[p]$ are used to define the operation of *composition*.

DEFINITION 2.3. Suppose $f: 1 \to p$ and $g = (g_1, \cdots, g_p): p \to q$ are Γ-trees. The *composition* $f \cdot g: 1 \to q$ is the tree defined as follows. For any $u, v, w \in [\omega]^*$,

(i) if $u \in \mathbf{v}(f)$ and $uf \in \Gamma$, then $u(f \cdot g) = uf$;

(ii) if $uf = i \in [p]$, then $u(f \cdot g) = \lambda g_i$;

(iii) if $u = wv$, where $wf = i$, then $u(f \cdot g) = vg_i$;

(iv) otherwise, $u(f \cdot g)$ is undefined.

Note that here case (ii) is a special case of case (iii). Thus the domain of $f \cdot g$, $\mathbf{v}(f \cdot g)$, is the union of $\mathbf{v}(f)$ with

$$\bigcup_{i=1}^{p} \{wv: wf = i \text{ and } v \in \mathbf{v}(g_i)\}.$$

DEFINITION 2.4. If $f=(f_1, \cdots, f_n): n \to p$, and $g: p \to q$ are Γ-trees, then $f \cdot g$ is defined to be the tree $(f_1 \cdot g, \cdots, f_n \cdot g)$. Each Γ-tree $f_i \cdot g$ was defined in 2.3.

We single out certain trees for special mention.

For each $\gamma \in \Gamma_k$, $k \geqq 0$, there is an *atomic* Γ-tree $\boldsymbol{\gamma}: 1 \to k$ defined only on the words $\{\lambda, 1, 2, \cdots, k\}$ as follows:

$$\lambda \boldsymbol{\gamma} = \gamma,$$
$$i\boldsymbol{\gamma} = i, \qquad i \in [k].$$

When $k=0$, $\boldsymbol{\gamma}: 1 \to 0$ is defined only on the empty word.

For each $n>0$ and each $i \in [n]$, the "distinguished" Γ-tree

$$\mathbf{i}: 1 \to n \tag{2.5}$$

is defined only on the empty word, and satisfies $\lambda\, \mathbf{i} = i$. (If we were fussy, we would add a subscript to i to indicate the target n.)

Let $I_n: n \to n$ be the Γ-tree $(\mathbf{1}, \mathbf{2}, \cdots, \mathbf{n})$.

The reader may verify that the following equations hold in $\Gamma\,\mathrm{Tr}$.

$$(2.6)\quad \begin{array}{l} \text{(i) } f \cdot (g \cdot h) = (f \cdot g) \cdot h, \quad \text{for any } f: n \to p,\ g: p \to q, \quad h: q \to r; \\ \text{(ii) } f \cdot I_p = I_n \cdot f = f, \quad \text{for any } f: n \to p; \\ \text{(iii) } \mathbf{i} \cdot (f_1, \cdots, f_n) = f_i, \quad \text{for any } f_1, \cdots, f_n: 1 \to p, \quad \text{any } i \in [n]; \\ \text{(iv) } (\mathbf{1} \cdot f, \mathbf{2} \cdot f, \cdots, \mathbf{n} \cdot f) = f, \quad \text{for any } f: n \to p. \end{array}$$

By virtue of satisfying (2.6), $\Gamma\,\mathrm{Tr}$ is an "algebraic theory." (Because only one minor result will be proved here about certain (ordered) algebraic theories, we relegate the definition of this concept to the appendix.)

Below we will make use of the fact that each set $\Gamma\,\mathrm{Tr}_{n,p}$ is a complete metric space (observed in [3], [5] and independently in [13] and [16]). Before defining the metric, we make a useful definition.

DEFINITION 2.7. Let f, g be Γ-trees $1 \to p$. A word $u \in [\omega]^*$ is *f-g critical* if $uf \neq ug$, but for every proper prefix w of u, $wf = wg$. (Here, $uf \neq ug$ if either both uf and ug are defined and unequal, or u is in the domain of only one of the functions.)

Of course there are no f-g critical words when $f=g$. When $f \neq g$, we let $\rho(f, g)$ be the *length* $|u|$ of a shortest f-g critical word u.

DEFINITION 2.8. Suppose f and g are Γ-trees $1 \to p$.

$$d(f, g) = \begin{cases} 0 & \text{if } f = g; \\ 2^{-\rho(f,g)} & \text{otherwise.} \end{cases}$$

If $f=(f_1, \cdots, f_n)$, $g=(g_1, \cdots, g_n)$ are Γ-trees $n \to p$, we define

$$d(f, g) = \max\{d(f_i, g_i): i \in [n]\}.$$

PROPOSITION 2.9 [5]. *For each $n, p \geqq 0$, the function d of Definition 2.8 is a complete metric on the sets $\Gamma\mathrm{Tr}_{n,p}$. Furthermore,*

$$d(f \cdot g_1, f \cdot g_2) \leqq d(g_1, g_2),$$

and

$$d(f_1 \cdot g, f_2 \cdot g) \leqq d(f_1, f_2),$$

whenever the compositions $f \cdot g_i$ and $f_i \cdot g$, $i=1, 2$, are defined.

Other examples of algebraic theories T connected with the theory of computation whose "hom sets" $T_{n,p}$ are complete metric spaces are discussed in [3] and [4].

Note that the sequence of trees

$$f_1, f_2, \cdots,$$

in $\Gamma\,\mathrm{Tr}_{1,p}$ converges to a tree f iff for any integer m all except a finite number of the trees f_i agree with f on all words in $[\omega]^*$ of length $\leqq m$.

In fact a slightly stronger assertion is true. If we denote the "tree of descendants of the vertex u in f" by $D_u f$, then $D_u f$ is the tree defined by the equation

$$wD_u f = (uw)f, \quad \text{for all } w \in [\omega]^*.$$

In this notation, if the sequence of trees $f_1, f_2 \cdots$, converges to the tree f, then for any vertex u in $\mathbf{v}(f)$, not only is u in $\mathbf{v}(f_n)$, for all sufficiently large n, but in fact

$$D_u f = \lim_{n\to\infty} D_u f_n. \tag{2.10}$$

This fact will be used in § 4.

3. Compatible orderings on Γ Tr. We begin with a definition.

DEFINITION 3.1. A *compatible* partial ordering on $\Gamma\,\mathrm{Tr}$ is a family of partial orderings $\sqsubseteq$ on the sets $\Gamma\,\mathrm{Tr}_{n,p}$ such that

(i) if $f_1 \sqsubseteq f_2$ in $\Gamma\,\mathrm{Tr}_{n,p}$ and $g_1 \sqsubseteq g_2$ in $\Gamma\,\mathrm{Tr}_{p,q}$ then $f_1 \cdot g_1 \sqsubseteq f_2 \cdot g_2$ in $\Gamma\,\mathrm{Tr}_{n,q}$;

(ii) if $f_i \sqsubseteq g_i$ in $\Gamma\,\mathrm{Tr}_{1,p}$, for each $i \in [n]$, then $(f_1, \cdots, f_n) \sqsubseteq (g_1, \cdots, g_n)$ in $\Gamma\,\mathrm{Tr}_{n,p}$.

The definition of a compatible partial ordering on an arbitrary algebraic theory T is obtained by replacing $\Gamma\,\mathrm{Tr}$ by T everywhere in Definition 3.1.

We will be interested in compatible partial orderings $\sqsubseteq$ on $\Gamma\,\mathrm{Tr}$ such that each set $\Gamma\,\mathrm{Tr}_{n,p}$ has a $\sqsubseteq$-least element. Because it is no more work to do so, we will prove a fact about any such compatible partial ordering on an *arbitrary* algebraic theory.

Recall that a morphism $f\colon A \to B$ in any category is an *epimorphism* if $g = h$ whenever $A \xrightarrow{f} B \xrightarrow{g} C = A \xrightarrow{f} B \xrightarrow{h} C$. One may easily show that a Γ-tree $f\colon 1 \to n$ is an epimorphism in $\Gamma\,\mathrm{Tr}$ iff for each $i \in [n]$, f has a leaf labeled i (i.e., $uf = i$ for some $u \in \mathbf{v}(f)$).

DEFINITION 3.2. A morphism $h\colon 1 \to p$ in an algebraic theory T is *homogeneous* if whenever h can be written as the composition $f \cdot g$ of an epimorphism $f\colon 1 \to n$ and a morphism $g\colon n \to p$ in T, then g must be $(h, h, \cdots, h)\colon n \to p$.

One may show that in $\Gamma\,\mathrm{Tr}$ a homogeneous morphism $1 \to p$ is a tree which has only one vertex, or else has an infinite number of vertices all of which have the same label in Γ.

The following theorem explains why only homogeneous trees may be least elements in a compatible ordering. Indeed, we prove a more general fact.

THEOREM 3.3. *Suppose that $\sqsubseteq$ is a compatible ordering on an algebraic theory T and $h\colon 1 \to p$ is the $\sqsubseteq$-least element in $T_{1,p}$. Then h is homogeneous.*

Proof. Suppose that $h = f \cdot g$, where $f\colon 1 \to n$ is an epimorphism. Since $h \sqsubseteq \mathbf{i} \cdot g$, for each $i \in [n]$, $(h, h, \cdots, h) \sqsubseteq g$, by Definition 3.1 (ii). Then by 3.1 (i), $f \cdot (h, \cdots, h) \sqsubseteq f \cdot g$. But since h is least in $T_{1,p}$,

$$h \sqsubseteq f \cdot (h, \cdots, h) \sqsubseteq f \cdot g = h.$$

Thus $f \cdot (h, h, \cdots, h) = f \cdot g$, and since f is an epimorphism, $g = (h, \cdots, h)$, proving h is homogeneous.

In the remainder of this section, we will define for each homogeneous tree $\perp\colon 1 \to 0$ in $\Gamma\,\mathrm{Tr}$ a compatible partial ordering $\sqsubseteq_\perp$ on $\Gamma\,\mathrm{Tr}$ such that for each $p \geqq 0$, $\perp \cdot 0_p$ is least in $\Gamma\,\mathrm{Tr}_{1,p}$. This ordering may be roughly described by saying that $f \sqsubseteq_\perp g$ if g may be obtained from f by replacing some "descendancy trees" of f equal to $\perp$ by other trees.

DEFINITION 3.4. Let $\perp : 1 \to 0$ be a homogeneous Γ-tree. For $f, g : 1 \to p$ in $\Gamma\,\mathrm{Tr}$, we define $f \sqsubseteq_\perp g$ if

(i) $D_u f = \perp \cdot 0_p$, whenever u is an f-g critical vertex.

If $f, g : n \to p$, we define $f \sqsubseteq_\perp g$ if

(ii) $\mathbf{i} \cdot f \sqsubseteq_\perp \mathbf{i} \cdot g$, for each $i \in [n]$.

Remark. Condition 3.4(i) is equivalent to the following Condition 3.4(i′) which we will use in the arguments below.

3.4(i′). $D_u f = \perp \cdot 0_p$ whenever $u \in \mathbf{v}(f)$ and $uf \neq ug$. Indeed it is clear that 3.4(i′) implies 3.4(i). Conversely, suppose 3.4(i) holds, $u \in \mathbf{v}(f)$ and $uf \neq ug$. Let w be the prefix of u of least length such that $wf \neq wg$. Then w is f-g critical, so that $D_w f = \perp \cdot 0_p$. But then $D_u f = \perp \cdot 0_p$ also, since $\perp$ is homogeneous.

We will show that the relation $\sqsubseteq_\perp$ defined in 3.4 is a compatible partial ordering in several steps. First we need the following facts; we omit the very easy proofs.

LEMMA 3.5. *Let $f, g ; 1 \to p$ in $\Gamma\,\mathrm{Tr}$. If $f \sqsubseteq_\perp g$ and $u \in \mathbf{v}(f)$, then*

$$D_u f \sqsubseteq_\perp D_u g.$$

LEMMA 3.6. *Assume $f, g : 1 \to p$ in $\Gamma\,\mathrm{Tr}$. If $u \in [\omega]^*$ is an f-g critical vertex, then both uf and ug are defined.*

LEMMA 3.7. *Assume $f, g : 1 \to p$ in $\Gamma\,\mathrm{Tr}$ and $uf \neq ug$. Then u has a unique prefix which is f-g critical.*

We can now prove that $\sqsubseteq_\perp$ is a partial ordering.

THEOREM 3.8. *When $\perp : 1 \to 0$ is homogeneous, the relation $\sqsubseteq_\perp$ defined in Definition 3.4 is a (family of) partial orderings and, for each $p \geqq 0$, $\perp \cdot 0_p$ is the $\sqsubseteq_\perp$-least Γ-tree $1 \to p$.*

Proof. Since $\perp$ and $\perp \cdot 0_p$ are homogeneous, it is clear that $\perp \cdot 0_p \sqsubseteq_\perp g$ for any tree $g : 1 \to p$. Since $\sqsubseteq_\perp$ is clearly reflexive, we need only show $\sqsubseteq_\perp$ is antisymmetric and transitive.

Thus suppose that $f, g : 1 \to p$ in $\Gamma\,\mathrm{Tr}$ and that $f \sqsubseteq_\perp g$ and $g \sqsubseteq_\perp f$. If there is a vertex u with $uf \neq ug$, then by Lemma 3.7 we may assume u is f-g critical. Hence by Lemma 3.6, both uf and ug are defined. Then, by the definition of $\sqsubseteq_\perp$, $D_u f = D_u g = \perp \cdot 0_p$, a contradiction. Thus $f = g$. By 3.4(ii), if f and g are in $\Gamma\,\mathrm{Tr}_{n,p}$ and $f \sqsubseteq_\perp g \sqsubseteq_\perp f$, $\mathbf{i} \cdot f = \mathbf{i} \cdot g$, for each $i \in [n]$. Hence $f = g$.

Note that the above argument has shown that if $f \sqsubseteq_\perp \perp \cdot 0_p$, then $f = \perp \cdot 0_p$.

As for transitivity, suppose that f, g, h are Γ-trees $1 \to p$ and $f \sqsubseteq_\perp g \sqsubseteq_\perp h$. Let u be a vertex in $\mathbf{v}(f)$ such that $uf \neq uh$. We must show $D_u f = \perp \cdot 0_p$. Either $uf = ug$ or not. If not, $D_u f = \perp \cdot 0_p$ since $f \sqsubseteq_\perp g$. If so, then $D_{u} g = \perp \cdot 0_p$. Thus, by Lemma 3.5, $D_u f = \perp \cdot 0_p$ completing the proof of the transitivity of $\sqsubseteq_\perp$ on $\Gamma\,\mathrm{Tr}_{1,p}$. The argument extends to $\Gamma\,\mathrm{Tr}_{n,p}$ in the obvious way. The proof of the theorem is complete.

Our next task is to show $\sqsubseteq_\perp$ is compatible with the theory operations, i.e., that Conditions 3.1(i) and 3.1(ii) hold. Since we have incorporated 3.1(ii) into our definition of $\sqsubseteq_\perp$ (in 3.4(ii)), we need only verify that 3.1(i) holds.

THEOREM 3.9. *The relation $\sqsubseteq_\perp$ is preserved by tree composition, so that $\sqsubseteq_\perp$ is a compatible ordering on $\Gamma\,\mathrm{Tr}$.*

Proof. We must show that if $f_1 \sqsubseteq_\perp f_2$ in $\Gamma\,\mathrm{Tr}_{n,p}$ and $g_1 \sqsubseteq_\perp g_2$ in $\Gamma\,\mathrm{Tr}_{p,q}$, then $f_1 \cdot g_1 \sqsubseteq_\perp f_2 \cdot g_2$ in $\Gamma\,\mathrm{Tr}_{n,q}$. From Definition 2.5 and Condition 3.4(ii) we may assume $n = 1$. Now suppose that u is a vertex in $\mathbf{v}(f_1 \cdot g_1)$ such that $u(f_1 \cdot g_1) \neq u(f_2 \cdot g_2)$. We must show $D_u({}_1 \cdot g_1) = \perp \cdot 0_q$.

Either

(a) $uf_1 \in \Gamma$, or

(b) $u = wv$, with $wf_1 = i$ and $v(\mathbf{i} \cdot g_1) = u(f_1 \cdot g_1)$.

In case (a), we must have $uf_1 \neq uf_2$ (or else $u(f_1 \cdot g_1) = u(f_2 \cdot g_2)$). But then, since $f_1 \sqsubseteq_\perp f_2$, $D_u(f_1 \cdot g_1) = (D_u f_1) \cdot g_1 = (\perp \cdot 0_p) \cdot g_1 = \perp \cdot 0_q$.

In case (b) if $wf_2 \neq i$, $D_w f_1 = \perp \cdot 0_p$, an impossibility. Thus $wv(f_2 \cdot g_2) = v(\mathbf{i} \cdot g_2)$, and $D_v(\mathbf{i} \cdot g_1) = \perp \cdot 0_q$, since $g_1 \sqsubseteq_\perp g_2$. But then $D_{wv}(f_1 \cdot g_1) = D_v(i \cdot g_1) = \perp \cdot 0_q$, completing the proof.

4. Propertiés of the orderings $\sqsubseteq_\perp$. The first task of this section is to show that if $\perp : 1 \to 0$ is a homogeneous Γ-tree, and $f_1 \sqsubseteq_\perp f_2 \sqsubseteq_\perp \cdots$, is an ω-chain of Γ-trees in $\Gamma\,\mathrm{Tr}_{1,p}$, then the chain has a least upper bound. In fact, we will first show that the sequence $f_1, f_2, \cdots$, is *Cauchy* and the metric limit f of the sequence (which exists since $\Gamma\,\mathrm{Tr}_{1,p}$ is a complete metric space) is the least upper bound. We will prove the sequence is Cauchy by using a series of lemmas.

For the remainder of this section $\perp : 1 \to 0$ is a fixed homogeneous Γ-tree and

$$f_1 \sqsubseteq_\perp f_2 \sqsubseteq_\perp f_3 \sqsubseteq_\perp \cdots, \tag{4.1}$$

is a *strictly increasing* ω-chain of trees in $\Gamma\,\mathrm{Tr}_{1,p}$.

LEMMA 4.2. *Suppose $f \sqsubseteq_\perp g \sqsubseteq_\perp h$ in $\Gamma\,\mathrm{Tr}_{1,p}$. Then every g-h critical vertex u has a prefix which is either f-g critical or f-h critical.*

Proof. If $uf \neq ug$, then u has an f-g critical prefix by Lemma 3.7. If $uf = ug$, then $uf \neq uh$ (since $ug \neq uh$), so again by Lemma 3.7, u has an f-h critical prefix.

With respect to the ω-chain (4.1), define a vertex u in $[\omega]^*$ to be *i-critical* if u is f_i-f_k critical for some $k > i$, but no proper prefix of u is f_i-f_j critical for any $j > i$. Now let ρ_i be the number defined by

$$\rho_i = \min\{|u| : u \text{ is } i\text{-critical}\}. \tag{4.3}$$

For each i, some i-critical vertex always exists since (4.1) is strictly increasing.

LEMMA 4.4. *If $i < j$, every f_i-f_j critical vertex has a unique i-critical prefix.*

COROLLARY 4.5. *If $i < j$, every j-critical vertex has a unique i-critical prefix.*

Proof. By Lemmas 4.2 and 4.4.

COROLLARY 4.6. *For $n \geqq i$, any n-critical vertex of length ρ_i is also i-critical.*

COROLLARY 4.7. *If $i < j$, $\rho_i \leqq \rho_j$.*

LEMMA 4.8. *For each $i \geqq 1$, there is some $n > i$ such that $\rho_i \neq \rho_n$ (so that $\rho_i < \rho_n$).*

Proof. Let $u_1, \cdots, u_m$ be all of the i-critical vertices of length ρ_i. Assume that in fact u_j is f_i-f_{n_j} critical, for $j \in [m]$. Now if $n = \max\{n_1, \cdots, n_m\}$, we will show that no vertex u_j is n-critical. Indeed, note that

$$D_{u_j} f_i = \perp \cdot 0_p \neq D_{u_j} f_{n_j}, \quad \text{for } j \in [m], \tag{4.9}$$

since u_j is f_i-f_{n_j} critical. If $u_j \notin \mathbf{v}(f_n)$, u_j is not n-critical, so assume that $u_j \in \mathbf{v}(f_n)$. But now, since $f_{n_j} \sqsubseteq_\perp f_n$, for each $j \in [m]$, $D_{u_j} f_{n_j} \sqsubseteq_\perp D_{u_j} f_n$. Thus by (4.9), $D_{u_j} f_n \neq \perp \cdot 0_p$, showing that u_j is not n-critical. It now follows from Corollary 4.6 that any n-critical vertex has length greater than ρ_i, completing the proof.

COROLLARY 4.10. *The sequence* (4.1) *is a Cauchy sequence.*

Proof. Recall from Definition 2.7 that $\rho(f, g)$ is the length $|u|$ of a shortest f-g critical vertex u. Since $\rho(f_i, f_j) \geqq \rho_i$, for $j > i$, and since $d(f_i, f_j) = 2^{-\rho(f_i, f_j)}$, we have $d(f_i, f_j) \leqq 2^{-\rho_i}$, for all i and $j > i$. But by Lemmas 4.7 and 4.8, $\lim_i \rho_i = \infty$, proving the Corollary.

We now show that the metric limit f of the Cauchy sequence f_n is the least upper bound of (4.1).

THEOREM 4.11. *Let* $f = \lim_{n\to\infty} f_n$. *Then* f *is the* $\sqsubseteq_\perp$*-least upper bound of* (4.1).

Proof. First we show f is an upper bound. Suppose that $u \in \mathbf{v}(f_i)$ and $uf_i \neq uf$. For all but a finite number of values of n, $uf = uf_n$. If n is sufficiently larger than i, $f_i \sqsubseteq_\perp f_n$ and $uf_i \neq uf_n$ and hence $D_u f_i = \perp \cdot 0_p$. Thus f is an upper bound to the sequence (4.1).

Now suppose that $f_i \sqsubseteq_\perp g$ for all i, and let $u \in \mathbf{v}(f)$ be a vertex such that $uf \neq ug$. Again, if n is large enough, $uf = uf_n$, so that $D_u f_n = \perp \cdot 0_p$. But $\lim_{n\to\infty} D_u f_n = D_u f$, by (2.10), so that $D_u f = \perp \cdot 0_p$. Thus $f \sqsubseteq_\perp g$, showing that f is the least upper bound of (4.1).

The following fact follows immediately.

COROLLARY 4.12. *Suppose that* $g_1 \sqsubseteq_\perp g_2 \sqsubseteq_\perp g_3 \sqsubseteq_\perp \cdots$, *is an* ω*-chain in* $\Gamma\,\mathrm{Tr}_{n,p}$, *for* $n > 1$. *Then the metric limit* $g = \lim_{n\to\infty} g_n$ *exists and is the least upper bound of the sequence* (g_n).

Luckily, the ω-continuity of composition follows easily from Corollary 4.12 (and Proposition 2.9).

THEOREM 4.13. *Let* $g_1 \sqsubseteq_\perp g_2 \sqsubseteq_\perp \cdots$, *and* $h_1 \sqsubseteq_\perp h_2 \sqsubseteq_\perp \cdots$, *be* ω*-chains in* $\Gamma\,\mathrm{Tr}_{n,p}$ *and* $\Gamma\,\mathrm{Tr}_{p,q}$ *respectively. If* $g = \sup(g_n)$ *and* $h = \sup(h_n)$, *then*

$$g \cdot h = \sup(g_n \cdot h_n).$$

Proof. It follows from Theorem 3.9 that for each n, $g_n \cdot h_n \sqsubseteq_\perp g_{n+1} \cdot h_{n+1}$, so the sequence $(g_n \cdot h_n)$ is an ω-chain. In order to show that $g \cdot h$ is the least upper bound of this sequence, we need only show that $g \cdot h$ is its metric limit, by Corollary 4.12.

But $d(g_n \cdot h_n, g \cdot h) \leqq d(g_n \cdot h_n, g \cdot h_n) + d(g \cdot h_n, g \cdot h)$, by the triangle inequality. From Proposition 2.9 it follows that the right-hand side is less than $d(g_n, g) + d(h_n, h)$, which goes to zero by Corollary 4.12 again. This completes the proof.

Theorems 3.8, 3.9, 4.12, 4.13 show that for any homogeneous tree $\perp : 1 \to 0$ in $\Gamma\,\mathrm{Tr}$ there is an ω-complete, ω-continuous compatible partial ordering on $\Gamma\,\mathrm{Tr}$ with $\perp$ least in $\Gamma\,\mathrm{Tr}_{1,0}$. One of the referees suggested that we answer the following

Question 4.14. Given a homogeneous morphism $\perp : 1 \to 0$ in an algebraic theory T, is there some ω-complete, ω-continuous compatible partial ordering on T having $\perp$ as the least morphism in $T_{1,0}$?

We answer this question negatively with the following example.

Example 4.15. Let S be the set consisting of the nonnegative integers N and two additional points $a \neq b$. Let f and g be the functions, $S \to S$ defined by:

$$nf = ng = n + 1, \qquad n \in N;$$
$$af = bf = a;$$
$$ag = bg = b.$$

Let T be the least subtheory of $S^\bullet$ containing the morphisms $f, g : 1 \to 1$ and the constants $\mathbf{0}: 1 \to 0$, $a : 1 \to 0$, $b : 1 \to 0$ (a morphism $n \to p$ in $S^\bullet$ is a function $S^p \to S^n$; the distinguished morphism $\mathbf{i}: 1 \to n$ is the ith projection $S^n \to S$; a morphism $1 \to 0$ in $S^\bullet$ may be identified with an element of S). It is easily seen that $\mathbf{0}: 1 \to 0$ is homogeneous, since the only epimorphism in T is the identity morphism $I_1 : 1 \to 1$. If $\sqsubseteq$ is an ω-complete, ω-continuous compatible partial ordering having $\mathbf{0}: 1 \to 0$ least in $T_{1,0}$, then

$$\mathbf{0} \sqsubseteq f \cdot \mathbf{0} \sqsubseteq f^2 \cdot \mathbf{0} \sqsubseteq \cdots \sqsubseteq f^n \cdot \mathbf{0} \cdots \text{l.u.b.}\,\{f^k \cdot \mathbf{0}: k \geqq 0\}.$$

Since $\sqsubseteq$ is ω-continuous, if $\alpha = \text{l.u.b.}\,\{f^k \cdot \mathbf{0}: k \geqq 0\}$ then $\alpha = f \cdot \alpha$, so that $\alpha = a : 1 \to 0$. However, for each $k \geqq 0$, $f^k \cdot \mathbf{0} = g^k \cdot \mathbf{0}$, so that $\alpha = g \cdot \alpha$ also. But then $\alpha = b$, a contradiction.

The last task of this section is to indicate (without proof) why the orderings $\sqsubseteq_\perp$ obtained from "different" homogeneous Γ-trees $\perp : 1 \to 0$ are distinct.

DEFINITION 4.16. *An ordered theory isomorphism* $F: (\Gamma\,\mathrm{Tr}, \sqsubseteq_\perp) \to (\Gamma\,\mathrm{Tr}, \sqsubseteq_{\perp'})$, where $\perp$ and $\perp'$ are homogeneous trees $1 \to 0$, is a family of bijections $\Gamma\,\mathrm{Tr}_{n,p} \to \Gamma\,\mathrm{Tr}_{n,p}$ (for each $n, p \geqq 0$) such that $\mathbf{i}F = \mathbf{i}$, for each distinguished tree $\mathbf{i}: 1 \to n$; $(f \cdot g)F = fF \cdot gF$, for all composable f and g; and such that $f \sqsubseteq_\perp g$ iff $fF \sqsubseteq_{\perp'} gF$.

The *degree* of a homogeneous tree $\perp: 1 \to 0$ is k if $\lambda \perp \in \Gamma_k$.

Using this terminology, we state the following.

THEOREM 4.17. *For homogeneous trees* $\perp, \perp': 1 \to 0$ *in* $\Gamma\,\mathrm{Tr}$, *there is an ordered theory isomorphism* $F: (\Gamma\,\mathrm{Tr}, \sqsubseteq_\perp) \to (\Gamma\,\mathrm{Tr}, \sqsubseteq_{\perp'})$ *iff* $\perp$ *and* $\perp'$ *have the same degree.*

In [1] it was shown that if $\perp$ has degree 0 (i.e., $\perp$ is an atomic tree) $(\Gamma\,\mathrm{Tr}, \sqsubseteq_\perp)$ is freely generated by Γ in the class of ω-complete, ω-continuous ordered theories. Thus only the homogeneous trees of degree zero yield free theories, since free theories are unique up to (ordered theory) isomorphism.

5. Order or metric. The association of a finite or infinite tree with a flowchart or recursive program is well known. Suppose the tree f is associated with the program scheme F with two exits (of Fig. 5.1). Then in order-theoretic semantics ([1], [2], [15]), the tree $f^\dagger$ associated with the program scheme indicated in Fig. 5.2 is the $\sqsubseteq_\perp$-least fixed point in $\Gamma\,\mathrm{Tr}_{1,1}$ of the operation

$$\xi \mapsto f \cdot (I_1, \xi), \tag{5.3}$$

where $\perp: 1 \to 0$ is a homogeneous tree of degree zero. Thus, this least fixed point is the $\sqsubseteq_\perp$-least upper bound of the sequence

$$\perp \cdot 0_1, \quad f \cdot (I_1, \perp \cdot 0_1), \quad f \cdot (I_1, f \cdot (I_1, \perp \cdot 0_1)), \cdots. \tag{5.4}$$

In this theory the tree $\perp$ is thought to represent "undefined" and the elements of the sequence (5.4) are thought to represent better and better "approximations" of the meaning of the scheme in Fig. 5.2.

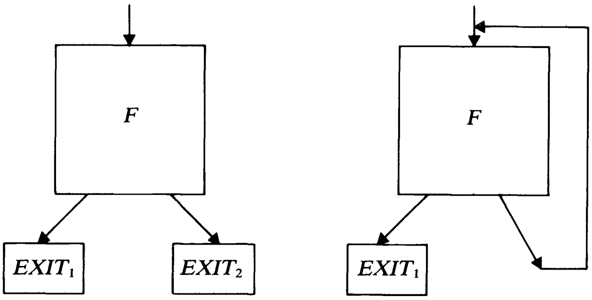


FIG. 5.1 FIG. 5.2

Conceivably, one might wish to represent "undefined" by a nonhomogeneous tree. Then, as shown in § 3, there is no compatible ordering on $\Gamma\,\mathrm{Tr}$ such that "undefined" is the least element, so that one is unable to define the "meaning" of the program scheme Fig. 5.2 as the *least upper bound* of the sequence (5.4).

However in [5] it was shown that for any tree $\perp: 1 \to 0$ (homogeneous or not), the sequence (5.4) is a Cauchy sequence and if its metric limit is denoted $f^\dagger$, then $f^\dagger$ is a fixed point of the operation (5.3), so that

$$f^\dagger = f \cdot (I_1, f^\dagger). \tag{5.5}$$

In fact in [5] it was shown that *all* equations (such as (5.5)) which hold when $f^\dagger$ is defined as the least upper bound of the sequence (5.4) when $\perp$ is homogeneous of degree zero, will also hold when $f^\dagger$ is defined as the metric limit of (5.4), for any choice of $\perp$.

This fact suggests that, at least for the study of ω-trees, metric limits are more versatile than least upper bounds.

Acknowledgment. We wish to thank Calvin C. Elgot for helpful comments on an earlier version of this paper.

Appendix: the definition of "algebraic theory." The notion "algebraic theory" was introduced by Lawvere [11]. The definition given below was first used by Elgot [8].

An *algebraic theory* T is a category whose objects are the nonnegative integers. For each $n \geqq 0$, there are n "distinguished morphisms" $\mathbf{i}: 1 \to n$, $i \in [n]$, with the following property:

(*) for any family $f_i: 1 \to p$, $i \in [n]$, of morphisms in T, there is a unique morphism $f: n \to p$ in T such that $f_i = 1 \xrightarrow{\mathbf{i}} n \xrightarrow{f} p$, for each $i \in [n]$.

In the case $n = 0$, the property (*) requires the existence of a unique morphism $0_p: 0 \to p$ in T.

The set of morphisms $n \to p$ in T is denoted $T_{n,p}$.

REFERENCES

[1] E. G. WAGNER, J. B. WRIGHT AND J. B. THATCHER, *Free continuous theories*, IBM Research Report RC-6906, December 1977.

[2] ———, *Rational algebraic theories and fixed point solutions*, Proc. 17th IEEE Symp. on Found. of Computing, Houston (1976), pp. 147–158.

[3] S. L. BLOOM, *Iterative and Metric Algebraic Theories*, publication of the Banach International Mathematical Center, Warsaw, to appear.

[4] ———, *All solutions of a system of recursion equations in trees and other contraction theories*, to appear.

[5] S. L. BLOOM, C. C. ELGOT AND J. B. WRIGHT, *Vector iteration in pointed iterative theories*, IBM Research Report RC-7322, Sept. 1978, this Journal, to appear.

[6] B. COURCELLE, *On recursive equations having a unique solution*, IRIA Research Report 285, March 1978.

[7] B. COURCELLE AND M. NIVAT, *Algebraic families of interpretations*, in 17th Symp. Found. Computer Science, Houston 1976.

[8] C. C. ELGOT, *Monadic computation and iterative algebraic theories*, Proc. Logic Collog., Bristol 1973, North Holland, Amsterdam 1975.

[9] ———, *Structured Programming with and without GO-TO Statements*, IEEE Trans. Software Eng. SE-2 No. 1 (March 1976); Erratum and Corrigendum (Sept. 1976).

[10] C. C. ELGOT, S. L. BLOOM AND R. TINDELL, *The algebraic structure of rooted trees*, J. Comput. System Sci., 16, No. 3 (1978), pp. 362–399.

[11] F. W. LAWVERE, *Functional semantics of algebraic theories*, Proc. Nat. Acad. Sci. U.S.A., 50 (1963), pp. 869–872.

[12] M. NIVAT, *On the interpretation of recursive polyadic program schemes*, Symposia Mathematica, 15 (1975), pp. 255–281.

[13] M. NIVAT AND A. ARNOLD, *Calculs infinis, interpretations metriques et plus grands points fixes*, in Colloque AFCET-SMF de Mathematiques Appliquees, Palaiseau (1978).

[14] B. ROSEN, *Program equivalence and context-free grammars*, J. Comput. Systems Sci., 11 (1975), pp. 358–374.

[15] D. SCOTT, *The lattice of flow diagrams*, in E. Engeler (editor), Symp. on Semantics of Algorithmic Languages, Springer-Verlag Lecture notes, No. 188 (1971).

[16] J. MYCIELSKI AND W. TAYLOR, *A compactification of the algebra of terms*, Algebra Universalis, 6 (1976), pp. 159–163.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

APPROXIMATE SOLUTIONS FOR THE BILINEAR FORM COMPUTATIONAL PROBLEM*

DARIO BINI†, GRAZIA LOTTI‡ AND FRANCESCO ROMANI§

Abstract. A set of bilinear forms can be evaluated with a multiplicative complexity lower than the rank of the associated tensor by allowing an arbitrarily small error. A topological interpretation of this fact is presented together with the error analysis. A complexity measure is introduced which takes into account the numerical stability of algorithms. Relations are established between the complexities of exact and approximate algorithms.

Key words. analysis of algorithms, approximate computations, computational complexity, numerical mathematics

1. Introduction. The nonscalar complexity of a problem is commonly defined as the minimal number of nonscalar multiplications required to solve it exactly by a straight-line algorithm. Here "exactly" means that no error is introduced by the algorithm. We call these Exactly Computing (EC) algorithms. In the computation with a d-digit floating point arithmetic, an error depending on d is introduced in the result.

In computing bilinear forms the complexity measure is related to the rank of the associated tensor. We show here that in some cases it is possible to decrease the number of the required multiplications by allowing an arbitrarily small error. We call these Arbitrary Precision Approximating (APA) algorithms. Thus we get algorithms of complexity lower than the rank of the associated tensor which solve the problem with the desired accuracy (superoptimal APA-algorithms). In § 2 an example of a superoptimal algorithm is given for a simple problem. In § 3 a theoretical interpretation is given related to several notions in topology. The concepts of border tensor, border tensorial basis, and border rank are introduced corresponding to the analogous well known concepts of tensor, tensorial basis, and tensorial rank. In § 4 an analysis of the error of APA-algorithms is made. In § 5 relations between exact and approximate algorithms are investigated.

2. Superoptimal algorithms. We introduce the concept of superoptimal APA-algorithm with an example.

Let

$$f_1(\underset{\sim}{x}, \underset{\sim}{y}) = x_1 y_1, \qquad f_2(\underset{\sim}{x}, \underset{\sim}{y}) = x_2 y_1 + x_1 y_2, \tag{2.1}$$

be bilinear forms where $\underset{\sim}{x}^T = (x_1, x_2)$, $\underset{\sim}{y}^T = (y_1, y_2)$.

Some results in arithmetic complexity theory (see for example [9]) allow us to show a lower bound of 3 multiplications when we compute (2.1). An optimal bilinear EC-algorithm is given trivially by

$$\begin{aligned} s_1 &\leftarrow x_2 y_1, \\ s_2 &\leftarrow x_1 y_2, \\ s_3 &\leftarrow s_1 + s_2, \\ s_4 &\leftarrow x_1 y_1, \end{aligned} \tag{2.2}$$

* Received by the editors January 24, 1979, and in revised form October 1, 1979.

† Istituto di Matematica, University of Pisa, Pisa, Italy.
‡ Istituto di Scienze dell'Informazione, University of Pisa, Pisa, Italy.
§ Istituto di Elaborazione dell'Informazione del C.N.R., via S. Maria 46, 56100, Pisa, Italy.

where the variables s_3, s_4 contain the results. Consider now the following bilinear forms:

$$\begin{aligned}\hat{f}_1(\underline{x}, \underline{y}) &= x_1y_1 + \varepsilon^2 x_2y_2 = f_1(\underline{x}, \underline{y}) + \varepsilon^2 x_2y_2,\\ \hat{f}_2(\underline{x}, \underline{y}) &= x_2y_1 + x_1y_2 = f_2(\underline{x}, \underline{y}),\end{aligned}\tag{2.3}$$

where $\varepsilon \neq 0$.

We have

$$\lim_{\varepsilon\to 0} \hat{f}_1(\underline{x}, \underline{y}) = f_1(\underline{x}, \underline{y}),$$

and

$$\hat{f}_2(\underline{x}, \underline{y}) = f_2(\underline{x}, \underline{y}),$$

so the bilinear forms (2.3) approximate the bilinear forms (2.1) with an arbitrarily small error.

There exists a bilinear APA-algorithm for the evaluation of (2.1) which needs 2 nonscalar multiplications, namely

$$\begin{aligned}&s_1' \leftarrow (x_1 + x_2\varepsilon)(y_1 + y_2\varepsilon),\\ &s_2' \leftarrow (x_1 - x_2\varepsilon)(y_1 - y_2\varepsilon),\\ &s_3' \leftarrow \tfrac{1}{2}s_1' + \tfrac{1}{2}s_2',\\ &s_4' \leftarrow \frac{1}{2\varepsilon}s_1' - \frac{1}{2\varepsilon}s_2',\end{aligned}\tag{2.4}$$

where the variables s_3', s_4' contain the results.

Algorithm (2.4) allows us to approximate the results of (2.2) with fewer multiplications than any EC-algorithm. We call such an algorithm a *superoptimal algorithm*.

3. Theoretical considerations. A set of bilinear forms over an infinite field $\mathbf{F}$, $f_s(\underline{x}, \underline{y})$, $s = 1, 2, \cdots, p$ ($\underline{x}$ n-vector, $\underline{y}$ m-vector), can be identified by a set of $n \times m$ matrices

$$\{A^{(s)} \equiv \{a_{ij}^{(s)}\}, s = 1, 2, \cdots, p, a_{ij}^{(s)} \in \mathbf{F}\},$$

such that

$$f_s(\underline{x}, \underline{y}) = \sum_{i=1}^{n} \sum_{j=1}^{m} x_i a_{ij}^{(s)} y_j, \qquad s = 1, 2, \cdots, p.\tag{3.1}$$

This set of matrices can be viewed as an $n \times m \times p$ third order tensor that is a three-dimensional array $\mathbb{A} = \{a_{ij}^{(s)}\}$. A *rank-t basis* of $\mathbb{A}$ over $\mathbf{F}$ is a set of triads $\{\underline{u}^{(r)} \otimes \underline{v}^{(r)} \otimes \underline{w}^{(r)}$, $r = 1, 2, \cdots, t\}$, such that

$$\mathbb{A} = \sum_{r=1}^{t} \underline{u}^{(r)} \otimes \underline{v}^{(r)} \otimes \underline{w}^{(r)},\tag{3.2}$$

where $\underline{u}^{(r)} \in \mathbf{F}^n$, $\underline{v}^{(r)} \in \mathbf{F}^m$, $\underline{w}^{(r)} \in \mathbf{F}^p$ and

$$\underline{u}^{(r)} \otimes \underline{v}^{(r)} \otimes \underline{w}^{(r)} \equiv \{u_i^{(r)} \cdot v_j^{(r)} \cdot w_s^{(r)}\}, \qquad r = 1, 2, \cdots, t.$$

The *rank* of $\mathbb{A}$ is the minimal integer t for which a rank-t basis of $\mathbb{A}$ exists. As is well known (see [5]) the number of nonscalar multiplications required to compute $f_s(\underline{x}, \underline{y})$ by a bilinear noncommutative EC-algorithm equals the rank of the associated tensor $\mathbb{A}$.

The optimal EC-algorithm is obtained from (3.2) as follows:

$$\underline{x}^T A^{(s)} \underline{y} = \sum_{r=1}^{t} (\underline{x}^T \underline{u}^{(r)})(\underline{v}^{(r)T} \underline{y}) w_s^{(r)}.$$

Example. The tensor associated with problem (2.1) is

$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0, & 1 & 0 \end{pmatrix}.$$

The rank-3 basis corresponding to algorithm (2.2) is

$$\{\underline{u}^{(r)}\} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \quad \{\underline{v}^{(r)}\} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \quad \{\underline{w}^{(r)}\} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Let $\mathcal{T}$ be the set of all $n \times m \times p$ tensors ($\mathcal{T} \simeq \mathbf{F}^{nmp}$), and let $\mathcal{T}_t$ be the class of rank-t tensors. We have

$$\mathcal{T} = \bigcup_{t=1}^{q} \mathcal{T}_t, \qquad q = \min\{nm, mp, np\},$$

$$\mathcal{T}_t \cap \mathcal{T}_{t'} = \emptyset, \qquad t \neq t'.$$

Let $\|\cdot\|$ be any norm on $\mathbf{F}^{nmp}$, and let $\partial\mathcal{T}_h$ be the boundary of $\mathcal{T}_h$ in the topology induced by the norm $\|\cdot\|$.

PROPOSITION 3.1. *If* $\mathbb{A} \in \mathcal{T}_h \cap \partial\mathcal{T}_h$, *then there exists a family of tensors* $\mathbb{E}(\varepsilon)$ *such that*

$$\lim_{\varepsilon \to 0} \|\mathbb{E}(\varepsilon)\| = 0,$$

$$\mathbb{A} + \mathbb{E}(\varepsilon) \in \mathcal{T}_k \qquad \forall \varepsilon > 0, \quad k \neq h.$$

Proof. For any neighborhood $\mathcal{U}_{\mathbb{A}}$ of $\mathbb{A}$, $\exists k \neq h$: $\mathcal{U}_{\mathbb{A}} \cap \mathcal{T}_k \neq \emptyset$. Since q is finite, some of $\mathcal{T}_k$ must intersect every neighborhood of $\mathbb{A}$, i.e., $\exists k \neq h$: $\forall \mathcal{U}_{\mathbb{A}}$, $\mathcal{U}_{\mathbb{A}} \cap \mathcal{T}_k \neq \emptyset$. Hence $\forall \varepsilon > 0, \exists \mathbb{A}_\varepsilon \in \mathcal{T}_k$: $\|\mathbb{A}_\varepsilon - \mathbb{A}\| < \varepsilon$. Taking $\mathbb{E}(\varepsilon) = \mathbb{A}_\varepsilon - \mathbb{A}$, we have proved the proposition.

Remark. In the preceding example,

$$\mathbb{A} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0, & 1 & 0 \end{pmatrix} \in \mathcal{T}_3,$$

$$\mathbb{A} + \mathbb{E}(\varepsilon) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0, & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \varepsilon^2, & 0 & 0 \end{pmatrix} \in \mathcal{T}_2.$$

DEFINITIONS. Let $\mathbb{A} \in \mathcal{T}_t \cap \partial\mathcal{T}_t$ and $\mathcal{S} = \{q : \mathbb{A} \in \partial\mathcal{T}_q\}$. A *rank-q border basis* of $\mathbb{A}$ is the family of bases of $\mathbb{A} + \mathbb{E}(\varepsilon) \in \mathcal{T}_q$. The *border rank* of $\mathbb{A}$ is $t_B = \min \mathcal{S}$, and the *optimal border basis* is the border basis of rank t_B.

It is obvious that $t_B \leqq t$. On the other hand, it can be easily proved that if the matrices $A^{(s)}$, $s = 1, 2, \cdots, p$, are linearly independent, then $t_B \geqq p$. Permuting the role of indices, under analogous independence hypotheses it follows $t_B \geqq \max(m, n, p)$.

DEFINITION. The rank-t tensor $\mathbb{A}$ is a *border tensor* if its border rank t_B is less than t.

A set of bilinear forms can be computed by superoptimal APA-algorithms when the associated tensor $\mathbb{A}$ is a border tensor. The resulting complexity equals the border rank of $\mathbb{A}$. Namely, let $\underline{u}^{(r)}(\varepsilon)\otimes\underline{v}^{(r)}(\varepsilon)\otimes\underline{w}^{(r)}(\varepsilon)$, $r=1,2,\cdots,t_B$, be a rank-t_B basis of $\mathbb{A}+\mathbb{E}(\varepsilon)$, i.e.,

$$\mathbb{A}+\mathbb{E}(\varepsilon)=\sum_{r=1}^{t_B}\underline{u}^{(r)}(\varepsilon)\otimes\underline{v}^{(r)}(\varepsilon)\otimes\underline{w}^{(r)}(\varepsilon). \tag{3.3}$$

Then

$$\begin{gathered} f_s(\underline{x},\underline{y})=\underline{x}^TA^{(s)}\underline{y}=\sum_{r=1}^{t_B}(\underline{x}^T\underline{u}^{(r)}(\varepsilon))(\underline{v}^{(r)T}(\varepsilon)\underline{y})w_s^{(r)}(\varepsilon)-\underline{x}^TE^{(s)}(\varepsilon)\underline{y}, \\ \mathbb{E}(\varepsilon)\equiv\{E^{(s)}(\varepsilon)\}\equiv\{e_{ij}^{(s)}(\varepsilon)\}. \end{gathered} \tag{3.4}$$

Example. The APA-algorithm (2.4) is given by

$$\{\underline{u}^{(r)}(\varepsilon)\}=\begin{pmatrix}1 & 1\\ \varepsilon & -\varepsilon\end{pmatrix},$$

$$\{\underline{v}^{(r)}(\varepsilon)\}=\begin{pmatrix}1 & 1\\ \varepsilon & -\varepsilon\end{pmatrix},$$

$$\{\underline{w}^{(r)}(\varepsilon)\}=\begin{pmatrix}\dfrac{1}{2} & \dfrac{1}{2}\\[2mm] \dfrac{1}{2\varepsilon} & -\dfrac{1}{2\varepsilon}\end{pmatrix}.$$

4. Error analysis. In the following we consider the $e_{ij}^{(s)}(\varepsilon)$ to be polynomials in ε of degree δ with null constant terms. Moreover we suppose that in (3.3) the vectors $\underline{u}^{(r)}(\varepsilon)$, $\underline{v}^{(r)}(\varepsilon)$, $\underline{w}^{(r)}(\varepsilon)$ have as elements rational functions of ε.

Suppose we want to compute (3.1) by the APA-algorithm (3.4) using d-digit floating point arithmetic. The order of the error is estimated as a function of d. The function $\varepsilon(d)$ which minimizes the order of the error is investigated too ($\varepsilon(d)\to 0$ as d tends to infinity).

In computing (3.4) there are two sources of errors, namely the error due to finite arithmetic (roundoff error) and the error introduced by the approximation.

Let $O(\varepsilon^{-\varphi})$ be the largest infinite in the triads of the basis, i.e.,

$$\begin{aligned} \varphi=\max\{z\,|\,u_i^{(r)}(\varepsilon)v_j^{(r)}(\varepsilon)w_s^{(r)}(\varepsilon)=O(\varepsilon^{-z}),\ i=1,2,\cdots,m,\\ j=1,2,\cdots,n,\ s=1,2,\cdots,p,\ r=1,2,\cdots,t_B\}. \end{aligned}$$

The error due to arithmetic is $O(2^{-d}\varepsilon^{-\varphi})$ [1].

On the other hand let $O(\varepsilon^{\sigma})$ be the slowest infinitesimal in $\mathbb{E}(\varepsilon)$, i.e.,

$$\sigma=\min\{z\,|\,e_{ij}^{(s)}=O(\varepsilon^{z}),\ i=1,2,\cdots,m,\ j=1,2,\cdots,n,\ s=1,2,\cdots,p\}.$$

PROPOSITION 4.1. [1] *The error produced by an* APA-*algorithm, is* $O(2^{-d/\omega})$, *where* $\omega=1+\varphi/\sigma$ *is a stability parameter depending on the border basis.*

Proof. The error due to the approximation in the sth bilinear form is

$$\underline{x}^TE^{(s)}\underline{y}=O(\varepsilon^{\sigma}).$$

Then the overall error is $O(\varepsilon^{\sigma}+2^{-d}\varepsilon^{-\varphi})$; by choosing $\varepsilon(d)=2^{-d/(\varphi+\sigma)}$ the order of the error is minimized. The error becomes $O(2^{-d\sigma/(\varphi+\sigma)})=O(2^{-d/\omega})$ where $\omega=1+\varphi/\sigma$ is the stability parameter of the APA-algorithm.

Remark. For EC-algorithms the error is $O(2^{-d})$ and the stability parameter is $\omega = 1$ by definition.

For the algorithm (2.4), $\varphi = 1$, $\sigma = 2$, and hence $\omega = \frac{3}{2}$. Then the error is $O(2^{-2d/3})$ with $\varepsilon = 2^{-d/3}$.

Consider now a noncommutative bilinear algorithm a with a nonscalar multiplicative complexity N. We introduce a complexity measure AC which takes into account the stability-complexity relations.

Let $d_a(s)$ be a function such that the relative error produced by the algorithm using a $d_a(s)$-digit arithmetic is bounded by 2^{-s}. It is easy to show that for EC- and APA-algorithms $d_a(s) = \omega s + o(s)$.

Let $m(x)$ be the complexity of x-digit binary multiplication in terms of bit operations. Then the ratio $m(d_a(s))/m(s)$ denotes the major cost of the algorithm a versus the ideal one (i.e., the algorithm producing no errors).

The asymptotical complexity of a is defined as

$$\mathrm{AC}_a = \lim_{s\to\infty} N\frac{m(d_a(s))}{m(s)}.$$

In [4], under some regularity hypotheses on $m(x)$, it is proved that $\mathrm{AC}_a = N\omega$, where ω is the stability parameter of a.

Remark. AC is a finite number and can be used to compare the "infinite precision" complexity of different algorithms. Namely,

$$\frac{\mathrm{AC}_a}{\mathrm{AC}_b} = \lim_{s\to\infty} \frac{N_a m(d_a(s))}{N_b m(d_b(s))}.$$

For algorithm (2.2) we have $N = 3$, $\omega = 1$, $\mathrm{AC} = 3$, and for algorithm (2.4) $N = 2$, $\omega = \frac{3}{2}$, $\mathrm{AC} = 3$. Thus this superoptimal APA-algorithm gives a gain of $\frac{2}{3}$ on complexity but a reduction of $\frac{2}{3}$ on the number of significant digits in the result, and the complexity measure AC remains constant.

5. Relations between approximate and exact algorithms. The existence of border bases for a given tensor $\mathbb{A}$ allows us to bound the tensorial rank of $\mathbb{A}$.

PROPOSITION 5.1. [2] *Given a rank-t_B border basis for $\mathbb{A}$ with a polynomial correction $\mathbb{E}(\varepsilon)$ of degree δ, a rank-$(1+\delta)t_B$ tensorial basis for $\mathbb{A}$ exists.*

Proof. Let $\underline{\alpha} = \{\alpha_j\}$ be the solution of the Vandermonde system

$$\sum_{j=0}^{d} \varepsilon_j^i \alpha_j = \begin{cases} 1, & i = 0, \\ 0, & i = 1, 2, \cdots, \delta, \end{cases}$$

for a given $\{\varepsilon_j\}$ with the property $\varepsilon_j \neq \varepsilon_k$, for $j \neq k$; and let $\underline{u}^{(r)}(\varepsilon) \otimes \underline{v}^{(r)}(\varepsilon) \otimes \underline{w}^{(r)}(\varepsilon)$ be a tensorial basis for $\mathbb{A} + \mathbb{E}(\varepsilon)$. Then

$$\sum_{j=0}^{\delta} \alpha_j \sum_{r=1}^{t_B} \underline{u}^{(r)}(\varepsilon_j) \otimes \underline{v}^{(r)}(\varepsilon_j) \otimes \underline{w}^{(r)}(\varepsilon_j) = \sum_{j=0}^{\delta} \alpha_j(\mathbb{A} + \mathbb{E}(\varepsilon_j)) = \mathbb{A}.$$

Hence

$$\alpha_j(\underline{u}^{(r)}(\varepsilon_j) \otimes \underline{v}^{(r)}(\varepsilon_j) \otimes \underline{w}^{(r)}(\varepsilon_j)), \qquad j = 0, 1, \cdots, \delta, \quad r = 1, 2, \cdots, t_B,$$

is a rank-$(1+\delta)t_B$ tensorial basis for $\mathbb{A}$.

COROLLARY. *An* APA-*algorithm with polynomial correction of degree* δ *and multiplicative complexity* t_B *produces an* EC-*algorithm of multiplicative complexity* $(1+\delta)t_B$; *namely,*

$$\underline{x}^T A^{(s)} \underline{y} = \sum_{j=0}^{\delta} \alpha_j \sum_{r=1}^{t_B} (\underline{x}^T \underline{u}^{(r)}(\varepsilon_j))(\underline{v}^{(r)T}(\varepsilon_j)\underline{y}) w_s^{(r)}(\varepsilon_j).$$

The algorithms of this type are called Exactly Computing Derived (ECD) algorithms.

Remark. By slightly modifying the proof of Proposition 5.1 it is possible to obtain a rank-$(1+\delta')t_B$ tensorial basis for $\mathbb{A}$, where $\delta' \leqq \delta$ is the number of linearly independent polynomials in $\mathbb{E}(\varepsilon)$.

In general, for APA-algorithms we have $\mathrm{AC} = \omega t_B$, and for ECD-algorithms $\mathrm{AC} = (1+\delta)t_B$. In the computation of $n \times n$ matrix multiplication using APA-algorithms and the technique of recursive partitioning we have $\omega = O(\log n)$, $\delta = O(\log n)$ [2], [7]. A detailed discussion of the various kinds of matrix multiplication algorithms and of their complexity measures can be found in [7].

6. Conclusion. The existence of a sequence of lower rank tensors converging to a tensor of higher rank is a topological property useful in computational complexity theory. It would be interesting to know something more about the topology of the classes $\mathcal{T}_h$.

The technique to derive exact algorithms from approximate ones, exposed in [2], gives fruitful results when applied to problems whose associated tensors are tensorial powers of given ones. This is the case of matrix multiplication. The use of border tensors allowed one to reduce the complexity of this problem to $O(n^{2.7799})$ [3]. This bound has been successively improved by Pan [6] and Schönhage [8] ($O(n^{2.52})$).

Acknowledgment. The authors wish to thank the referees for their helpful comments.

REFERENCES

[1] D. BINI, *Border tensorial rank of triangular Toeplitz matrices*, Report B-78-26, I.E.I. Pisa (December 1978).

[2] ———, *Relations between exact and approximate bilinear algorithms. Applications.* Calcolo, to appear.

[3] D. BINI, M. CAPOVANI, G. LOTTI AND F. ROMANI, *$O(n^{2.7799})$ complexity for $n \times n$ approximate matrix multiplication*, Information Processing Lett., 8 (1979), pp. 234–235.

[4] D. BINI, G. LOTTI AND F. ROMANI, *Stability and complexity in the evaluation of a set of bilinear forms*, Report B-78-25, I.E.I. Pisa (November 1978).

[5] 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.

[6] V. YA. PAN, *Field extension and trilinear aggregating, uniting and canceling for the acceleration of matrix multiplication*, Proc. 20th Ann. Symp. on Foundation of Comp. Science, 1979, pp. 28–30.

[7] F. ROMANI, *Complexity measures for matrix multiplication algorithms*, Calcolo, to appear.

[8] A. SCHÖNHAGE, *Partial and total matrix multiplication*, Internal Report, University of Tübingen (January 1980).

[9] J. VAN LEEUWEN AND P. VAN EMDE BOAS, *Some elementary proofs of lower bounds in complexity theory*, Linear Algebra and Appl., 19 (1978), pp. 63–80.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0005 $01.00/0

THE APPLICATION OF MULTIVARIATE POLYNOMIALS TO INFERENCE RULES AND PARTIAL TESTS FOR UNSATISFIABILITY*

DAVID A. PLAISTED†

Abstract. There are some relationships between unsatisfiability of sets of clauses, and properties of polynomials in several variables. These polynomials can be used to obtain a polynomial time solution to a certain problem involving sets of clauses. Using these polynomials, one can establish a correspondence between unsatisfiable sets of clauses and a convex region of Euclidean space. Also, some inference rules based on these polynomials may provide shorter proofs of inconsistency than are possible using other known inference rules.

Key words. unsatisfiability, propositional calculus, polynomials in several variables, proof length, coNP, inference rules, linear programming

Introduction. The question of whether there is an efficient algorithm to decide the satisfiability of a set of clauses in the propositional calculus has important relationships to many problems in computer science [1], [5]. The related question, whether there is a proof system containing short proofs of inconsistency of all inconsistent sets of clauses, is also of interest. A survey of the known relationships among several proof systems has been done [2]. The known results are mainly of two types, simulation results and lower bounds. The simulation results show that one proof system can simulate another with only a polynomial increase in proof length. The lower bounds show that certain proof systems require proofs of more than polynomial size on specified sets of examples. Among the proof systems considered so far are resolution [10], regular resolution [11], resolution with extension [11], and Davis and Putnam's method [3]. It is known that regular resolution and Davis and Putnam's method require more than polynomial size proofs on certain sets of examples. The best known lower bound for these systems is given by Galil [4]. The corresponding questions for resolution and resolution with extension are still open.

We present some partial tests for unsatisfiability, based on linear programming and some inference rules which use polynomials in several variables. These inference rules may provide shorter proofs of inconsistency than resolution or other known inference rules can provide. Of course, if all inconsistent sets of clauses have short proofs, then NP = CoNP. Another possibility is that short proofs exist relative to a slowly growing, but infinite, set of axioms. We explore these possibilities. It turns out that polynomials associated with inconsistent sets of clauses over n variables correspond to a region of Euclidean space which is convex and is the intersection of 2^n halfspaces. We present polynomial time algorithms for several problems involving these polynomials, and present a problem for which no polynomial time solution is known. This work contrasts with earlier work of the author [9], in which the satisfiability problem is related to sparse polynomials in *one* variable.

1. Polynomials in many variables.

DEFINITION 1. With a vector $\mathbf{x}$ in $\{0, 1\}^n$ we associate an interpretation $I(\mathbf{x})$ of the variables $x_1, x_2, \cdots, x_n$ in the usual way. That is, x_i is true in $I(\mathbf{x})$ if x_i (the ith component of $\mathbf{x}$) is 1; x_i is false in $I(\mathbf{x})$ if x_i is 0.

* Received by the editors May 31, 1979, and in final revised form February 4, 1980.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This research was supported in part by the National Science Foundation under Grant MCS 77-22830.

DEFINITION 2. Suppose S is a set of clauses over the variables $x_1, x_2, \cdots, x_n$ and f is a function assigning a rational weight to each clause in S. Then Poly(S, f) is defined to be the polynomial p over the variables $x_1, x_2, \cdots, x_n$ having the following properties:

1. For all $\mathbf{x} \in \{0, 1\}^n$, $p(\mathbf{x}) = f(C_1) + f(C_2) + \cdots + f(C_k)$, where $\{C_1, C_2, \cdots, C_k\}$ is the set of clauses of S that are false in $I(\mathbf{x})$. We assume that the C_i are all distinct. Thus $p(\mathbf{x})$ is the weighted sum of the clauses of S that are false in the interpretation $I(\mathbf{x})$.

2. The polynomial p is a sum of terms of the form $rx_{i_1}x_{i_2} \cdots x_{i_m}$, where $i_1, i_2, \cdots, i_m$ are all distinct and r is a rational number. Thus no variable occurs in p to a power higher than the first power.

It is not difficult to show using properties of polynomials of several variables that Poly(S, f) is uniquely defined, given S and f. Therefore $p(\mathbf{x}) = 0$ for all $\mathbf{x} \in \{0, 1\}^n$ iff all coefficients of p are zero. Also, if S is a set of 3-literal clauses, then Poly(S, f) can be computed from S and f in a number of arithmetic operations that is linear in the size of S. We represent the rational number a/b by the ordered pair (a, b) for integers a and b.

Examples.

$$\text{Poly}(\{x_1 \vee x_2 \vee x_3\}, 1) = (1 - x_1) * (1 - x_2) * (1 - x_3)$$
$$= 1 - x_1 - x_2 - x_3 + x_1x_2 + x_2x_3 + x_1x_3 - x_1x_2x_3,$$
$$\text{Poly}(\{\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3\}, 1) = x_1x_2x_3,$$
$$\text{Poly}(S, f) = \sum_{C \in S} f(C) * \text{Poly}(\{C\}, 1).$$

(We denote the constant function $f(\mathbf{x}) \equiv c$ by c; thus 1 denotes the constant function $f(\mathbf{x}) \equiv 1$.)

This construction gives an efficient algorithm for the following problem, first treated in [7].

*Problem P*1. Given sets $S1$ and $S2$ of 3-literal clauses over $x_1, \cdots, x_n$, to decide whether for all interpretations I of $x_1, \cdots, x_n$, the number of clauses of $S1$ that are false in I equals the number of clauses of $S2$ that are false in I.

We solve this problem by computing Poly$(S1, 1)$ and Poly$(S2, 1)$. The sets $S1$ and $S2$ satisfy the above condition iff Poly$(S1, 1) =$ Poly$(S2, 1)$. This test only requires a number of arithmetic operations and comparisons that is linear in the size of $S1$ and $S2$. We do not know whether problem $P1$ can be solved in polynomial time if the number of literals per clause is unbounded.

We can also get an efficient, trivial algorithm for the following problem, using these polynomials:

*Problem P*2. Given sets $S1$ and $S2$ of arbitrarily large negative clauses over $\{x_1, \cdots, x_n\}$, to decide whether for all interpretations I of $x_1, x_2, \cdots, x_n$, the number of clauses of $S1$ that are false in I equals the number of clauses of $S2$ that are false in I. (A clause is negative if all literals in the clause are negative. Thus $\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$ is a negative clause.) This problem was posed in [7], in a different form.

Note that if $S1$ and $S2$ consist entirely of negative clauses, then Poly$(S1, 1)$ and Poly$(S2, 1)$ can be obtained very easily, and the above condition is true iff Poly$(S1, 1) =$ Poly$(S2, 1)$. However, it turns out that in this case Poly$(S1, 1) =$ Poly$(S2, 1)$ iff $S1 = S2$. Thus the condition is true iff $S1 = S2$. This problem is closely related to an NP-complete problem mentioned in [8]. If the number of *positive literals* per clause is bounded, we can compute Poly$(S1, 1)$ and Poly$(S2, 1)$ in linear time and so obtain a fast algorithm for this generalized problem. In fact, we can still get a linear algorithm if the number of x_i such that x_i and $\bar{x}_i$ both appear in $S1 \cup S2$ is bounded. This is because changing the sign of a propositional variable does not affect the property we are testing for. In particular, we can still solve problem $P2$ efficiently if all the clauses are *positive* (that is, have only positive literals).

THEOREM 1. *The following problem, Problem $P3$, is* NP*-complete.*

Problem $P3$. Given a polynomial $p(x_1, x_2, \cdots, x_n)$ with rational coefficients, to determine whether there exists $\mathbf{x} \in \{0, 1\}^n$ such that $p(\mathbf{x}) = 0$.

Proof. This problem is clearly in NP. Also, a set S of 3-literal clauses over the variables $x_1, x_2, \cdots, x_n$ is consistent iff $\exists \bar{x} \in \{0, 1\}^n$ such that $\text{Poly}(S, 1)(\mathbf{x}) = 0$. Furthermore, $\text{Poly}(S, 1)$ can be computed from S in polynomial time.

This result is not very profound, but polynomials in several variables have a convenient mathematical structure which helps to give us insight into the nature of the satisfiability problem.

THEOREM 2. *Suppose $S1$ and $S2$ are sets of 3-literal clauses over $x_1, x_2, \cdots, x_n$ and f_1 and f_2 are weighting functions for $S1$ and $S2$, respectively. Suppose $f_1(C) > 0$ for all $C \in S1$ and $f_2(C) > 0$ for all $C \in S2$. Suppose* $\text{Poly}(S1, f_1) = \text{Poly}(S2, f_2)$. *Then $S1$ is inconsistent iff $S2$ is.*

This result suggests inference rules for unsatisfiability. Namely, if we know that $S1$ as in the theorem is inconsistent, so is $S2$. However, there does not appear to be any relationship between such sets $S1$ and $S2$ in terms of proofs of inconsistency. Hence we might hope to obtain short proofs of inconsistency using inference rules based on $\text{Poly}(S, f)$ for a set S of 3-literal clauses.

For example, the following sets $S1$, $S2$, and $S3$ of clauses satisfy $\text{Poly}(S1, 1) = \text{Poly}(S2, 1) = \text{Poly}(S3, 1) = 1$:

$$\begin{array}{rll}
S1: & x_1 \vee x_2 \vee x_3 & \bar{x}_1 \vee x_2 \vee x_3 \\
 & x_1 \vee x_2 \vee \bar{x}_3 & \bar{x}_1 \vee x_2 \vee \bar{x}_3 \\
 & x_1 \vee \bar{x}_2 \vee x_3 & \bar{x}_1 \vee \bar{x}_2 \vee x_3 \\
 & x_1 \vee \bar{x}_2 \vee \bar{x}_3 & \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, \\
S2: & x_1 \vee x_2 \vee x_3 & x_1 \vee \bar{x}_2 \\
 & \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 & x_2 \vee \bar{x}_3 \\
 & & x_3 \vee \bar{x}_1, \\
S3: & x_1 & \\
 & \bar{x}_1 \vee x_2 & \\
 & \bar{x}_1 \vee \bar{x}_2 \vee x_3 & \\
 & \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3. &
\end{array}$$

Thus if we know one of these sets to be inconsistent, we can easily show the others to be inconsistent since their polynomials are all identical.

2. Linearity properties of the coefficients. Notice that the coefficients of $\text{Poly}(S1, f)$ for fixed $S1$ are linear combinations of the values $f(C)$ for $C \in S1$. Thus we can get a polynomial time solution to the following problem.

Problem $P4$. Given sets $S1$ and $S2$ of 3-literal clauses over $x_1, \cdots, x_n$, and given a weighting function f_1 for $S1$, to find a weighting function f_2 for $S2$ such that $\text{Poly}(S1, f_1) = \text{Poly}(S2, f_2)$, if such f_2 exists.

We can solve this problem in polynomial time since each coefficient of Poly$(S2, f_2)$ is a linear combination of the values $f_2(C)$ for $C \in S2$. By Gaussian elimination, we can obtain rational values for the quantities $f_2(C)$ so that Poly$(S2, f_2) = $ Poly$(S1, f_1)$, if such values exist. This gives a polynomial time solution since the numerators and denominators do not grow in size too quickly.

The significance of this result is that if for all $\mathbf{x} \in \{0, 1\}^n$, Poly$(S_1, f_1)(\mathbf{x}) \neq 0$, and if such f_2 exists, then $S2$ is inconsistent. Further, if $S1$ is inconsistent and $f_1(C) > 0$ for all C in $S1$, then Poly$(S1, f_1)(\mathbf{x}) \neq 0$ for all $\bar{x} \in \{0, 1\}^n$.

Consider the following problem.

Problem P5. Given sets $S1$ and $S2$ of 3-literal clauses over $x_1, x_2, \cdots, x_n$, to find weighting functions f_1 and f_2 such that $f_1(C) > 0$ for all $C \in S1$ and such that Poly$(S1, f_1) = $ Poly$(S2, f_2)$, if such f_1 and f_2 exist. The significance of this problem is that if $S1$ is inconsistent and if f_1 and f_2 exist, then $S2$ is inconsistent also. We have the following easy result:

THEOREM 3. *Problem P5 can be solved in polynomial time by reducing it to linear programming and applying Khachian's algorithm* [6]. *We reduce P5 to the following problem: Given an integer matrix A and an integer l, to determine whether there exists a vector* $\mathbf{z}$ *such that* $A\mathbf{z} = 0$ *and such that* $z_i > 0$ *for* $i = 1, 2, \cdots, l$.

Consider the following problem.

Problem P6. Given sets $S1$ and $S2$ of 3-literal clauses and a weighting function f_1 for $S1$, to determine if there exists a weighting function f_2 for $S2$ such that every coefficient of Poly$(S2, f_2) - $ Poly$(S1, f_1)$ is nonnegative. Note that if such an f_2 exists, and if Poly$(S1, f_1)(\mathbf{x}) > 0$ for all $\mathbf{x} \in \{0, 1\}^n$, then Poly$(S2, f_2)(\mathbf{x}) > 0$ for all $\mathbf{x} \in \{0, 1\}^n$ also, and so $S2$ is inconsistent.

We can easily get the following result.

THEOREM 4. *Problem P6 can be solved in polynomial time by reducing it to the following problem: Given an integer matrix A and a rational vector* $\mathbf{b}$, *to determine whether there exists a rational vector* $\mathbf{z}$ *such that* $A\mathbf{z} \geqq \mathbf{b}$. *Here inequality is applied componentwise. As in Problem P5, we use Khachian's algorithm.*

3. Isomorphism.

DEFINITION 3. Suppose $S1$ and $S2$ are sets of 3-literal clauses over $x_1, x_2, \cdots, x_n$. We say $S1 \sim S2$, if $S2$ can be obtained from $S1$ by permuting variables and by changing signs of variables.

It is clear that if $S1 \sim S2$, then $S1$ is inconsistent iff $S2$ is. Also, it is not hard to show that determining whether $S1 \sim S2$ is polynomially equivalent to graph isomorphism. Similarly, given polynomials p_1 and p_2 over $x_1, \cdots, x_n$, determining whether p_1 can be obtained from p_2 by permuting variables, is polynomially equivalent to graph isomorphism. We do not know whether this is still true if we also allow replacements of the form $x \leftarrow 1 - x_j$.

DEFINITION 4. Suppose p_1 and p_2 are polynomials in the variables $x_1, x_2, \cdots, x_n$. We say that $p_1 \sim p_2$ if p_2 can be obtained from p_1 by permuting variables and by replacements of the form $x_j \leftarrow 1 - x_j$. Note that this is an equivalence relation.

4. Denseness of nonzero values. The following results give us more insight into the behavior of the functions Poly(S, f). In particular, the values of Poly$(S, f)(\mathbf{x})$ on all $\mathbf{x}$ in $\{0, 1\}^n$ are determined by the values at a small set of such $\mathbf{x}$, as we will show. Let $\mathbb{R}$ be the set of real numbers.

DEFINITION 5. Suppose $\mathbf{x}, \mathbf{y} \in \{0, 1\}^n$. We say $\mathbf{x} \leqq \mathbf{y}$ if for $i = 1, 2, \cdots, n$, $x_i \leqq y_i$.

DEFINITION 6. If $\mathbf{x}$ is an n-tuple of real numbers, then $\|\mathbf{x}\|$ is $\sum_{i=1}^{n} |x_i|$.

THEOREM 5. *Suppose S is a set of 3-literal clauses over $x_1, x_2, \cdots, x_n$ and f is a weighting function for S. Suppose* Poly(S, f) *is not identically zero. Then there exists* $\mathbf{x} \in \{0, 1\}^n$ *such that* $\|\mathbf{x}\| \leqq 3$ *and such that* Poly$(S, f)(\mathbf{x}) \neq 0$.

Proof. Pick a nonzero term in Poly(S, f) having the smallest number of variables. Let $\mathbf{x}$ be chosen so that $x_i = 1$ if x_i occurs in this term and $x_i = 0$ otherwise. Then Poly$(S, f)(\mathbf{x}) \neq 0$ and $\|\mathbf{x}\| \leqq 3$.

COROLLARY. Poly(S, f) *is completely determined by the* $\binom{n}{3} + \binom{n}{2} + n + 1$ *values* Poly$(S, f)(\mathbf{x})$ *for* $\|\mathbf{x}\| \leqq 3$.

It follows that Poly$(S, f) \equiv 0$ if Poly$(S, f)(\mathbf{x}) = 0$, for all $\mathbf{x} \in \{0, 1\}^n$ with $\|\mathbf{x}\| \leqq 3$. In fact, if Poly(S, f) is not identically zero, then for *all* $\mathbf{y} \in \{0, 1\}^n$, there exists $\mathbf{x} \in \{0, 1\}^n$ such that $\|\mathbf{x} - \mathbf{y}\| \leqq 3$ and such that Poly$(S, f)(\mathbf{x}) \neq 0$. Thus interpretations giving nonzero values are "dense". It does not follow, however, that Poly$(S, f)(\mathbf{x}) \geqq 0$ for all $\mathbf{x} \in \{0, 1\}^n$ iff Poly$(S, f)(\mathbf{x}) \geqq 0$ for all $\mathbf{x} \in \{0, 1\}^n$ with $\|\mathbf{x}\| \leqq 3$. For example, let S be the set of $2\binom{n}{3}$ clauses over $x_1, x_2, \cdots, x_n$ in which $x_i \vee x_j \vee x_k$ and $\bar{x}_i \vee \bar{x}_j \vee \bar{x}_k$ occur in S for all i, j, k with $i < j < k$. Define f. by $f(C) = -1$ on clauses C of the form $\bar{x}_i \vee \bar{x}_j \vee \bar{x}_k$, and $f(C) = 1$ on clauses C of the form $x_i \vee x_j \vee x_k$. Suppose $n \geqq 7$. Then Poly$(S, f)(\mathbf{x}) > 0$ for all $\mathbf{x} \in \{0, 1\}^n$ with $\|\mathbf{x}\| \leqq 3$, but Poly$(S, f)(1, 1, \cdots, 1)$ is $-\binom{n}{3}$.

5. The unsatisfiable region of Euclidean space.

DEFINITION 7. Let M be the set of polynomials p with real coefficients over the variables $x_1, \cdots, x_n$, such that p can be expressed as a sum of terms of one of the following forms, for $i < j$ and $j < k$:

$$a_{ijk}x_ix_jx_k,$$

$$b_{ij}x_ix_j,$$

$$c_ix_i,$$

$$d.$$

Note that such a polynomial is specified by $\binom{n}{3} + \binom{n}{2} + n + 1$ real coefficients. We thus identify polynomials in M with points in N-dimensional Euclidean space, where $N = \binom{n}{3} + \binom{n}{2} + n + 1$. Usually we are interested in the set of rational coefficient polynomials of M.

We would like to know which region of $\mathbb{R}^N$ corresponds to polynomials $p \in M$ such that $p(\mathbf{x}) > 0$ for all $\mathbf{x} \in \{0, 1\}^n$. Such polynomials represent inconsistent sets of clauses. Therefore we have the following definition.

DEFINITION 8. Let UNSATP be $\{p \in M: p(\mathbf{x}) > 0 \text{ for all } \mathbf{x} \in \{0, 1\}^n\}$. We also use UNSATP to refer to the corresponding subset of $\mathbb{R}^N$.

THEOREM 6. UNSATP *is a convex polyhedral cone. That is,* UNSATP *is the intersection of* 2^n *halfspaces in* $\mathbb{R}^N$. *Also, if* $\mathbf{z} \in$ UNSATP *then* $a\mathbf{z} \in$ UNSATP *for all* $a > 0$.

Proof. Let $\mathbf{w}$ be an element of $\{0, 1\}^n$. Suppose p is a polynomial in UNSATP. Then $p(\mathbf{w})$ is a sum of coefficients of p. Hence $\{p \in M: p(\mathbf{w}) > 0\}$ is a halfspace of $\mathbb{R}^N$. Therefore UNSATP is the intersection of 2^n halfspaces in $\mathbb{R}^N$. Since each halfspace is convex, so is UNSATP. Also, if $p(\mathbf{w}) > 0$ then $ap(\mathbf{w}) > 0$ for all $a > 0$. Hence $p \in$ UNSATP implies $ap \in$ UNSATP for all $a > 0$.

6. Inference rules. We now show how the polynomials associated with sets of clauses can be used to obtain more inference rules for unsatisfiability. That is, we obtain inference rules that can be used to show that a set of clauses is unsatisfiable. It is conceivable that the use of these rules, together with other inference rules such as resolution, will make possible much shorter proofs than are possible without using these rules. Therefore, this work is closely related to the NP vs. CoNP question.

We use GE(p_1, p_2) to abbreviate $(\forall \mathbf{x} \in \{0,1\}^n) p_1(\mathbf{x}) \geqq p_2(\mathbf{x})$. Also, the polynomial whose value is the constant k is written k. Thus GE(p, 1) means $(\forall \mathbf{x} \in \{0,1\}^n) p(\mathbf{x}) \geqq 1$. Further, if f is a weighting function for a set S of clauses, then $f \geqq k$ abbreviates $(\forall C \in S) f(C) \geqq k$. Similarly, $f > k$ abbreviates $(\forall C \in S) f(C) > k$. Note that a set S of clauses is inconsistent iff $(\exists f)$GE(Poly(S, f), 1). We introduce inference rules involving expressions of the form GE(p, q). For rules 1.6 and 1.8, assume that $f(C)$ is an integer for all $C \in S$.

TABLE 1
List of inference rules.

Group 1	1.	Poly($S, f_1 + f_2$) = Poly(S, f_1) + Poly(S, f_2)
	2.	Poly(S, kf) = $k *$ Poly(S, f)
	3.	$S1 \cap S2 = \varnothing \supset$ Poly($S1 \cup S2, f$) = Poly($S1, f$) + Poly($S2, f$)
	4.	(S is inconsistent) iff $(\exists f)$ GE(Poly(S, f), 1)
	5.	(S is inconsistent) iff $(\exists f)$ $f \geqq 0 \wedge$ GE(Poly(S, f), 1)
	6.	$f > 0 \supset$ [(S is inconsistent) iff GE(Poly(S, f), 1)]
	7.	$S1 \subset S2 \wedge f > 0 \supset$ GE(Poly($S2, f$), Poly($S1, f$))
	8.	$f_1 > 0 \wedge f_2 > 0 \supset$ [GE(Poly(S, f_1), 1) $\equiv$ Ge(Poly(S, f_2), 1)]
	9.	$f_1 > 0 \wedge f_2 > 0 \wedge$ Poly($S1, f_1$) = Poly($S2, f_2$) $\supset S1 \equiv S2$
	10.	$S1 \equiv S2 \supset$ ($S \cup S1$ is inconsistent) iff ($S \cup S2$ is inconsistent)
Group 2	1.	GE(p, p)
	2.	GE(p, q) $\wedge$ GE(q, r) $\supset$ GE(p, r)
	3.	GE(p_1, q_1) $\wedge$ GE(p_2, q_2) $\supset$ GE($p_1 + q_1, p_2 + q_2$)
	4.	GE(p, q) iff GE($-q, -p$)
	5.	GE(q_1, 0) $\wedge$ GE(q_2, 0) $\wedge$ GE(p_1, q_1) $\wedge$ GE(p_2, q_2) $\supset$ GE($p_1 * p_2, q_1 * q_2$)
	6.	$k_1 \geqq 0 \wedge k_2 \geqq 0 \wedge$ GE(p_1, k_1) $\wedge$ GE(p_2, k_2) $\supset$ GE($p_1 * p_2, k_1 * k_2$)
	7.	$k > 0 \supset$ [GE(p, q) $\equiv$ GE(kp, kq)]
	8.	GE(q, 1) $\supset$ [GE(p_1, p_2) $\equiv$ GE($p_1 * q, p_2 * q$)]
	9.	GE(x_i, 0) for $1 \leqq i \leqq n$ and GE($1 - x_i$, 0) for $1 \leqq i \leqq n$
	10.	GE(x_i, x_i^k) for $1 \leqq i \leqq n, k > 0$
	11.	GE(x_i^k, x_i) for $1 \leqq i \leqq n, k > 0$
Group 3	1.	$S1 \sim S2 \supset S2 \sim S1$
	2.	$p_1 \sim p_2 \supset p_2 \sim p_1$
	3.	$S1 \sim S2 \supset$ [($S1$ is inconsistent) iff ($S2$ is inconsistent)]
	4.	$p_1 \sim p_2 \supset k * p_1 \sim k * p_2$
	5.	GE(p_1, k) $\wedge$ $p_1 \sim p_2 \supset$ GE(p_2, k)
	6.	$S1 \sim S2 \wedge f_1 > 0 \supset$ [$(\exists f_2) f_2 > 0 \wedge$ Poly($S1, f_1$) $\sim$ Poly($S2, f_2$)]

We now illustrate ways in which these rules can be used. Suppose $S1$ is inconsistent and $S1 \subset S2$. Then by 1.6, GE(Poly(S1, 1), 1). Also, by 1.7, GE(Poly($S2$, 1), Poly($S1$, 1)). Hence by 2.2, GE(Poly($S2$, 1), 1). Hence by 1.6, $S2$ is inconsistent. Thus we only need to worry about minimal inconsistent sets of clauses. These can be reduced in number by 3.3. In addition, from 2.5, 2.7, and 2.9 it follows that GE(p, 0) is true if all coefficients of p are nonnegative. Also, it follows from 2.5, 2.7, and 2.9 that GE(Poly(S, f), 0) for all S if $f \geqq 0$. Suppose $S1$ is a minimal inconsistent set of clauses, and for some weighting function f_1, Poly($S1, f_1$) = Poly($S2, f_2$) + p where $f_2 > 0$ and GE(p, 0) is known. Suppose $S2$ is known to be inconsistent. Then it follows by 1.6 that

GE(Poly($S2, f_2$), 1), and by 2.3 that GE(Poly($S1, f_1$), 1), and by 1.4 that $S1$ is inconsistent. Hence we may be able to exhibit short proofs of inconsistency of minimal inconsistent sets of clauses by methods other than isomorphism. Also, it could be that distinct minimal inconsistent sets $S1$ and $S2$ of clauses will have the same polynomials Poly($S1, f_1$) = Poly($S2, f_2$), and in this way we may get short proofs of inconsistency. Finally, the rules 2.10 and 2.11 can be used to eliminate powers of x_i higher than the first power after applying 2.5 or 2.6. The rules 2.5 or 2.6 will usually result in polynomials of degree higher than 3, even after such reduction in exponents has been done.

The following limited results concern minimal inconsistent sets of clauses.

THEOREM 7. *Suppose $S1$ and $S2$ are minimal inconsistent sets of clauses over $x_1, x_2, \cdots, x_k$. That is, no proper subset of $S1$ or $S2$ is inconsistent. Suppose $f_1 > 0$ and $f_2 > 0$ and* Poly($S1, f_1$) = Poly($S2, f_2$). *Then* $\min\{f_1(C): C \in S1\} = \min\{f_2(C): C \in S2\}$.

Proof. Let $C1 \in S1$ be a clause such that $f_1(C1)$ is minimal among $\{f_1(C): C \in S1\}$. Let $C2 \in S2$ be a clause such that $f_2(C2)$ is minimal among $\{f_2(C): C \in S2\}$. Since $S1$ is minimal inconsistent, $S1 - \{C1\}$ is consistent and so some interpretation makes all clauses in $S1 - \{C1\}$ true. Thus there exists $\mathbf{x} \in \{0, 1\}^n$ such that Poly($S1, f_1$)($\mathbf{x}$) = $f_1(C1)$. Hence Poly($S2, f_2$)($\mathbf{x}$) = $f_1(C1)$ also. Since Poly($S2, f_2$)($\mathbf{x}$) is a sum of weights of clauses in $S2$, $f_1(C1) \geqq f_2(C2)$. Similarly, $f_2(C2) \geqq f_1(C1)$.

THEOREM 8. *Suppose S is a minimal inconsistent set of clauses over $x_1, x_2, \cdots, x_n$. Suppose f is a weighting function. Then* GE(Poly(S, f), 1) *is true iff $f > 0$.*

Proof. If $f > 0$, GE(Poly(S, f), 1) follows because S is inconsistent. If for some $C \in S$, $f(C) \leqq 0$ then GE(Poly(S, f), 1) is false, as follows: Since S is minimal inconsistent, there is an interpretation in which C is false and all other clauses of S are true. Hence there exists $\mathbf{x} \in \{0, 1\}^n$ such that Poly(S, f)($\mathbf{x}$) = $f(C)$. Since $f(C) \leqq 0$, we cannot have GE(Poly(S, f), 1).

There is still another technique that may be applied to show inconsistency. Let f be a weighting function for S such that for no nonempty subset $\{C1, C2, \cdots, Ck\}$ of k distinct elements of S does $f(C1) + f(C2) + \cdots + f(Ck) = 0$. Such weighting functions can be obtained from instances of the knapsack problem that are known not to have a solution. And such instances can be obtained by polynomial time reductions from known inconsistent sets of clauses! In any event, if f is such a weighting function, and S is inconsistent, then $(\forall \mathbf{x} \in \{0, 1\}^n)$Poly($S, f$)($\mathbf{x}$) $\neq 0$. Hence if $S1$ is another set of clauses and f_1 is a weighting function for $S1$, and if Poly($S1, f_1$) = Poly(S, f), then $S1$ is inconsistent also. Such a function f need not satisfy $f \geqq 0$, and so we get a more general method than that of rules 1.4, 1.5, and 1.6.

Finally, it would be interesting to know whether there is a "small" set A_n of axioms from which the inconsistency of all inconsistent sets of 3-literal clauses over $x_1, \cdots, x_n$ can be shown by short proofs. These axioms would be of the form GE(Poly(S, f), 1) for various S and f or of the form GE(p, 0) for various p. If so, unsatisfiability could be decided in nondeterministic polynomial time relative to a "slowly utilized" oracle [7]. Along this line, how many distinct polynomials p are there in the set IP = {Poly(S, 1): S is a minimal inconsistent set of clauses over $x_1, \cdots, x_n$}? How many equivalence classes are there in this set under the relationship $p_1 \sim p_2$?

Not all of these equivalence classes are really necessary. Suppose we eliminate from IP all equivalence classes of polynomials p satisfying the following condition:

There exist $S1$, $S2$, f_1, f_2, q such that p = Poly($S1$, 1) and $S1$, $S2$ are minimal inconsistent sets of clauses and Poly($S1, f_1$) = Poly($S2, f_2$) + q and $f_2 > 0$ and it is known that GE(q, 0) is true.

If this condition is true, then given that $S2$ is known to be inconsistent we can construct a short proof that $S1$ is inconsistent. Hence GE(Poly($S1$, 1), 1) need not be kept as an axiom. The polynomial q may have nonnegative coefficients, or be of the form $\mathrm{Poly}(S, f) - 1$ where S is known to be inconsistent and $f > 0$. Also, we can eliminate from IP all equivalence classes of polynomials $\mathrm{Poly}(S, 1)$ such that S has a short resolution proof of inconsistency. How many equivalence classes are then left in IP? If this number is small, we might hope to get short proofs of inconsistency relative to a small number of axioms.

Conclusions. Polynomials with several variables give insight into the structure of unsatisfiable sets of clauses. The polynomials associated with sets of clauses seem to have properties that do not have any relationship to the difficulty of proving inconsistency of the sets of clauses. It is possible, therefore, that these polynomials will provide methods of obtaining short proofs of inconsistency. It turns out that polynomials of unsatisfiable sets of clauses correspond to a region of Euclidean space which is the intersection of 2^n halfspaces, for sets of clauses over n variables. Some inference rules based on these polynomials can be used to show that a set of clauses is unsatisfiable. Several problems associated with these polynomials have polynomial time solutions.

REFERENCES

[1] S. A. COOK, *The complexity of theorem proving procedures*, Proceedings of Third Annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[2] S. A. COOK AND R. RECKHOW, *On the lengths of proofs in the propositional calculus*, Proceedings of Sixth Annual ACM Symposium on Theory of Computing, 1974, pp. 135–148.

[3] M. DAVIS AND H. PUTNAM, *A computing procedure for quantification theory*, J. Assoc. Comput. Mach., 7 (1960), pp. 201–215.

[4] Z. GALIL, *On the complexity of regular resolution and the Davis-Putnam procedure*, Theoret. Comput. Sci., 4 (1977), pp. 23–46.

[5] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[6] L. G. KHACHIAN, *Polynomial algorithm in linear programming*, Dokl. Akad. Nauk SSSR, 244 (1979), pp. 1093–1096.

[7] D. PLAISTED, *New* NP-*hard and* NP-*complete polynomial and integer divisibility problems*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 241–253.

[8] ———, *Some polynomial and integer divisibility problems are* NP-*hard*, SIAM J. Comput., 7 (1978), pp. 458–464.

[9] ———, *Sparse complex polynomials and polynomial reducibility*, J. Comput. System Sci., 14 (1977), pp. 210–221.

[10] J. A. ROBINSON, *A machine oriented logic based on the resolution principle.* J. Assoc. Comput. Mach., 12 (1965), pp. 23–41.

[11] G. S. TSEITIN, *On the complexity of derivation in propositional calculus*, in Studies in Constructive Mathematics and Mathematical Logic, Part II, A. O. Slisenko, ed., (translated from Russian), Consultants Bureau, New York, 1969, pp. 115–125.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

CONSTANT TIME GENERATION OF ROOTED TREES*

TERRY BEYER† AND SANDRA MITCHELL HEDETNIEMI‡

Abstract. This paper generalizes a result of Ruskey [SIAM J. Comput., 7(1978), pp. 424–439] for generating k-ary trees lexicographically to generating all rooted trees with n vertices. An algorithm is presented which generates canonical representations of these trees in a well-defined order. As in other works, the average number of steps per tree is constant.

Key words. rooted tree generation, lexicographic order, algorithm

1. Introduction. Interest has arisen recently in generating random trees, all binary trees, and all k-ary trees. The present work has centered on rooted trees, that is, those trees with a designated vertex as the root. The algorithm of Nijenhuis and Wilf [3] provides a method for generating random trees, but it does not provide for a systematic generation of all trees. Read [5] has formulated an algorithm to generate all trees on n vertices. This algorithm unfortunately must process trees on $n-1$ vertices.

The generation of all binary trees by Zaks [10] and Ruskey and Hu [7] uses "feasible" sequences which are altered so as to remain "feasible" and create the "next" tree in lexicographic order. Both Zaks [11] and Ruskey [6] have generalized this technique to k-ary trees. The feasible sequences which Zaks manipulates are related to the degrees of the vertices in the tree ordered by a preorder traversal of the tree. The sequences which Ruskey employs in his algorithm contain the level numbers of the endvertices in the tree; where an endvertex is a vertex of degree 1 and the level of a vertex is the number of vertices on the path from the root to an endvertex, counting both the root and the endvertex in the sum. The algorithm for generating k-ary trees with n leaves in lexicographic order is $O(k)$ per sequence generated.

Trojanowski [9] has presented another algorithm for generating all k-ary trees which manipulate tree permutations.

Zaks [10] has extended his result to the case of a general tree. His feasible sequences contain n entries for a tree of n vertices, and the work in generating the next sequence is $O(1)$.

In this paper we generalize the work of Ruskey to the generation of all rooted trees on n vertices. The feasible sequences used to generate the trees contain an entry of the level number for all vertices in the tree. The average number of computational steps per tree generated is shown to be uniformly bounded by a constant, independent of n.

2. Level representation of rooted, ordered trees. A tree T is *rooted* if there exists a distinct vertex v designated the *root*. The remaining vertices are partitioned into disjoint sets $T_{u_1}, T_{u_2}, \cdots, T_{u_k}$ which are subtrees of v rooted at $u_1, u_2, \cdots, u_k$, respectively. A rooted tree is said to be *ordered* if there exists a linear order imposed upon the subtrees.

Define the *level* of a vertex v, l_v, to be 1 if v is the root; or one greater than the level of its parent, where v's *parent* is the adjacent vertex on the path from v to the root. A *level sequence* $L(T)=[l_1l_2\cdots l_n]$ for a given rooted, ordered tree with n vertices is obtained by traversing T in preorder (cf. Knuth [2]), and recording the level of each vertex as it is visited. T_i, the subtree of T rooted at vertex i, is described by a contiguous

* Received by the editors January 24, 1979, and in final revised form January 31, 1980.

† Computing Center and Department of Computer and Information Science, University of Oregon, Eugene, Oregon, 97403.

‡ Department of Computer and Information Science, Universtity of Oregon, Eugene, Oregon, 97403.

sequence of $L(T)$, denoted $L(T_i)$, which begins with l_i and ends just before the first element, if any, which is less than or equal to l_i (cf. Fig. 1).

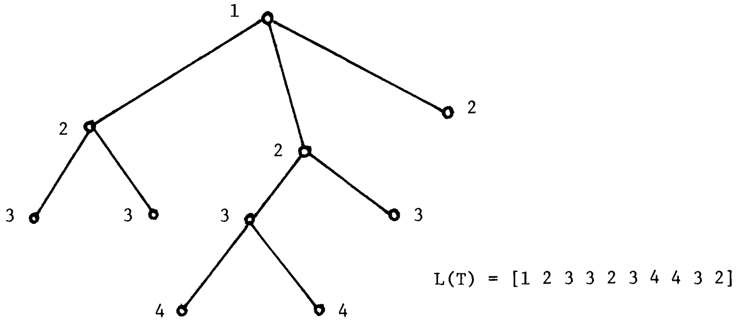


FIG. 1. *An ordered, rooted tree T, with vertex levels shown, and its level sequence.*

The level sequence provides a linear order of the subtrees (cf. Scoins [7]). Given two ordered trees T_1 and T_2, T_1 *dominates* T_2, denoted $T_1 > T_2$, if $L(T_1) > L(T_2)$ in the lexicographic ordering of integer sequences. It follows from the definition that any two trees having the same level sequence are isomorphic.

Two subtrees T_i and T_j are said to be *adjacent subtrees* of T if vertices i and j are consecutive children of the same parent. (Note: $L(T_i)$ and $L(T_j)$ are consecutive subsequences of $L(T)$.) If vertex i is a child of the root of T, T_i is a *principal subtree of T.*

3. Canonical representation of rooted trees. Given a rooted tree T, there exist many non-isomorphic ordered trees corresponding to T (and hence many level sequences). We use the following to ensure a canonical ordering of any rooted tree (i.e. a specific ordering of the subtrees).

The canonical ordering of T will be that ordered tree T^* which dominates all other ordered trees T' corresponding to T, i.e. $L(T^*) > L(T')$ for all T' corresponding to T. We denote the canonical level sequence by $L(T)^*$. Fig. 2 illustrates two trees T_1 and T_2 which correspond to the same underlying rooted tree T. Since these are the only distinct trees corresponding to T, and since $L(T_1) > L(T_2)$, $L(T)^* = [1\ 2\ 3\ 3\ 2]$, and $T_1 = T^*$.

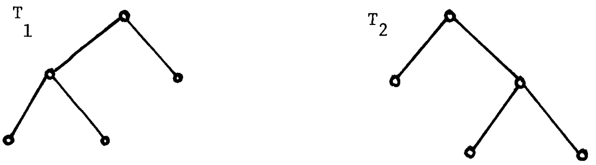


FIG. 2. *Two ordered trees with T_1 dominating T_2.*

A level sequence $L(T)$ of a rooted, ordered tree T is *regular* if every pair of adjacent subtrees T_i and T_j, $i < j$, appear in decreasing order of dominance, i.e. $L(T_i) \geqq L(T_j)$.

LEMMA 1. *An ordered tree T is the canonical ordering of its underlying rooted tree T_R, if and only if, $L(T)$ is regular.*

Proof. Assume $L(T_R)^* = L(T)$ but $L(T)$ is not regular. Then there exist adjacent subtrees T_i and T_j such that $i < j$ and $L(T_i) < L(T_j)$. Let T' be the ordered tree obtained by exchanging T_i and T_j. Then $L(T') > L(T)$ which contradicts $L(T_R)^* = L(T)$.

By induction we show that any rooted tree has at most one corresponding regular sequence. For a tree with one vertex this follows immediately. Assume that all trees with less than n vertices have at most one regular sequence. Let $L(T)$ be a regular sequence for a rooted tree with n vertices. $L(T)$ consists of a 1 followed by subsequences corresponding to the principal subtrees of T. Let T_i be a principal subtree. Subtracting 1 from each element in $L(T_i)$ results in a level sequence $L'(T_i)$ for the rooted tree T_i. Since $L(T)$ is regular, $L'(T_i)$ is also regular. But by the inductive hypothesis, $L'(T_i)$ is the unique regular sequence for T_i. Furthermore, the relative order of the subsequences is unique up to isomorphism since $L(T)$ is regular. Therefore, $L(T)$ is the only regular sequence representing the rooted tree T.

LEMMA 2. *If $L(T)$ is regular, and two consecutive elements have a value* 2, *then all the remaining elements have a value* 2.

Proof. Let $l_i = l_{i+1} = 2$ in the regular sequence $L(T)$. Then l_i must represent a principal subtree of T consisting of a single vertex. Since the principal subtrees appear in lexicographic order, and since 2 is the least possible sequence for a subtree, all following subtrees must be represented by 2's.

4. Successor function. One conceivable method for generating all rooted trees on n vertices would be to generate, in lexicographically decreasing order, the level sequences for all ordered trees on n vertices and filter out those which are not regular. We will now show, however, that it is possible to define a simple successor function which allows us to pass directly from one regular sequence to the next.

Let $L(T) = [l_1 l_2 \cdots l_n]$ be a level sequence containing an element greater than 2. Let p be the position of the rightmost such element. Let q be the rightmost position preceding p such that $l_q = l_p - 1$. Note that the vertex corresponding to l_q is the parent of the vertex corresponding to l_p. Define the successor of $L(T)$ to be the sequence $s(L(T)) = [s_1, s_2 \cdots s_n]$, where

(i) $s_i = l_i$ for $i = 1, 2, \cdots, p-1$,

(ii) $s_i = s_{i-(p-q)}$ for $i = p, \cdots, n$.

The relationships between $L(T)$ and $s(L(T))$ can be seen from the following lines:

$$\begin{aligned} L(T) &= [l_1 \cdots l_q \cdots l_{p-1}\ l_p\ 2\ 2\ 2 \cdots] \\ s(L(T)) &= [l_1 \cdots \underbrace{l_q \cdots l_{p-1}}_{T_q}\ \underbrace{l_q \cdots l_{p-1}}_{T_q}\ \underbrace{l_q \cdots}_{T_q}]. \end{aligned}$$

The subsequence labeled T_q represents a subtree of $s(L(T))$ which is repeated as many times as possible, concluding with a partial repetition if necessary to reach a total of n vertices.

Here are some examples of canonical representations and their successors.

$L(T)$	$s(L(T))$
[1 2 3 2 2 2]	[1 2 2 2 2 2]
[1 2 3 4 2 2]	[1 2 3 3 3 3]
[1 2 3 4 3 2 2 2 2 2 2]	[1 2 3 4 2 3 4 2 3 4 2 3]
[1 2 3 4 5 5 2 2 2 2]	[1 2 3 4 5 4 5 4 5 4]

LEMMA 3. *If $L(T)$ is regular and contains an element greater than* 2, *then $s(L(T))$ is also regular.*

Proof. Let $i<j$ be any two indices corresponding to adjacent subtrees T_i and T_j in $s(L(T))$. Let p denote the index of the rightmost value which is not 2, and let q denote the index of the corresponding vertex's parent. If $q \leqq i$, then T_i and T_j are adjacent subtrees in the sequence $T_q, T_q, T_q, \cdots$, from which it follows that $T_i \geqq T_j$. If $i<q$ and $L(T_j)$ does not overlap position p, then T_i and T_j are also adjacent subtrees in the regular sequence $L(T)$, and hence $T_i \geqq T_j$. In the remaining case, $i<q$ and T_j does overlap position p. In this case, the subsequences representing T_i and T_j are the same in $L(T)$ as in $s(L(T))$ up to position p. In that position the value for vertex p is one less in $s(L(T))$ than it was in $L(T)$. Hence we still have $T_i \geqq T_j$.

LEMMA 4. *Let $L(T)$ be a regular sequence of length $n \geqq 2$. If $L(T)$ is of the form* $[1\ 2\ 2 \cdots 2]$, *then it is the lexicographically least sequence of length n. Otherwise, $s(L(T))$ is defined and is the first regular sequence following $L(T)$ in the lexicographic ordering.*

Proof. The fact that $[1\ 2\ 2 \cdots 2]$ is the least sequence is immediate. By Lemma 3, $s(L(T))$ is regular. It remains to show that no regular sequence is lexicographically between $L(T)$ and $s(L(T))$. Let $L(T)=[l_1 \cdots l_n]$, $s(L(T))=[s_1 \cdots s_n]$, $L(T')=[m_1 \cdots m_n]$, and assume that $L(T)>L(T')>s(L(T))$. Let p, q, and T_q be as previously defined. Since $l_i=s_i$ for all $1 \leqq i \leqq p$, all three sequences are identical in the first $p-1$ positions. If $l_p=m_p$, then $L(T')=L(T)$, since $l_i=2$ for $i \geqq p+1$, and since the only 1 in $L(T')$ appears in position 1. But since $L(T') \neq L(\Gamma)$, $l_p \neq m_p$. But $s_p=l_p-1$. Hence, since $m_i=s_i$ for all $i<p$ and $L(T)>s(L(T))$, we have $m_p=s_p$. Thus the first position, r, in which $L(T')$ and $s(L(T))$ differ, is such that $r>p$. But with respect to $s(L(T))$, this must occur in one of the copies of T_q with another copy of T_q to its left. Thus in the sequence $[m_1 \cdots m_n]$ there are two adjacent subtrees not in lexicographic order. That is, $L(T')$ is not regular.

5. Generating algorithm. An algorithm to generate all rooted trees on N vertices begins with the lexicographically greatest possible regular sequence, then passes from one regular sequence to the next using the successor function until finally the lexicographically least sequence has been generated.

We present in Fig. 3 a program to implement this algorithm written in FLECS [1], a structured extension of Fortran. Variables L, N, and P correspond to the level sequence, the number of vertices, and the position p in § 4, respectively. The array PREV is used to quickly find the values of q as given in § 4. If value i appears in array L to the left of position P, then PREV(i) is the index of the rightmost such appearance; otherwise, PREV(i) = 0. SAVE is used to efficiently update PREV when adjusting P.

6. Analysis of complexity. The average computational effort per tree generated is bounded by a constant which is independent of N. That is, there is a constant K such that given $N \geqq 1$, the effort expended in generating all $A(N)$ rooted trees on N vertices is less than $K * A(N)$.

For $N \geqq 4$, the number of rooted trees is greater than N. Hence the cost of generating the first tree, which is linear in N, can be ignored. It remains to analyze the total cost of generating the next tree over all $A(N)$ invocations.

Since initially $P=N$ and finally $P=1$, P undergoes a total change in value of $N-1$. But P is increased once for each execution of Line 31, and is decreased once for each execution of Line 36, and is not altered elsewhere. Hence, it remains to show that the total number of decrements is $\leqq K * A(N)$ for some constant K.

On entry to GENERATE-NEXT-TREE, L contains a regular sequence and P indicates its rightmost non-2 element. By Lemma 2, there cannot be two consecutive 2's to the left of position P. Thus, if the condition in Line 26 is false, execution passes

```
 1   SUBROUTINE GENRT (N, L, PREV, SAVE)
 2
 3   INTEGER N, L(N), PREV(N), SAVE(N)
 4   INTEGER P, DIFF, I
 5
 6   GENERATE-FIRST-TREE
 7   CALL PROCESS(N,L)
 8   WHILE (P .GT. 1)
 9   .  GENERATE-NEXT-TREE
10   .  CALL PROCESS(N,L)
11   ...FIN
12   RETURN
13
14   TO GENERATE-FIRST-TREE
15   .  DO (I+1,N) L(I) = I
16   .  WHEN (N .LE. 2) P = I
17   .  ELSE P = N
18   .  IF (P .GT. 1)
19   .  .  DO (I=1,P-1)  PREV(I) = I
20   .  .  DO (I=1,P-1)  SAVE(I) = 0
21   .  ...FIN
22   ...FIN
23
24   TO GENERATE-NEXT-TREE
25   .  L(P) = L(P) - 1
26   .  IF (P .LT. N  .AND.  (L(P) .NE. 2  .OR.  L(P-1) .NE. 2) )
27   .  .  DIFF = P - PREV( L(P) )
28   .  .  WHILE (P .LT. N)
29   .  .  .  SAVE(P) = PREV( L(P) )
30   .  .  .  PREV( L(P) ) = P
31   .  .  .  P = P + 1
32   .  .  .  L(P) = L(P-DIFF)
33   .  .  ...FIN
34   .  ...FIN
35   .  WHILE (L(P) .EQ. 2)
36   .  .  P = P - 1
37   .  .  PREV( L(P) ) = SAVE (P)
38   .  ...FIN
39   ...FIN
40
41   END
```

FIG. 3. *A program to generate rooted trees.*

directly to Line 35 and no more than two iterations of the loop occur. If the condition in Line 26 is true, then repeatedly a copy is made to the right of the subsequence from the left of the subsequence of a subsequence which does not have two consecutive 2's in it. In this case, the loop in Lines 35–38 will be executed at most once. In either case, the loop is executed at most two times per invocation. Thus, the number of decrements of P is $\leqq K * A(N)$.

Although the proof shows that on the average no more than two times is the loop executed per tree generated, a more complicated proof could be given which lowers this bound. Empirically, we have found that the average number of executions decreases as N increases. For $N=7$ the loops are executed .6 times per tree generated, and by $N=11$ this number is down to .5.

A variant of Subroutine GENRT has been executed on a DEC-System-10 (KA-processor), and found to generate roughly $\frac{1}{2}$ million trees per minute.

7. Computation of $A(N)$. It is significant to enquire exactly how many rooted trees exist and therefore will be generated upon execution of Subroutine GENRT for a given input value of N; i.e. what is $A(N)$?

In Knuth [2], the following recursive function of Otter [4] is presented for calculating this number:

$$N * A(N+1) = A(1) * S(N,1) + 2 * A(2) * S(N,2) + \cdots + N * A(N) * S(N,N),$$

where:

$$S(N, K) = \sum_{1 \leq j \leq N/K} [A(N+1-J+K)].$$

The $S(N, K)$ terms can be calculated very simply using iterations where the step size is dependent upon J and K. Table 1 presents the values $A(N)$ for the first 10 values of N. It also displays the number of trees of various height that are generated; Subroutine GENRT, in fact, generates the trees according to height.

Although it is common in the literature (cf. [6]–[11]) to provide ranking and unranking algorithms based on the lexicographic order, Table 1 illustrates that this might be quite difficult. Unranking algorithms are often used to generate a random tree. Since the process of generating trees is quite efficient, a suitable alternative would be to simply stop the generative process at a randomly determined time.

TABLE 1.
A catalog of rooted unordered trees having fewer than ten vertices.

Number of Vertices	$A(N)$	Number of trees of DEPTH									
		1	2	3	4	5	6	7	8	9	10
1	1	1									
2	1	0	1								
3	2	0	1	1							
4	4	0	1	2	1						
5	9	0	1	4	3	1					
6	20	0	1	6	8	4	1				
7	48	0	1	10	18	13	5	1			
8	115	0	1	14	38	36	19	6	1		
9	286	0	1	21	46	113	61	26	7	1	
10	719	0	1	29	147	225	180	94	34	8	1

7. Concluding remarks. The basic level sequence generation algorithm presented in this paper has also been adapted to the generation of unrooted (free) trees. The resulting algorithm also appears to have the constant time per tree property. An implementation of this algorithm generates roughly $\frac{1}{4}$ million trees per minute on the DEC-System-10 (KA-processor).

Read [5] has proposed a very general mechanism for generating classes of combinatorial objects. In the case of rooted trees, the method presented in this paper appears to be superior because of both its actual speed and its constant time per tree property. Read's method would not appear to have the constant time per tree property, since it involves processing representations on $n-1$ vertices in order to get representations on n vertices.

Acknowledgments. We wish to thank Professors Steve Hedetniemi and Andrzej Proskurowski for encouragement and stimulation.

REFERENCES

[1] T. BEYER, FLECS: *User Manual*, Computing Center, University of Oregon, Eugene OR, 1975.

[2] D. KNUTH, *Fundamental Algorithms*, Addison-Wesley, Reading MA, 1968.

[3] A. NIJENHUIS AND H. WILF, *Combinatorial Algorithms*, Academic Press, New York, 1975.

[4] R. OTTER, *The number of trees*, Ann. Math., 49 (1948), pp. 583–599.

[5] R. READ, *How to grow trees*, in Combinatorial Structures and their Applications, Gordon and Breach, New York, 1970.

[6] F. RUSKEY, *Generating t-ary trees lexicographically*, this Journal, 7 (1978), pp. 424–439.

[7] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, this Journal, 6 (1977), pp. 745–758.

[8] H. SCOINS, *Placing trees in lexicographic order*, Machine Intelligence, 3 (1969), pp. 43–60.

[9] A. TROJANOWSKI, *Ranking and listing algorithms for k-ary trees*, this Journal, 7 (1978), pp. 492–509.

[10] S. ZAKS, *Lexicographic generation of ordered trees*, Theoret. Comput. Sci., 10 (1980), pp. 63–82.

[11] ———, *Generating k-ary Trees Lexicographically*, University of Illinois Tech. Rept. UICSCS-R 77-901, Urbana IL, 1977.

[12] S. ZAKS AND D. RICHARDS, *Generating trees and other combinatorial objects lexicographically*, this Journal, 8 (1979), pp. 73–81.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0007 $01.00/0

ON THE COMPLEXITY OF BILINEAR FORMS WITH COMMUTATIVITY*

JOSEPH JA'JA'†

Abstract. We consider the general problem of computing sets of bilinear forms in commuting indeterminates. We develop lower bound techniques which seem to be more powerful than those already known in the literature. We show that duality theory as it is known for bilinear forms with noncommuting indeterminates does *not* hold in the commutative case; we prove that the multiplication of $2 \times n$ by'$n \times 2$ matrices requires at least $\lceil 27n/8 \rceil$ multiplications while it is possible to multiply 2×2 by $2 \times n$ matrices using only $3n+2$ multiplications. Moreover we settle the question of whether commutativity can reduce the number of multiplications by a factor of $\frac{1}{2}$, by showing that this can never happen. We also show that, over algebraically closed fields, the complexity of computing a pair of bilinear forms is the same whether or not commutativity is allowed.

Key words. algebraic complexity, bilinear forms, matrix multiplication, tensor rank

1. Introduction. The problem of computing sets of bilinear forms has received considerable attention in recent years (e.g., [2], [5], [6], [8], [9], [12], [13], [14], [15]). This class includes many important problems such as the multiplication problems of matrices and polynomials. In this paper, we shall investigate the multiplicative complexity of a set of bilinear forms with commuting indeterminates as it relates to the same problem with noncommuting indeterminates. We will establish the fact that duality theory does *not* hold in the commutative case by showing that the multiplication of $2 \times n$ by $n \times 2$ matrices requires at least $\lceil 27n/8 \rceil$ multiplications over the integers, while it is possible to multiply 2×2 by $2 \times n$ matrices (or $n \times 2$ by 2×2) using only $3n+2$ multiplications with integer coefficients [20], [21]. On the other hand, we show that, over algebraically closed fields, the complexity of computing a pair of bilinear forms is the same regardless of whether or not commutativity is allowed. We feel that, in fact, commutativity will have little effect in general whenever the constants are drawn from an algebraically closed field. We also develop lower bound techniques which seem to be more powerful than those already known for the commutative case [15]. Moreover, we settle the question of whether commutativity can reduce the number of multiplications required to compute a set of bilinear forms by a factor of $\frac{1}{2}$ by showing that this can never happen.

2. Preliminaries. We will review quickly some of the formulations of bilinear complexity known in the literature, as they will be used freely in later sections. The general problem of computing a set of bilinear forms can be defined as follows. Let K be a commutative ring (with a unit element) 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 $m \geqq 1$ bilinear forms

$$B_i = \sum_{j=1}^{p} \sum_{k=1}^{q} r_{ijk} x_j y_k = x^T G_i y, \qquad i = 1, 2, \cdots, m, \quad r_{ijk} \in K,$$

where G_i is a $p \times q$ matrix with elements in K. Our model of computation consists of the class of *bilinear programs*, where a bilinear program consists of a sequence of instructions of the form $f_j \leftarrow a_j \circ b_j$, $1 \leqq j \leqq l$, $\circ \in \{+, -, \times\}$, each f_j is a new variable, and each a_j or b_j is either (1) an f_t, $t < i$, or (2) an indeterminate or (3) a constant from K. All the active

* Received by the editors March 17, 1979, and in revised form September 10, 1979.

† Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802. This research was supported in part by the National Science Foundation under Grant MCS 78-06118.

multiplications are multiplications between linear forms of x and y with coefficients from K. It is well known that it is no loss of generality to consider this class of algorithms rather than the more general class of straight-line programs [15], [22]. Based on these facts, we introduce the following definition.

DEFINITION. Given a set of bilinear forms $\{B_i\}_{i=1}^m$ over a commutative ring K, the *commutative complexity* of the set $\{B_i\}_{i=1}^m$ is the smallest integer μ such that

$$B_i = \sum_{l=1}^{\mu} \alpha_{il} f_l(x, y) f'_l(x, y), \qquad 1 \leqq i \leqq m,$$

where $f_l(x, y)$ and $f'_l(x, y)$ are linear forms in x and y over K.

If we assume that the indeterminates do not commute, then we can restrict the active multiplications to be of the type $f(x)f'(y)$, where $f(x)$ and $f'(y)$ are linear forms in x and y respectively.

DEFINITION. Given a set of bilinear forms $\{B_i\}_{i=1}^m$ over a commutative ring K, the *noncommutative complexity* (or simply, *complexity*) is the smallest integer δ such that

$$(*) \qquad B_i = \sum_{j=1}^{\delta} \beta_{ij} f_j(x) f'_j(y), \qquad 1 \leqq i \leqq m,$$

where $f_j(x)$ and $f'_j(y)$ are linear forms in x and y respectively.

From now on, we will use the notation $\delta\{B_i\}$ or $\delta\{G_i\}$ to mean the noncommutative complexity of $\{B_i\}_{i=1}^m$, and $\mu\{B_i\}$ or $\mu\{G_i\}$ to mean the commutative complexity of the same set. Since studies in bilinear complexity nearly always have dealt with the case of noncommutative indeterminates only, we will try to extend the techniques already known in this field to get results for the commutative case. We now recall these results which will be needed later.

Let $f_j(x) = \langle b_j, x\rangle$ and $f'_j(y) = \langle c_j, y\rangle$ in (*); we have

$$B_i = x^T G_i y = \sum_{j=1}^{\delta} \beta_{ij} \langle b_j, x\rangle \langle c_j, y\rangle = x^T\left(\sum_{j=1}^{\delta} \beta_{ij} b_j c_j^T\right) y, \qquad i = 1, 2, \cdots, m.$$

Since the above equality must hold for all values of the indeterminates x and y over K, we conclude that

$$G_i = \sum_{j=1}^{\delta} \beta_{ij} b_j c_j^T, \qquad i = 1, 2, \cdots, m.$$

Therefore, δ is equal to the smallest number of rank one matrices necessary to include the G_i's in their span [2], [6], [14].

Another interesting formulation given in [2] and [14] is obtained by introducing a set of indeterminates $\{s_i\}_{i=1}^m$ to yield the trilinear form

$$h(s, x, y) = \sum_{i=1}^{m} s_i B_i = \sum_{i=1}^{m} \sum_{j=1}^{p} \sum_{k=1}^{q} r_{ijk} s_i x_j y_k.$$

It is easy to see that δ is the smallest number such that

$$h(s, x, y) = \sum_{l=1}^{\delta} \langle \beta_l, s\rangle \langle b_l, x\rangle \langle c_l, y\rangle,$$

and we now have a completely symmetric problem with respect to s, x and y; for example, the above problem is equivalent to that of computing p bilinear forms associated with the $m \times q$ matrices

$$G_j = (r_{ijk})_{ik}, \qquad j = 1, 2, \cdots, p.$$

This property, referred to as *duality* in the literature, was also discovered independently by Hopcroft and Musinski [8], Probert [16] and Strassen [17]. 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. As we will see later, this property does *not* hold in the commutative case.

Another definition, which we will find quite useful, is that of the *characteristic matrix* $G(s) = \sum_{i=1}^{m} s_i G_i$ associated with the set $B_i = x^T G_i y$, $i = 1, 2, \cdots, m$, where the $\{s_i\}_{i=1}^{m}$ are, as before, a set of indeterminates. The corresponding commutative and noncommutative complexities will be respectively denoted by $\delta\{G(s)\}$ and $\mu\{G(s)\}$. Note that δ is the smallest number such that

$$G(s) = \sum_{j=1}^{\delta} \langle \beta_j, s\rangle b_j c_j^T, \qquad B_j \in K^m,\ b_j \in K^p \text{ and } c_j \in K^q.$$

An interesting observation made in [2] and [4] is that δ is invariant under the action of the group $\mathscr{G} = Gl(K, m) \times Gl(K, p) \times Gl(K, q)$ in the following sense[1]: for any $(P, Q, R) \in \mathscr{G}$, the trilinear form $h(Ps, Qx, Ry)$ has the same length as that of $h(s, x, y)$. Of particular interest is the subgroup I of $\mathscr{G}$, called the *isotropy group*, consisting of those elements of $\mathscr{G}$ which satisfy $h(Ps, Qx, Ry) = h(s, x, y)$, and which could be used to generate many optimal algorithms out of a single optimal algorithm. We will see how to use this fact to establish good lower bounds (see also [7], [10]).

3. Lower bound techniques.

Howell and Lafon [10] have shown how linear independence arguments can be used to obtain a lower bound for the quaternion product in the commutative case. Van Leeuwen and van Emde Boas [15] have stated a general criterion for any set of bilinear forms; they applied this argument to several specific problems and obtained (relatively) good lower bounds. In this section, we combine the linear independence argument with other techniques, used in determining lower bounds for the rank of a tensor [2], to obtain more powerful lower bound techniques.

Let $B_i = x^T G_i y$, $1 \leqq i \leqq m$, be a set of bilinear forms over a commutative ring K. The commutative complexity of computing $\{B_i\}_{i=1}^{m}$ is the smallest integer μ such that

$$B_i = \sum_{l=1}^{\mu} \alpha_{il} f_l(x, y) f_l'(x, y), \qquad i = 1, 2, \cdots, m,$$

where $\alpha_{il} \in K$, $f_l(x, y)$ and $f_l'(x, y)$ are two linear forms in x and y, say $f_l(x, y) = \langle a_l, x\rangle + \langle b_l, y\rangle$ and $f_l'(x, y) = \langle \tilde{a}_l, x\rangle + \langle \tilde{b}_l, y\rangle$. Hence,

$$B_i = \sum_{l=1}^{\mu} \alpha_{il}\{\langle a_l, x\rangle + \langle b_l, y\rangle\}\{\langle \tilde{a}_l, x\rangle + \langle \tilde{b}_l, y\rangle\}, \qquad i = 1, 2, \cdots, m.$$

Expanding and substituting $x^T G_i y$ for B_i, we get

$$x^T G_i y = \sum_{l=1}^{\mu} \alpha_{il}\{x^T a_l \tilde{a}_l^T x + x^T(a_l \tilde{b}_l^T)y + x^T(\tilde{a}_l b_l^T)y + y^T(b_l \tilde{b}_l^T)y\}, \qquad i = 1, 2, \cdots, m,$$

[1] $Gl(K, n)$ denotes the general linear group of nonsingular $n \times n$ matrices over K.

i.e.,

$$(\dagger)\qquad \begin{aligned} &\sum_{l=1}^{\mu} \alpha_{il} a_l \tilde{a}_l^T = \sum_{l=1}^{\mu} \alpha_{il} b_l \tilde{b}_l^T = 0, \\ &G_i = \sum_{l=1}^{\mu} \alpha_{il} \{a_l \tilde{b}_l^T + \tilde{a}_l b_l^T\}, \qquad i = 1, 2, \cdots, m. \end{aligned}$$

We will use (†) to establish several of the results in this section.

We can also formulate the above problem in a matrix times a vector form [21] as follows:

$$\begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_m \end{bmatrix} = \begin{bmatrix} x^T G_1 \\ x^T G_2 \\ \vdots \\ x^T G_m \end{bmatrix} y = \check{G}(x) y.$$

Since the indeterminates commute, we equally have

$$B_i = \sum_{j=1}^{p} \sum_{k=1}^{q} r_{ijk} y_k x_j = \sum_{j=1}^{p} \left(\sum_{k=1}^{q} r_{ijk} y_k \right) x_j,$$

and the above multiplication problems can be also viewed as a product of the form $\hat{G}(y)x$. We now recall an important notion to get lower bounds [21]. Let $H(x)$ be a $p \times q$ matrix, whose entries are functions of the vector x over a field $\mathcal{F}$. The rows $r_1(x), \cdots, r_p(x)$ are said to be *linearly independent* if whenever $\sum_{i=1}^{p} \lambda_i r_i(x) \in \mathcal{F}^q$, $\lambda_i \in \mathcal{F}$, then $\lambda_i = 0\ \forall i$. Otherwise, the rows are called linearly dependent. Define the *row rank* of $H(x)$ to be the maximum number of linearly independent rows of $H(x)$. We can similarly define the *column rank* of $H(x)$. The following result is well known [21], [5].

LEMMA 3.1. *Let $\{B_i\}_{i=1}^m$ be a set of bilinear forms over a field $\mathcal{F}$, and let $\check{G}(x)$ be as defined above. Then the commutative complexity of computing $\{B_i\}_{i=1}^m$ satisfies*

$$\mu\{B_i\} \geqq \max\,(\text{column rank } (\check{G}(x)), \text{row rank } (\check{G}(x))).$$

The same is true for $\hat{G}(y)$.

Before establishing the next lower bound result, we give the following characterization of $\mu\{G(s)\}$ which has been shown in [9], [10].

THEOREM 3.2 [9], [10]. *Let $G(s)$ be the characteristic matrix associated with a given set of $p \times q$ bilinear forms $\{B_i = x^T G_i y\}_{i=1}^m$ over a ring K (of characteristic $\neq 2$). Let $N(s)$ be a $(p+q)\times(p+q)$ matrix whose entries are linear in s and such that*

$$N(s) + N(s)^T = \begin{bmatrix} 0 & G(s) \\ G^T_{(s)} & 0 \end{bmatrix},$$

where $\delta\{N(s)\}$ is minimal among all $(p+q)\times(p+q)$ matrices $N(s)$ which satisfy the above equation. Then $\delta\{N(s)\} = \mu\{G(s)\}$.

Proof. (a) Consider (†) again:

$$(\dagger)\qquad \begin{aligned} &\sum_{l=1}^{\mu} \alpha_{il} a_l \tilde{a}_l^T = \sum_{l=1}^{\mu} \alpha_{il} b_l \tilde{b}_l^T = 0, \\ &G_i = \sum_{l=1}^{\mu} \alpha_{il} \{a_l \tilde{b}_l^T + \tilde{a}_l b_l^T\}, \qquad i = 1, 2, \cdots, m. \end{aligned}$$

Note that

$$\begin{bmatrix} 0 & G_i \\ G_i^T & 0 \end{bmatrix} = \sum_{l=1}^{\mu} \alpha_{il} \begin{bmatrix} 0 & a_l \tilde{b}_l^T + \tilde{a}_l b_l^T \\ \tilde{b}_l a_l^T + b_l \tilde{a}_l^T & 0 \end{bmatrix}$$

$$= \sum_{l=1}^{\mu} \alpha_{il} \begin{bmatrix} a_l \\ b_l \end{bmatrix} [\tilde{a}_l^T \quad \tilde{b}_l^T] + \sum_{l=1}^{\mu} \alpha_{il} \begin{bmatrix} \tilde{a}_l \\ \tilde{b}_l \end{bmatrix} [a_l^T \quad b_l^T], \qquad i = 1, 2, \cdots, m.$$

Now, if we take $H(s) = \sum_{i=1}^{m} s_i H_i$, where

$$(**) \qquad H_i = \sum_{l=1}^{\mu} \alpha_{il} \begin{bmatrix} a_l \\ b_l \end{bmatrix} [\tilde{a}_l^T \quad \tilde{b}_l^T], \qquad i = 1, 2, \cdots, m,$$

then

$$H(s) + H(s)^T = \begin{bmatrix} 0 & G(s) \\ G_{(s)}^T & 0 \end{bmatrix}.$$

Equation $(**)$ implies that $\delta\{H(s)\} \leqq \mu\{G(s)\}$ and hence $\delta\{N(s)\} \leqq \mu\{G(s)\}$.

(b) We now prove the reverse inequality. Let $N(s)$ be any $(p+q) \times (p+q)$ characteristic matrix of minimal degree and which satisfies

$$N(s) + N(s)^T = \begin{bmatrix} 0 & G(s) \\ G(s)^T & 0 \end{bmatrix}.$$

It follows that

$$[x^T \quad y^T]\{N(s) + N(s)^T\} \begin{bmatrix} x \\ y \end{bmatrix} = [x^T \quad y^T] \begin{bmatrix} 0 & G(s) \\ G(s)^T & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

i.e.,

$$[x^T \quad y^T] N(s) \begin{bmatrix} x \\ y \end{bmatrix} = x^T G(s) y$$

if characteristic $(R) \neq 2$.

Consider any expansion for $N(s)$ into rank one matrices, say

$$N(s) = \sum_{l=1}^{\delta} \langle \alpha_l, s \rangle \begin{bmatrix} a_{l1} \\ a_{l2} \end{bmatrix} [b_{l1}^T \quad b_{l2}^T],$$

where a_{l1}, $h_{l1} \in K^p$ and a_{l2}, $h_{l2} \in K^q$. Thus,

$$x^T G(s) y = \sum_{l=1}^{\delta} \langle \alpha_l, s \rangle (x^T a_{l1} + y^T a_{l2})(b_{l1}^T x + b_{l2}^T y)$$

and hence

$$\mu\{G(s)\} \leqq \delta\{N(s)\}. \quad \square$$

We use the above characterization to establish a next lower bound result. The main idea is the same as that of substitution arguments (see, e.g., [1]) or partitioning techniques [2]. Suppose a set of bilinear forms is partitioned into two subsets, the first indexed by $\{s_i\}_{i=1}^m$, the second by $\{t_i\}_{i=1}^r$. Let the corresponding characteristic matrix be given by $G(s) + H(t)$. Without loss of generality, we assume the H_i's to be linearly independent.

THEOREM 3.3. *Let $G(s)+H(t)$ be as given above. Then, over any ring K, we have*

$$\mu\{G(s)+H(t)\} \geqq \dim t + \min_R \mu\{G(s)+H(Rs)\},$$

where R is a $(\dim t)\times(\dim s')$ *matrix over K.*

Proof. From Theorem 3.2, we have $\mu\{G(s)+H(t)\}=\delta\{N_1(s)+N_2(t)\}$, where

$$\{N_1(s)+N_2(t)\}+\{N_1(s)+N_2(t)\}^T=\begin{bmatrix} 0 & G(s)+H(t) \\ G(s)^T+H(t)^T & 0 \end{bmatrix},$$

and $\delta\{N_1(s)+N_2(t)\}$ is minimal. Using the partitioning technique [2, Thm. 10], we obtain

$$\delta\{N_1(s)+N_2(t)\} \geqq \dim t + \min_R \delta\{N_1(s)+N_2(Rs)\},$$

and thus

$$\mu\{G(s)+H(t)\} \geqq \dim t + \min_R \mu\{G(s)+H(Rs)\}. \quad \square$$

The main lower bound result stated in [15] is an immediate corollary of Theorem 3.3 and Lemma 3.1.

We now establish a lower bound theorem which, for a small number of bilinear forms, gives stronger lower bounds than those of the previous techniques.

THEOREM 3.4. *Let $G(s)$ be the characteristic matrix of a set of bilinear forms $\{B_i\}_{i=1}^m$ over a ring K. Then*

$$\mu\{G(s)\} \geqq \frac{1}{2}\delta\begin{bmatrix} G(s) & 0 \\ 0 & G(s)^T \end{bmatrix}.$$

Proof. We again consider (†):

$$(\dagger)\qquad \begin{aligned} &\sum_{l=1}^{\mu} \alpha_{il}a_l\tilde{a}_l^T = \sum_{l=1}^{\mu} \alpha_{il}b_l\tilde{b}_l^T = 0, \\ &G_i = \sum_{l=1}^{\mu} \alpha_{il}\{a_l\tilde{b}_l^T + \tilde{a}_l b_l^T\}, \qquad i=1,2,\cdots,m. \end{aligned}$$

Notice that

$$\begin{aligned} \begin{bmatrix} G_i & 0 \\ 0 & G_i^T \end{bmatrix} &= \sum_{l=1}^{\mu} \alpha_{il}\begin{bmatrix} a_l\tilde{b}_l^T+\tilde{a}_l b_l^T & 0 \\ 0 & \tilde{b}_l a_l^T + b_l\tilde{a}_l^T \end{bmatrix} \\ &= \sum_{l=1}^{\mu} \alpha_{il}\begin{bmatrix} a_l \\ b_l \end{bmatrix}\begin{bmatrix} \tilde{b}_l^T & \tilde{a}_l^T \end{bmatrix} + \sum_{l=1}^{\mu} \alpha_{il}\begin{bmatrix} \tilde{a}_l \\ \tilde{b}_l \end{bmatrix}\begin{bmatrix} b_l^T & a_l^T \end{bmatrix}, \qquad i=1,2,\cdots,m. \end{aligned}$$

It follows that 2μ rank one matrices include the $\begin{bmatrix} G_i & 0 \\ 0 & G_i^T \end{bmatrix}$ in their span. Therefore,

$$\delta\begin{bmatrix} G(s) & 0 \\ 0 & G(s)^T \end{bmatrix} \leqq 2\mu. \quad \square$$

COROLLARY. *Suppose we partition a set of bilinear forms into two subsets indexed respectively by s and t. Then*

$$\mu\{G(s)+H(t)\} \geqq \dim t + \frac{1}{2}\min_R \delta\begin{bmatrix} G(s)+H(Rs) & 0 \\ 0 & G(s)^T+H(Rs)^T \end{bmatrix},$$

where R is a $(\dim t)\times(\dim s)$ *matrix over K.*

It is well known that, for any characteristic matrix $G(s)$, we have $\delta\{G(s)\} \leqq 2\mu\{G(s)\}$ [21]. An open question is whether there exist bilinear forms for which commutativity will reduce the number of multiplications by a precise factor of $\frac{1}{2}$. Using the above theorem, we will prove that this can never happen.

THEOREM 3.5. *Let $G(s)$ be any characteristic matrix whose row and column ranks are r and c respectively. Then, we have*

$$\mu\{G(s)\} \geqq \tfrac{1}{2}\{\delta\{G(s)\} + \max(r, c)\} > \tfrac{1}{2}\delta\{G(s)\}.$$

Proof. We know that

$$\delta\begin{bmatrix} G(s) & 0 \\ 0 & G^T(s) \end{bmatrix} \geqq \delta\{G(s)\} + \text{column rank}\,\{G^T(s)\} = \delta\{G(s)\} + r.$$

In [12], we proved that

$$\delta\begin{bmatrix} G(s) & 0 \\ 0 & G^T(s) \end{bmatrix} = \delta\begin{bmatrix} G^T(s) & 0 \\ 0 & G(s) \end{bmatrix},$$

and hence

$$\delta\begin{bmatrix} G(s) & 0 \\ 0 & G^T(s) \end{bmatrix} \geqq \delta\{G^T(s)\} + \text{column rank}\,\{G(s)\} = \delta\{G(s)\} + c.$$

Using Theorem 3.4, we get

$$\mu\{G(s)\} \geqq \tfrac{1}{2}\{\delta\{G(s)\} + \max(r, c)\} > \tfrac{1}{2}\delta\{G(s)\}. \quad \square$$

COROLLARY. *There exist no sets of bilinear forms for which commutativity reduces the optimal number of multiplications by $\frac{1}{2}$.*

Note that Theorem 3.5 gives

$$\mu\{M_{nnn}^{(s)}\} \geqq \tfrac{1}{2}\{\delta\{M_{nnn}^{(s)}\} + n^2\},$$

where $M_{nnn}^{(s)}$ is the characteristic matrix of the (n, n, n) matrix multiplication problem.

We will now give an example to show that our lower bound techniques will produce stronger lower bounds than any of the previously known techniques.

Let $n = 2k$, $k \in Z^+$. Consider the computation of the following pair of bilinear forms,

$$B_1 = x_1y_1 + x_2y_2 + \cdots + x_ny_n,$$

$$B_2 = x_1y_2 + x_3y_4 + \cdots + x_{n-1}y_n.$$

This can be viewed as the following multiplication problem:

$$\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & & x_{n-1} & x_n \\ 0 & x_1 & 0 & x_3 & 0 \ \cdots & 0 & x_{n-1} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Applying the techniques of [15], it is not possible to obtain anything better than $n+1$ as a lower bound. However, using our techniques, we will see how to obtain $3n/2$ as a lower bound.

THEOREM 3.6. *The complexity of computing the two bilinear forms*

$$B_1 = x_1y_1 + x_2y_2 + \cdots + x_ny_n,$$

$$B_2 = x_1y_2 + x_3y_4 + \cdots + x_{n-1}y_n,$$

is precisely $\lfloor 3n/2 \rfloor$, *even if we use commutativity.*

Proof. We will illustrate the case $n = 4$ and the same proof carries for any n. Note that the characteristic matrix is given by

$$G(s) = \begin{bmatrix} s_1 & s_2 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_1 & s_2 \\ 0 & 0 & 0 & s_1 \end{bmatrix}.$$

From Theorem 3.4, we have

$$\mu\{G(s)\} \geqq \frac{1}{2}\delta\begin{bmatrix} G(s) & 0 \\ 0 & G^T(s) \end{bmatrix} = \frac{1}{2}\delta\{H(s)\}.$$

Rearranging columns, we get

$$H(s) \equiv \begin{bmatrix} s_2 & 0 & 0 & 0 & s_1 & 0 & 0 & 0 \\ s_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 & 0 & s_1 & 0 & 0 \\ 0 & s_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & s_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & s_2 & 0 & 0 & 0 & s_1 & 0 \\ 0 & 0 & 0 & s_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & s_2 & 0 & 0 & 0 & s_1 \end{bmatrix}.$$

Applying Theorem 10 of [2] to the last four columns, we get

$$\delta\{H(s)\} \geqq 4 + \delta \begin{bmatrix} s_2 + \alpha_1 s_1 & \alpha_2 s_1 & \alpha_3 s_1 & \alpha_4 s_1 \\ s_1 & 0 & 0 & 0 \\ \beta_1 s_1 & s_2 + \beta_2 s_1 & \beta_3 s_1 & \beta_4 s_1 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_1 & 0 \\ r_1 s_1 & r_2 s_1 & s_2 + r_3 s_1 & r_4 s_1 \\ 0 & 0 & 0 & s_1 \\ \tau_1 s_1 & \tau_2 s_1 & \tau_3 s_1 & s_2 + \tau_4 s_1 \end{bmatrix},$$

where $\alpha_i, \beta_i, r_i, \tau_i \in K$.

Using proper row operations, we obtain

$$\delta\{H(s)\} \geqq 4 + \delta \begin{bmatrix} s_2 & 0 & 0 & 0 \\ s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & s_1 & 0 & 0 \\ 0 & 0 & s_1 & 0 \\ 0 & 0 & s_2 & 0 \\ 0 & 0 & 0 & s_1 \\ 0 & 0 & 0 & s_2 \end{bmatrix} = 12.$$

Therefore, $\mu\{G(s)\} \geqq \frac{1}{2}\delta\{H(s)\} \geqq 6$. It follows that $\mu\{G(s)\} = 6$. □

4. Nonvalidity of duality theory for matrix multiplication problems. As we have mentioned in the Introduction, each set of bilinear forms is equivalent to five other sets of bilinear forms obtained by permuting the indices of the corresponding trilinear form. This property, which is known as duality in the literature, has been used to obtain new algorithms (e.g., [8]) and to establish lower bounds. Brockett and Dobkin [2] have exploited this property to get relatively good lower bounds for matrix multiplication in spite of the fact that their techniques are essentially based on linear independence and substitution arguments. In this section, we will establish the fact that duality theory does *not* hold if the indeterminates commute. We consider, for this purpose, the $(n, 2, 2)$ matrix multiplication problems, i.e., the problems of multiplying $n \times 2$ by 2×2, 2×2 by $2 \times n$ and $2 \times n$ by $n \times 2$ matrices. We know that all of these problems are equivalent, in the noncommutative case, and that each of the optimal algorithms requires precisely $\lceil 7n/2 \rceil$ multiplications [7]. It is also well known [20] that it is possible to multiply $n \times 2$ by 2×2 matrices (or 2×2 by $2 \times n$) with $3n + 2$ multiplications in the commutative case with only integer constants. Waksman [19] reduced the number of multiplications by 1, although his algorithm uses the constant $\frac{1}{2}$. Surprisingly enough, none of these algorithms generates a fast commutative algorithm for the multiplication of $2 \times n$ by $n \times 2$ matrices. We will prove that this problem requires at least $\lceil 27n/8 \rceil$ multiplications whenever we use integer constants; this shows that this problem is harder than any of its duals. In this section we describe Winograd's algorithm and the modification introduced by Waksman, and we establish our lower bound for the $2 \times n$ by $n \times 2$ matrix multiplication problem.

Let X and Y be the following two matrices:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \qquad Y = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \end{bmatrix}.$$

Winograd's algorithm to compute XY goes as follows. Compute

$$\begin{aligned} &\zeta_1 = x_{11}x_{12}, \qquad \zeta_2 = x_{21}x_{22}, \\ &\eta_i = y_{1i}y_{2i}, \\ &(x_{11} + y_{2i})(x_{12} + y_{1i}), \\ &(x_{21} + y_{2i})(x_{22} + y_{1i}), \qquad i = 1, 2, \cdots, n. \end{aligned}$$

Then we have

$$\begin{aligned} &x_{11}y_{1i} + x_{12}y_{2i} = (x_{11} + y_{2i})(x_{12} + y_{1i}) - \zeta_1 - \eta_i, \\ &x_{21}y_{1i} + x_{22}y_{2i} = (x_{21} + y_{2i})(x_{22} + y_{1i}) - \zeta_2 - \eta_i, \qquad 1 \leqq i \leqq n. \end{aligned} \tag{††}$$

It is obvious that the above algorithm requires $3n + 2$ multiplications.

We can reduce the number of multiplications by one as follows. Compute

$$\begin{aligned} &(x_{11} + y_{2i})(x_{12} + y_{1i}), \\ &(x_{21} + y_{2i})(x_{22} + y_{1i}), \\ &(x_{11} - y_{2i})(x_{12} - y_{1i}), \qquad 1 \leqq i \leqq n; \\ &(x_{21} - y_{21})(x_{22} - y_{11}). \end{aligned}$$

Note the following:

$$\begin{aligned} &\zeta_1 + \eta_i = \tfrac{1}{2}\{(x_{11} + y_{2i})(x_{12} + y_{1i}) + (x_{11} - y_{2i})(x_{12} - y_{1i})\}, \qquad 1 \leqq i \leqq n, \\ &\zeta_2 + \eta_1 = \tfrac{1}{2}\{(x_{21} + y_{21})(x_{22} + y_{11}) + (x_{21} - y_{21})(x_{22} - y_{11})\}. \end{aligned}$$

Now we can compute the remaining $\zeta_2+\eta_i$ by the equation

$$\zeta_2+\eta_i=(\zeta_2+\eta_1)+(\zeta_1+\eta_i)-(\zeta_1+\eta_1), \qquad i=2,\cdots,n.$$

Using the same equations (††) as before, the above algorithm requires $3n+1$ multiplications.

We prove that the above algorithms are essentially optimal up to 2 or 1 multiplication.

THEOREM 4.1. *The problem of multiplying* 2×2 *by* $2\times n$ *matrices requires at least* $3n$ *multiplications in the commutative case, even if the constants come from a field.*

Proof. It is easy to check that (one form of) the corresponding characteristic matrix is given by

$$G(s)=\begin{bmatrix} s_1s_2 & \cdots & s_n & & & 0\\ s_{n+1} & \cdots & s_{2n} & & & \\ & & & s_1s_2 & \cdots & s_n\\ 0 & & & s_{n+1} & \cdots & s_{2n}\end{bmatrix}.$$

Applying Theorem 3.3 with $t=(s_{n+1},\cdots,s_{2n})$, we get

$$\mu\{G(s)\}\geqq n+\mu\left\{\begin{bmatrix} s_1s_2 & \cdots & s_n & & & 0\\ \sim\sim & \cdots & \sim & & & \\ & & & s_1s_2 & \cdots & s_n\\ 0 & & & \sim\sim & \cdots & \sim\end{bmatrix}\right\},$$

where a $\sim$ means a linear combination of $s_1, s_2, \cdots, s_n$.

Since

$$\text{column rank}\left(\begin{bmatrix} s_1s_2 & \cdots & s_n & & & \\ \sim\sim & \cdots & \sim & & & \\ & & & s_1s_2 & \cdots & s_n\\ & & & \sim\sim & \cdots & s_n\end{bmatrix}\right)=2n,$$

we obtain $\mu\{G(s)\}\geqq 3n$. □

Note that the same lower and upper bounds hold for the multiplication problem of $n\times 2$ by 2×2 matrices.

We now consider the problem of multiplying $2\times n$ by $n\times 2$ matrices. Before establishing a lower bound we need a couple of results. Let $G(s)$ be the characteristic matrix of the corresponding bilinear problem. It is straightforward to check that (one form of) $G(s)$ is given by

$$G(s)=\begin{bmatrix} \begin{matrix} s_1 & s_2\\ s_3 & s_4\end{matrix} & 0 & & \\ 0 & \begin{matrix} s_1 & s_2\\ s_3 & s_4\end{matrix} & & \\ & & \ddots & 0\\ & & 0 & \begin{matrix} s_1 & s_2\\ s_3 & s_4\end{matrix}\end{bmatrix} \quad (n \text{ blocks}).$$

Recall that $(A, B, C) \in Gl(2n) \times Gl(4) \times Gl(2n)$ is an element of the isotropy group I_G of $G(s)$ if $AG(Bs)C = G(s)$. We need the following theorem from [2].

THEOREM 4.2 [2]. *$(A, B, C) \in I_G$ if and only if there exist nonsingular matrices P, Q and R of dimensions n, 2 and 2 respectively, such that $A = P \otimes Q$, $C = P \otimes R$ and $B = (Q \otimes R^T)^{-1}$.*

THEOREM 4.3. *Let $(A, B, C) \in I_G$ and let $H(s)$ be the characteristic matrix*

$$H(s) = \begin{bmatrix} G(s) & 0 \\ 0 & G(s)^T \end{bmatrix}.$$

Let A', B' and C' be given by

$$A' = \begin{pmatrix} A & 0 \\ 0 & C^T \end{pmatrix}, \qquad B' = B \qquad \textit{and} \qquad C' = \begin{pmatrix} C & 0 \\ 0 & A^T \end{pmatrix}.$$

Then $(A', B', C') \in I_H$.

Proof. We have $AG(Bs)C = G(s)$. Hence, $C^T G^T(Bs) A^T = G^T(s)$. On the other hand,

$$\begin{aligned} A'H(B's)C' &= \begin{pmatrix} A & 0 \\ 0 & C^T \end{pmatrix} \begin{pmatrix} G(Bs) & \\ & G^T(Bs) \end{pmatrix} \begin{pmatrix} C & 0 \\ 0 & A^T \end{pmatrix} \\ &= \begin{pmatrix} AG(Bs)C & 0 \\ 0 & C^T G^T(Bs) A^T \end{pmatrix} = H(s). \end{aligned}$$

Therefore, $(A', B', C') \in I_H$. □

We need also to establish the following two facts whose proofs are immediate from [12].

LEMMA 4.4. *Over any field, we have*

$$\delta \begin{bmatrix} \begin{array}{cc|c} s_1 & s_2 & \\ s_2 & 0 & \\ \hline & & \ddots \end{array} & & 0 \\ & & \\ 0 & & \begin{array}{c|cc} \ddots & & \\ \hline & s_1 & s_2 \\ & s_2 & 0 \end{array} \end{bmatrix} = 3k,$$

where k is the number of blocks, and

$$\delta \begin{bmatrix} s_1 & 0 & & & & & & & & \\ s_2 & s_1 & & & & & & & & \\ & & \ddots & & & & & & & \\ & & & s_1 & 0 & & & & & \\ & & & s_2 & s_1 & & & & & \\ & & & & & s_1 & s_2 & & & \\ & & & & & 0 & s_1 & & & \\ & & & & & & & \ddots & & \\ & & & & & & & & s_1 & s_2 \\ & & & & & & & & 0 & s_1 \end{bmatrix} = 3k,$$

where again k is the number of blocks.

We are ready for the following theorem.

THEOREM 4.5. *Any commutative bilinear algorithm which computes the matrix product of* $2\times n$ *by* $n\times 2$ *requires at least* $\lceil 27n/8\rceil$ *multiplications over* Z_2 *or* Z.

Proof. The techniques we will use are similar to those introduced in [13]. We prove that

$$\delta_{Z_2}\{H(s)\}=\delta_{Z_2}\begin{bmatrix} G(s) & 0 \\ 0 & G^T(s)\end{bmatrix}\geqq\frac{27n}{4},$$

and, using Theorem 3.4, the result would follow.

Since $s\in Z_2^4$, all linear forms over Z_2 are of the form $\sum_{i=1}^4 \sigma_i s_i$, where $\sigma_i\in Z_2$ $(i=1,\cdots,4)$. Thus, any decomposition of $H(s)$ into rank one matrices

$$H(s)=\sum_{l=1}^{\tau}\langle\alpha_l, s\rangle D_l$$

can be partitioned into different subsums, each subsum corresponding to one linear form of s over Z_2. Let $\gamma(\sigma_1,\sigma_2,\sigma_3,\sigma_4)$ be the number of rank one matrices associated with the linear form $\sum_{i=1}^4\sigma_i s_i$, $\gamma(0,0,0,0)=0$. Note that $\tau=\sum_{\sigma_1,\cdots,\sigma_4=0}^{1}\gamma(\sigma_1,\sigma_2,\sigma_3,\sigma_4)$. Set $s_4=0$ and $s_2=s_3$. Then at least $\gamma(0,0,0,1)+\gamma(0,1,1,0)+\gamma(0,1,1,1)$ terms vanish. Using Lemma 4.4, the resulting problem requires $6n$ multiplications; thus

(1) $$\tau\geqq 6n+\gamma(0,0,0,1)+\gamma(0,1,1,0)+\gamma(0,1,1,1).$$

Similarly, setting $s_2=0$ and $s_1=s_4$, we get

(2) $$\tau\geqq 6n+\gamma(0,1,0,0)+\gamma(1,0,0,1)+\gamma(1,1,0,1).$$

On the other hand, if we set $s_1=0$ and then $s_3=0$, we get the following

(3) $$\tau\geqq 6n+\gamma(1,0,0,0),$$

(4) $$\tau\geqq 6n+\gamma(0,0,1,0).$$

From Theorems 4.2 and 4.3 it follows that there exists $(P,B,Q)\in I_H$ such that $B=\begin{pmatrix}1&0\\1&1\end{pmatrix}\otimes\begin{pmatrix}1&1\\0&1\end{pmatrix}$, i.e.,

$$B=\begin{pmatrix}1&1&0&0\\0&1&0&0\\1&1&1&1\\0&1&0&1\end{pmatrix}.$$

Hence

$$Bs=\begin{pmatrix}s_1+s_2\\ s_2\\ s_1+s_2+s_3+s_4\\ s_2+s_4\end{pmatrix}=\begin{pmatrix}s_1'\\ s_2'\\ s_3'\\ s_4'\end{pmatrix}.$$

This transformation generates a new algorithm given by

$$H(s)=PH(Bs)Q=\sum_{l=1}^{\tau}\langle\alpha_l', s'\rangle PD_lQ.$$

Note that if $r(s)$ is a linear form in the original algorithm with α rank one matrices, $r(Bs)$ has also α (different) rank one matrices in the new algorithm.

Setting $s_2=s_3$ and $s_4=0$ in the new algorithm, we get

$$\tau \geqq 6n+\gamma(0,1,0,1)+\gamma(1,0,1,1)+\gamma(1,1,1,0). \tag{5}$$

Setting $s_3=0$ and then $s_1=0$, we get

$$\tau \geqq 6n+\gamma(1,1,1,1), \tag{6}$$

$$\tau \geqq 6n+\gamma(1,1,0,0). \tag{7}$$

Similarly, we have

$$\tau \geqq 6n+\gamma(1,0,1,1), \tag{8}$$

$$\tau \geqq 6n+\gamma(0,0,1,1). \tag{9}$$

Add up (1)–(9) to get

$$9\tau \geqq 54n+\tau, \quad \text{that is,} \quad 8\tau \geqq 54n.$$

Thus $\tau \geqq \lceil 27n/4 \rceil$. Therefore, $\mu\{G(s)\} \geqq \frac{1}{2}\delta\{H(s)\} \geqq 27n/8$. □

5. The commutative complexity of pairs of bilinear forms. The complexity of computing a pair of bilinear forms has been determined in [12], in the case where the indeterminates do not commute. We prove that the complexity of computing a pair of bilinear forms remains the same whether or not commutativity is assumed. In order to establish this fact, we will recall some basic facts shown in [12].

Let $G(s)$ be the characteristic matrix corresponding to a pair of bilinear forms over a field $\mathcal{F}$. Then $G(s)$ is equivalent to a canonical characteristic matrix of the following form

$$\begin{bmatrix} L_{\varepsilon_1}(s) & & & & & & & & \\ & \ddots & & & & & & & \\ & & L_{\varepsilon_r}(s) & & & & & & \\ & & & L^T_{\eta_1}(s) & & & & & \\ & & & & \ddots & & & & \\ & & & & & L^T_{\eta_k}(s) & & & \\ & & & & & & C_1(s) & & \\ & & & & & & & \ddots & \\ & & & & & & & & C_t(s) \end{bmatrix},$$

$$L_\varepsilon(s)=\underbrace{\begin{bmatrix} s_2 & s_1 & & & \\ & s_2 & s_1 & & \\ & & \ddots & \ddots & \\ & & & s_2 & s_1 \end{bmatrix}}_{\varepsilon+1}\left.\vphantom{\begin{bmatrix} s_2\\ s_2\\ \ddots\\ s_2\end{bmatrix}}\right\}\varepsilon$$

and

$$C_i(s)=\begin{bmatrix} s_1 & s_2 & & & \\ & s_1 & s_2 & & \\ & & \ddots & \ddots & \\ & & & s_1 & s_2 \\ \alpha_{i0}s_2 & \cdots & & \alpha_{in_i-2}s_2 & s_1+\alpha_{in_i-1}s_2 \end{bmatrix}, \qquad 1 \leqq i \leqq t,$$

where $0<\varepsilon_1\leqq\varepsilon_2\leqq\cdots\leqq\varepsilon_r$, $0<\eta_1\leqq\eta_2\leqq\cdots\leqq\eta_k$ are called the *minimal indices* $(r, k\geqq 0)$, and such that the polynomials

$$p_i(x)=\alpha_{i0}-\alpha_{i1}x+\cdots+(-1)^{n_i-1}\alpha_{n_i-1}x^{n_i-1}+x^{n_i}$$

are the nontrivial invariant polynomials of the corresponding pairs of matrices [12].

THEOREM 5.1 [12]. *Let $\{\varepsilon_i\}_{i=1}^r$, $\{\eta_j\}_{j=1}^k$, $\{p_q(x)\}_{q=1}^t$ be as defined above for a pair of bilinear forms $\{B_1, B_2\}$ and let $\mathcal{F}$ be a field which contains the roots of $p_1(x)$ and* Card $\mathcal{F}\geqq\max_{i,j,r}\{\varepsilon_i, \eta_j, \deg p_q(x)\}$. *Then*

$$\delta_{\mathcal{F}}\{B_1, B_2\}=\sum_{i=1}^r\varepsilon_i+\sum_{j=1}^k\eta_j+r+k+l,$$

where l is the number of $p_r(x)$'s which do not factor into distinct linear factors over $\mathcal{F}$.

The following theorem is the main result of this section.

THEOREM 5.2. *Let $G(s)$ be the characteristic matrix corresponding to a pair of bilinear forms over a field $\mathcal{F}$ which contains the roots of a $p_1(x)$ (as defined above) and such that* Card $\mathcal{F}$ *is large enough. Then $\delta\{G(s)\}=\mu\{G(s)\}$.*

Proof. Let $H(s)$ be defined by

$$H(s)=\begin{bmatrix}G(s) & 0\\ 0 & G(s)^T\end{bmatrix}.$$

Note that $H(s)$ corresponds to a pair of bilinear forms such that its minimal indices are the union of the minimal indices of $G(s)$ and those of $G(s)^T$. Note also that the invariant polynomials of $G(s)$ are the same as those of $G^T(s)$. Therefore, the invariant polynomials of $H(s)$ are the same as those of $G(s)$ repeated twice. Theorem 5.1 implies

$$\delta\{H(s)\}=2\sum_{i=1}^r\varepsilon_i+2\sum_{j=1}^k\eta_j+2r+2k+2l,$$

i.e., $\delta\{H(s)\}=2\delta\{G(s)\}$.

Theorem 3.4 implies that $\mu\{G(s)\}\geqq\frac{1}{2}\delta\{H(s)\}=\delta\{G(s)\}$ and, therefore, $\mu\{G(s)\}=\delta\{G(s)\}$. □

The lower bounds for the case when the field has a small cardinality which have been stated in [13] hold also for the commutative case. We consider one example.

Let $\mathcal{F}=Z_2$. Consider the following characteristic matrix, which corresponds to the problem of multiplying a linear polynomial by an $(n-1)$-degree polynomial,

$$L_n(s)=\left.\underbrace{\begin{bmatrix}s_1 & s_2 & & & \\ & s_1 & s_2 & & \\ & & \ddots & \ddots & \\ & & & s_1 & s_2\end{bmatrix}}_{n+1}\right\}n.$$

In [13], we have proved that $\delta_{Z_2}\{L_n(s)\}=n+\lceil n/2\rceil$. We now prove a similar result for $\mu_{Z_2}\{L_n(s)\}$. From Theorem 3.4, we know that

$$\mu_{Z_2}\{L_n(s)\}\geqq\frac{1}{2}\delta_{Z_2}\begin{bmatrix}L_n(s) & 0\\ 0 & L_n^T(s)\end{bmatrix}.$$

We use an argument similar to that used in the proof of Theorem 4.5.

Consider any algorithm for $\begin{bmatrix} L_n(s) & 0 \\ 0 & L_n^T(s) \end{bmatrix}$,

$$H_n(s)=\begin{bmatrix} L_n(s) & 0 \\ 0 & L_n^T(s) \end{bmatrix}=\sum_{l=1}^{\tau} \langle \alpha_l, s\rangle D_l.$$

Since the only linear forms over Z_2 are s_1, s_2 and s_1+s_2, the above decomposition can be partitioned as follows:

$$H_n(s)=\sum_{l=1}^{\alpha_1} s_1 D_l^{(1)}+\sum_{l=1}^{\alpha_2} s_2 D_l^{(2)}+\sum_{l=1}^{\alpha_3} (s_1+s_2) D_l^{(3)}.$$

Hence, $\tau=\alpha_1+\alpha_2+\alpha_3$.

Set $s_1=0$; α_1 terms disappear. However, the resulting problem requires at least $2n$ multiplications. Thus

$$\tau \geqq 2n+\alpha_1. \tag{10}$$

Similarly

$$\tau \geqq 2n+\alpha_2, \qquad (s_2=0), \tag{11}$$

$$\tau \geqq 2n+\beta, \qquad (s_1=s_2). \tag{12}$$

Summing up (10), (11) and (12), we get

$$3\tau \geqq 6n+\tau,$$

i.e., $\tau \geqq 3n$.

It follows that $\mu\{L_n(s)\} \geqq \frac{1}{2}\delta_{Z_2}\{H_n(s)\} \geqq 3n/2$ and, therefore, $\mu_{Z_2}\{L_n(s)\}=n+\lceil n/2\rceil=\delta_{Z_2}\{L_n(s)\}$.

6. Acknowledgment. The author wishes to thank the referees for their careful reading of the original manuscript and for their very constructive comments.

REFERENCES

[1] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] 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.

[3] ———, *On the number of multiplications required for matrix multiplication*, this Journal, 5 (1976), pp. 624–628.

[4] H. F. deGroote, *On varieties of optimal algorithms for the computation of bilinear mappings. I. The isotropy group of a bilinear mapping*, Theoret. Comput. Sci., 7 (1978), pp. 1–24.

[5] C. M. Fiduccia, *Fast matrix multiplication*, Proceedings of the 3rd Annual Symposium on Theory of Computing, Shaker Heights, Ohio, 1977, pp. 45–49.

[6] ———, *On obtaining upper bounds on the complexity of matrix multiplication*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, New York, 1972.

[7] J. E. Hopcroft and L. Kerr, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[8] J. E. Hopcroft and J. Musinski, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, this Journal, 2 (1973), pp. 159–173.

[9] T. D. Howell, *Tensor rank and the complexity of bilinear forms*, Ph.D. thesis, Cornell University, Ithaca, NY, 1976.

[10] T. D. Howell and J. C. Lafon, *The complexity of the quaternion product*, TR 75-245, Dept. of Computer Science, Cornell University, Ithaca, NY, 1975.

[11] L. HYAFIL, *The power of commutativity*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, 1977.

[12] J. JA'JA', *Optimal evaluation of pairs of bilinear forms*, this Journal, 8 (1979), pp. 443–462.

[13] ———, *Computation of bilinear forms over finite fields*, Tech. Rep. CS-78-03, Dept. of Computer Science, Pennsylvania State University, 1978.

[14] J. C. LAFON, *Optimum computation of p bilinear forms*, Linear Algebra and Appl., 10 (1975), pp. 225–260.

[15] J. VAN LEEUWEN AND P. VAN EMDE BOAS, *Some elementary proofs of lower bounds in complexity theory*, Ibid., 19 (1978), pp. 63–80.

[16] R. L. PROBERT, *On the complexity of symmetric computations*, Canad. J. Information Processing and Operational Res., 12 (1974), pp. 71–86.

[17] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

[18] ———, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[19] A. WAKSMAN, *On Winograd's algorithm for inner products*, IEEE Trans. Computers, 19 (1970), pp. 360-361.

[20] S. WINOGRAD, *A new algorithm for inner products*, Ibid., 17 (1968), pp. 693–694.

[21] ———, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

[22] ———, *On the multiplication of* 2 *by* 2 *matrices*, Linear Algebra and Appl., 4 (1971), pp. 381–388.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0008 $01.00/0

EQUALITY SETS AND COMPLEXITY CLASSES*

RONALD V. BOOK† AND FRANZ-JOSEF BRANDENBURG‡

Abstract. If h_1, h_2 are two homomorphisms, then the *equality set* Eq (h_1, h_2) *of* h_1, h_2 is Eq $(h_1, h_2) = \{w | h_1(w) = h_2(w)\}$. In this paper it is shown how to characterize complexity classes of formal languages in terms of equality sets of pairs of homomorphisms with bounded balance. In addition the *complete twin shuffle* language is investigated, and it is shown that for alphabets with at least two letters, this language cannot be represented as the equality set of a pair of homomorphisms unless both homomorphisms are erasing and have linear bounded balance.

Key words. equality set, time bounds, space bounds, complexity classes, complete twin shuffle language, balance of homomorphisms

Introduction. Characterizations and representations of classes of languages play an important role in formal language theory. For example, if a class of languages can be shown to be a basis for the class of recursively enumerable sets, then one knows immediately that certain questions about the languages in that class are undecidable. Also, characterization theorems are useful in showing that certain generators or hardest sets exist. Recently there have been a number of new results regarding the decidability or undecidability of certain combinatorial questions about formal languages, grammars, L-systems, homomorphisms, and other types of mappings (e.g., [6]). These results have motivated several studies that resulted in three related characterizations of the class of recursively enumerable sets [3], [7], [11].

Let h_1, h_2 be homomorphisms with a common domain Σ^*. Define the *equality set* Eq (h_1, h_2) *of* h_1, h_2 as Eq $(h_1, h_2) = \{w | h_1(w) = h_2(w)\}$. A DGSM *mapping* $g: \Sigma^* \to \Delta^*$ is a function computed by a deterministic generalized sequential machine with accepting states. For any function $f: \Sigma^* \to \Sigma^*$, the *fixed-point language* Fp (f) *of the function* f is defined to be Fp $(f) = \{w \in \Sigma^* | f(w) = w\}$. For any language L, let MIN $(L) = \{x \in L | \text{there are no nonempty } y, z \text{ such that } yz = x \text{ and } y \in L\}$.

PROPOSITION. *Let L be a language. The following are equivalent*:

(i) *L is recursively enumerable*;

(ii) *there exist two homomorphisms* h_1, h_2 *and a* DGSM *mapping* g *such that* $g(\text{Eq}(h_1, h_2)) = L$ [11];

(iii) *there exists a* DGSM *mapping g and a homomorphism h such that* $h(\text{Fp}(g)) = L$ [7];

(iv) *there exist homomorphisms* h_0, h_1, h_2 *such that* $h_0(\text{MIN}(\text{Eq}(h_1, h_2))) = L$ [3].

In each case the proof proceeds by showing that the set of accepting computations of a Turing machine or the set of proper derivations of a phrase structure grammar can be encoded in the appropriate way. The fact that certain questions are undecidable follows immediately from the fact that the system under study forms a basis or a sub-basis for the class of all recursively enumerable sets.

In this paper we approach equality sets as part of the study of complexity classes of formal languages and exploit the equality mechanism in its simplest form. Using the

* Received by the editors March 1, 1979, and in revised form November 21, 1979. This research was supported in part by the National Science Foundation under Grant MCS77-11360. Some of these results were presented at the Sixth International Colloquium on Automata, Languages, and Programming, July 1979, Graz, Austria. An extended abstract appears in the proceedings of that symposium.

† Department of Mathematics, University of California, Santa Barbara, California 93106.

‡ Institut für Informatik, Universität Bonn, Wegelerstrasse 6, 5300 Bonn, Federal Republic of Germany. The work of this author was supported in part by the German Academic Exchange Service under Grant No. 430/402/777/9.

basic encoding strategy employed by Salomaa [11], we pay strict attention to the amount of time and space used in the accepting computations of a Turing machine, and we show that many classes such as NP, the Grzegorcyzk classes, and PSPACE can be represented in terms of equality sets of pairs of homomorphisms. (The representation theorem for machines is given as Theorem 2.1 and the characterizations of classes are developed in § 3.)

Culik [3] and Engelfriet and Rozenberg [7] have studied a specific equality set with very interesting properties. Let 0, 1, $\bar{0}$, $\bar{1}$ be four symbols. Let g be the homomorphism determined by defining $g(0)=\bar{0}$ and $g(1)=\bar{1}$. Let $L_{\{0,1\}}=\{x_1y_1\cdots x_ny_n\,|\,x=x_1\cdots x_n,$ $g(x)=y_1\cdots y_n,\ x\in\{0,1\}^*\}$. The language $L_{\{0,1\}}$ is called the *complete twin shuffle* language. For homomorphisms h_1, h_2 determined by defining $h_1(a)=h_2(\bar{a})=a$ and $h_1(\bar{a})=h_2(a)=e$ for $a\in\{0,1\}$, it is the case that $L_{\{0,1\}}=\text{Eq}\,(h_1,h_2)$. It is shown in [3], [7] that $L_{\{0,1\}}$ is a full semiAFL generator for the class of all recursively enumerable sets.

While $L_{\{0,1\}}$ is the equality set for a pair of homomorphisms, it is the case that $L_{\{0,1\}}$ cannot be the equality set for a pair of homomorphisms if either is nonerasing (Lemma 4.1). On the other hand, we show that equality sets of pairs of nonerasing homomorphisms can be used to represent accepting computations of Turing machines (Theorem 4.6) and that this power is sufficient to enable us to represent complexity classes when we use the notation of "balance" of homomorphisms.

The results presented in §§ 2 and 3 are closely related to those in [3]–[5], [7]. However, by considering both the time and the space used by a Turing machine, our representations for the language recognized by a Turing machine are very tight. Further, these representations lead to characterizations of a wide variety of complexity classes of formal languages specified by time-bounded or space-bounded machines.

1. Theories of automata, computability and formal languages. It is assumed that the reader is familiar with the basic concepts from the theories of automata, computability and formal languages. We review some of the concepts that are most important for this paper, discuss the notion of equality sets of homomorphisms, and establish notation.

For a string w, $|w|$ denotes the length of w.

A *homomorphism* is a function $h:\Sigma^*\to\Delta^*$ such that for all x, $y\in\Sigma^*$, $h(xy)=h(x)h(y)$. A homomorphism is *nonerasing* if for all strings w, $|w|>0$ implies $|h(w)|>0$; otherwise, h is *erasing.*

Let h_1, h_2 be homomorphisms. Define the *equality set of* h_1, h_2 by $\text{Eq}\,(h_1,h_2)=\{w\,|\,h_1(w)=h_2(w)\}$ A language L is an *equality set* if there exist two homomorphisms h_1, h_2 such that L is the equality set of h_1, h_2.

Note that there are equality sets that are regular, or context-free and nonregular, or non-context-free. For example, if $L_1=\Sigma^*$ and for each $a\in\Sigma$, $h_1(a)=h_2(a)$, then $\text{Eq}\,(h_1,h_2)=L_1$; if $L_2=\{w\in\{a,b\}^*|$ the number of a's in w equals the number of b's in $w\}$ and $g_1(a)=g_2(b)=a$ and $g_1(b)=g_2(a)=e$, then $\text{Eq}\,(g_1,g_2)=L_2$; if $L_3=\{cb^2a^4b^8\cdots a^{2^{n-2}}b^{2^{n-1}}d^{2^n}\,|\,n\geqq 2,\ n\ \text{even}\}$ and $f_1(a)=f_1(d)=a$, $f_1(b)=b$, $f_1(c)=c$, $f_2(a)=b^2$, $f_2(b)=a^2$, $f_s(c)=cb^2$, $f_2(d)=e$, then $\text{Eq}\,(f_1,f_2)=L_3$. See [7] for more discussion concerning equality sets.

It is clear that every equality set is recursive. In fact, every equality set can be recognized in linear time by a two-way deterministic Turing machine with one work tape. Also, every equality set can be recognized by a two-way deterministic Turing machine that uses only log n work space.

Unless otherwise specified, a *Turing machine* will be a nondeterministic machine with exactly one tape and exactly one read-write head. The *language* accepted by a

Turing machine M is denoted by $L(M)$. A Turing machine may operate within some time bound T or space bound S, where T and S are functions of the length of the input string. Both time and space bounds are assumed to be nondecreasing. With this model of a Turing machine, the time and space bounds are bounded below by linear functions.

2. First result. In this section we develop our first result. This result is closely related to the proposition stated in the Introduction.

Let $L \subseteq \Sigma^*$ be the language and let $h : \Sigma^* \to \Delta^*$ be a homomorphism. If f is a function such that for some $k > 0$ and for all but finitely many $w \in L$, $|w| \leqq kf(|h(w)|)$, then we say that *h is f-erasing on L.*

THEOREM 2.1. *Let M be a Turing machine, let T be the function that measures M's running time and let S be the function that measures the space M uses. Then there exist homomorphisms h_0, h_1, h_2 and a regular set R such that $h_0(\mathrm{Eq}\,(h_1, h_2) \cap R) = L(M)$ and h_0 is $T \cdot S$-erasing on $\mathrm{Eq}\,(h_1, h_2) \cap R$.*

While Theorem 2.1 uses equality sets, regular sets and homomorphisms instead of fixed points of DGSM mappings or minimal subsets of equality sets, the essential difference between Theorem 2.1 and the proposition stated in the Introduction is the attention paid to the time and space bounds and the amount of erasing allowed.

Before proving the result, we introduce some notation.

Let Σ be an alphabet and let x and y be strings of the same length over Σ, say $x = a_1 \cdots a_n$ and $y = b_1, \cdots b_n$, where each $a_i, b_i \in \Sigma$. The *pairing of x and y* is the string $[x, y] = (a_1, b_1)(a_2, b_2) \cdots (a_n, b_n)$ in $(\Sigma \times \Sigma)^*$. For strings x, y, the string $[x, y]$ is said to be the *parallel encoding* of the pair x, y. For properties of this encoding, see [14].

We wish to consider copies of the alphabet Σ. To do this we consider for each symbol $a \in \Sigma$, a new symbol $\bar{a}$ which we call the *barred copy of a*. Then we let $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$ be the *barred copy of* Σ, so that $\Sigma \cap \bar{\Sigma} = \phi$. For a string $w = a_1 \cdots a_n$, each $a_i \in \Sigma$, the *barred copy of* w is $\bar{w} = \bar{a}_1 \cdots \bar{a}_n$.

Proof of Theorem 2.1. Let $M = (K, \Sigma, \Gamma, \delta, q_0, F)$ be a Turing machine, where K is the set of states, Σ is the input alphabet, Γ is the tape alphabet, $\Sigma \subseteq \Gamma$, δ is the set of instructions, $q_0 \in K$ is the initial state, and $F \subseteq K$ is the set of accepting states. Let $\mathrm{B} \in \Gamma$ denote the blank symbol and let $\#$ be a new symbol, $\# \notin K \cup \Gamma$. We represent an instantaneous description (ID) of M as a string $xqy \in \Gamma^* K \Gamma^*$ and interpret this to mean that q is the current state of M's computation, xy is the current content of M's tape, the read-write head of M is currently scanning the first symbol of y, and the blank symbol B occurs only as the first or last symbol of xy. For any string w accepted by M, there exists at least one sequence $\mathrm{ID}_0, \mathrm{ID}_1, \cdots, \mathrm{ID}_t$ of instantaneous descriptions such that $\mathrm{ID}_0 = q_0 w$, $\mathrm{ID}_t \in \Gamma^* F \Gamma^*$, and for each $i = 0, \cdots, t-1$, ID_{i+1} is a successor to ID_i according to the set δ of M's instructions. For each w accepted by M and each sequence of instantaneous descriptions representing an accepting computation of M on w, we wish to consider a string in the following form:

$$(*) \quad [\overline{Bw\,\#}, \mathrm{ID}_0\#][\mathrm{ID}_0\#, \mathrm{ID}_1\#][\mathrm{ID}_1\#, \mathrm{ID}_2\#] \cdots [\mathrm{ID}_i\#, \mathrm{ID}_{i+1}\#]$$
$$\cdots [\mathrm{ID}_{t-2}\#, \mathrm{ID}_{t-1}\#][\mathrm{ID}_{t-1}\#, \mathrm{ID}_t\#][\mathrm{ID}_t\#, \overline{\mathrm{ID}_t\#}],$$

where for each i, at most one of ID_i, ID_{i+1} contains an occurrence of the blank symbol.

Let $\Delta = K \cup \Gamma \cup \{\#\} \cup \bar{K} \cup \bar{\Gamma} \cup \{\bar{\#}\}$. Let h_1 and h_2 be the homomorphisms from $(\Delta \times \Delta)^*$ to $(K \cup \Gamma \cup \{\#\})^*$ determined by defining

$$h_1(a, b) = \begin{cases} a & \text{if } a \text{ is not barred and } a \neq B, \\ e & \text{otherwise;} \end{cases}$$

and

$$h_2(a,b)=\begin{cases} b & \text{if } b \text{ is not barred and } b \neq B,\\ e & \text{otherwise.}\end{cases}$$

Now consider any w accepted by M, any sequence of instantaneous descriptions representing an accepting computation of M on w, and the corresponding string z having the form of (*). From the definition of h_1 and h_2, it is clear that $h_1(z)=h_2(z)=\mathrm{ID}_0\#\mathrm{ID}_1\#\cdots\mathrm{ID}_{t-1}\#\mathrm{ID}_t\#$ so that $z\in\mathrm{Eq}(h_1,h_2)$. Let $h_0\colon(\Delta\times\Delta)^*\to\Sigma^*$ be the homomorphism determined by defining $h_0(\bar{a},b)=a$ for all $\bar{a}\in\bar{\Sigma}$ and all $b\in\Delta$ and $h_0(c,b)=e$ for all $c\notin\bar{\Sigma}$ and all $b\in\Delta$. Clearly $h_0(z)=w$, so we have $L(M)\subseteq\{h_0(z)\,|\,z\in\mathrm{Eq}(h_1,h_2)\}$.

We wish to obtain precisely $L(M)$ and so we will construct a regular set R such that $\mathrm{Eq}(h_1,h_2)\cap R$ is the set of all strings of the form (*) that represent accepting computations of M. To do this we describe the accepting computations of a finite-state machine D.

The finite-state machine D has input alphabet $\Delta\times\Delta$. D begins its computation on an input string z by checking whether z begins with a prefix of the form $[Bw\#, q_0w\#]$. If z does begin with such a prefix, then D checks to see that z is made up by concatenating strings of the form $[u\#, v\#]$, where u is a candidate for an instantaneous description with at most one occurrence of the blank symbol B; i.e., $u\in\Gamma^*K\Gamma^*\cup K\{B\}\Gamma^*\cup\Gamma^*K\{B\}$, where v is also a candidate for an instantaneous description of M with at most occurrence of B, and where v is a successor of u according to the set δ of instructions of M. It should be noted that the finite-state machine D is capable of checking whether a string $[u\#, v\#]$ has the proper form since the pair of strings $(u\#, v\#)$ have been represented by means of the parallel encoding, and that the difference between u and v, when they do represent successive instantaneous descriptions, occurs in at most three adjacent symbols. Finally, the string z must end with a suffix of the form $[u\#, u\#]$ where u represents an accepting instantaneous description of M.

Let R be the set of strings accepted by D. Since a string in $\mathrm{Eq}\,(h_1,h_2)\cap R$ is in R, any portion of the form $[\mathrm{ID}_i\#, \mathrm{ID}_{i+1}\#]$ represents successive instantaneous descriptions of M. The equality mechanism of (h_1,h_2) forces the first component of $[\mathrm{ID}_i\#, \mathrm{ID}_{i+1}\#]$ to agree with the second component of $[\mathrm{ID}_{i-1}\#, \mathrm{ID}_i\#]$ as long as occurrences of B are discounted. Hence, we claim that $\{h_0(z)\,|\,z\in\mathrm{Eq}\,(h_1,h_2)\cap R\}$ is precisely $L(M)$.

If z is a string in $\mathrm{Eq}\,(h_1,h_2)\cap R$ and $w=h_0(z)$, then z represents an accepting computation of M on w. Since M runs in time T, z is the concatenation of at most $T(|w|)+1$ strings of the form $[u\#, v\#]$. Since M uses at most $S(|w|)$ space during its computation, each instantaneous description has length at most $S(|w|)$. Thus, it is clear that $|z|\leqq 2T(|w|)S(|w|)$. Since $w=h_0(z)$, we see that h_0 is $T\cdot S$-erasing on $\mathrm{Eq}\,(h_1,h_2)\cap R$. □

Theorem 2.1 can be restated in terms of a DGSM mapping applied to an equality set or a homomorphism applied to the fixed point of a DGSM mapping or a homomorphism applied to the minimal subset of an equality set. In each case the amount of erasing performed is bounded by the product of time and space used by the Turing machine. Formal statements parallel the proposition stated in the Introduction.

The proof of Theorem 2.1 shows that the set of accepting computations of an arbitrary Turing machine can be represented as $\mathrm{Eq}\,(h_1,h_2)\cap R$ for some h_1, h_2, and R. If the running time and the work space of the Turing machine are bounded by total recursive functions, then a recursive set is accepted; otherwise, the running time and the work space are bounded by partial recursive functions. Since each set of the form $Eq\ (h_1, h_2) \cap R$ is recursive, the class of all such sets forms a "basis" for the class of recursively enumerable sets since each recursively enumerable set is the homomorphic image of a set of the form $\mathrm{Eq}\ (h_1, h_2) \cap R$. Thus all of the "usual" questions such as finiteness or emptiness are undecidable for arbitrary languages of the form $\mathrm{Eq}\ (h_1, h_2) \cap R$ (see [12]). This will be true for the class of fixed points of DGSM mappings etc., for exactly the same reason.

3. Complexity classes of formal languages. In this section we consider complexity classes of formal languages. We establish characterizations of many such classes in terms of equality sets of pairs of homomorphisms and some simple operations.

For a time bound T, let $\mathrm{NTIME}(T) = \{L(M) \mid M$ is a nondeterministic multi-tape Turing machine that runs in time $T\}$. If $\mathcal{B}$ is a set of time bounds, then let $\mathrm{NTIME}(\mathcal{B}) = \bigcup_{T \in \mathcal{B}} \mathrm{NTIME}(T)$. Similarly, for a space bound S, let $\mathrm{NSPACE}(S) = \{L(M) \mid M$ is a nondeterministic Turing machine that uses space $S\}$, and if $\mathcal{B}$ is a set of space bounds, then let $\mathrm{NSPACE}(\mathcal{B}) = \bigcup_{S \in \mathcal{B}} \mathrm{NSPACE}(S)$. As special cases, let NP be the class of languages accepted by nondeterministic Turing machines that run in polynomial time and let PSPACE be the class of languages accepted by Turing machines that use polynomial space.

Let $\mathcal{B}$ be a set of functions that serve as time bounds. Such a class $\mathcal{B}$ is said to be a *good* set if it has the following properties:

(i) there exists a function t in $\mathcal{B}$ such that for all $n \geqq 0$, $t(n) \geqq n^2$;

(ii) $\mathcal{B}$ is closed under composition.

Notice that each of the following sets of time bounds is good: the set of polynomials; for each $k \geqq 3$, the Grzegorzcyk class ξ^k; the set of primitive recursive functions; the set of total recursive functions. We·will provide new characterizations of classes of languages of the form $\mathrm{NTIME}(\mathcal{B})$ for good sets $\mathcal{B}$. To do so we must make certain definitions.

Let $\mathcal{B}$ be a set of functions. Let $L \subseteq \Sigma^*$ be a language and let $h : \Sigma^* \to \Delta^*$ be a mapping. The mapping h is *$\mathcal{B}$-erasing on L* if for some function $f \in \mathcal{B}$, h is f erasing on L. A class $\mathcal{L}$ of languages is *closed under $\mathcal{B}$-erasing homomorphisms* (DGSM *mappings*) if for every $L \in \mathcal{L}$ and every homomorphism (DGSM mapping) h that is $\mathcal{B}$-erasing on L, $h(L) \in \mathcal{L}$.

Now we can establish our characterizations of classes of the form $\mathrm{NTIME}(\mathcal{B})$.

THEOREM 3.1. *Let $\mathcal{B}$ be a good set of time bounds. For every language $L \in \mathrm{NTIME}(\mathcal{B})$, there exist homomorphisms h_0, h_1, h_2 and a regular set R such that $h_0(\mathrm{Eq}\ (h_1, h_2) \cap R) = L$ and h_0 is $\mathcal{B}$-erasing on* $\mathrm{Eq}\ (h_1, h_2)$.

Proof. If $L \in \mathrm{NTIME}(\mathcal{B})$, then there exists a single-tape Turing machine M and a function $T \in \mathcal{B}$ such that $L(M) = L$ and M runs in time T. Since M runs in time T, M uses no more than T space. Thus by Theorem 2.1, there exist homomorphisms h_0, h_1, h_2 and a regular set R such that $h_0(\mathrm{Eq}\ (h_1, h_2) \cap R) = L(M) = L$ and h_0 is T^2-erasing on $\mathrm{Eq}\ (h_1, h_2) \cap R$, where $T^2(n) = (T(n))^2$. Since $T \in \mathcal{B}$, there is a function f in $\mathcal{B}$ such that for all $n \geqq 0$, $f(n) \geqq T^2(n)$. Thus h_0 is $\mathcal{B}$-erasing on $\mathrm{Eq}\ (h_1, h_2) \cap R$. □

THEOREM 3.2. *Let $\mathcal{B}$ be a good set of time bounds. The class* $\mathrm{NTIME}(\mathcal{B})$ *is the smallest class containing all equality languages and closed under $\mathcal{B}$-erasing homomorphisms and intersection with regular sets.*

Proof. By the definition of a good set, the function $f(n) = n^2$ is majorized by a function in $\mathcal{B}$. As noted above for any two homomorphisms h_1, h_2 the language $\mathrm{Eq}\ (h_1, h_2)$ can be recognized by a deterministic single-tape Turing machine in time n^2. Hence, every equality set is in NTIME($\mathscr{B}$). Clearly NTIME($\mathscr{B}$) is closed under intersection with regular sets. It is well known that $\mathscr{B}$ is closed under composition if and only if NTIME($\mathscr{B}$) is closed under $\mathscr{B}$-erasing homomorphisms. Thus NTIME($\mathscr{B}$) is a class containing all equality languages and closed under $\mathscr{B}$-erasing homomorphisms and intersection with regular sets. The fact that NTIME($\mathscr{B}$) is the *smallest* such class now follows from Theorem 3.1. □

Theorems 3.1 and 3.2 provide representations for each of the following classes of languages:

(a) NP;

(b) for each $k \geqq 3$, the class ξ_*^k of languages accepted by Turing machines that run within time bounds in the Grzegorcyzk class ξ^k;

(c) the class of primitive recursive sets;

(d) the class of recursive sets.

Now we turn to a representation of PSPACE.

Let R be a binary relation on strings over an alphabet Σ.

The relation R is *length-preserving* if for all x, y, when $R(x, y)$ holds, then $|x| = |y|$. Let R^* be the transitive reflexive closure of R. If $\#$ is a symbol not in Σ, then let $\mathrm{SE}_{\#}(R) = \{x \# y \mid x, y \in \Sigma^* \text{ and } R(x, y) \text{ holds}\}$. If $L \subseteq \Sigma^*$ and $a \in \Sigma$, let $R_a(L) = \{\langle x, y\rangle \mid x, y \in (\Sigma - \{a\})^* \text{ and } xay \in L\}$.

A class $\mathscr{L}$ of languages is *weakly transitively closed* [1] if the following condition holds: Let $L \in \mathscr{L}$, let Σ be a finite alphabet such that $L \subseteq \Sigma^*$, and let $a \in \Sigma$. If $R_a(L)$ is length-preserving, then $SE_a(R_a^*(L)) \in \mathscr{L}$.

THEOREM 3.3. *The class* PSPACE *is the smallest weakly transitively closed class containing all equality languages and closed under intersection with regular sets and polynomial-erasing homomorphisms.*

Theorem 3.3 follows from Theorem 2.1 and the techniques of [1]. It should be noted that the representation given in Theorem 3.3 can be generalized to other classes of languages specified by space-bounded machines. However, as soon as a set $\mathscr{B}$ of space bounds contains a function that majorizes 2^n and is is closed under composition, then NSPACE($\mathscr{B}$) = NTIME($\mathscr{B}$) as long as the set $\mathscr{B}$ is suitably well-behaved. Thus no new information is gained from generalizing Theorem 3.3.

We have chosen to present our results in terms of equality sets. It should be noted that if one chooses to discuss fixed-point languages for DGSM's or minimal subsets of equality sets, the analogous results follow easily since in each case a result similar to Theorem 2.1 can be established by showing that the set of accepting computations of a Turing machine can be represented appropriately.

4. Subclasses of the class of all equality sets. Here we investigate subclasses of the class of all equality sets and we strengthen the representation theorems for complexity classes given in § 3.

In [7] a "hardest" equality set is introduced. For strings $x, y \in \Sigma^*$, let $\mathrm{shuffle}(x, y) = \{x_1y_1x_2y_2 \cdots x_ny_n \mid x = x_1 \cdots x_n, y = y_1, \cdots, y_n\}$ and let $L_\Sigma = \bigcup\{\mathrm{shuffle}\ (w, \bar{w}) \mid w \in \Sigma^*\}$. Note that $L_\Sigma = \mathrm{Eq}\ (h_1, h_2)$, where $h_1(a) = h_2(\bar{a}) = a$ and $h_1(\bar{a}) = h_2(a) = e$ for $a \in \Sigma$. In [7] it is shown that the class of all equality sets is the smallest class containing $L_{\{0,1\}}$ and closed under inverse homomorphism. This means that Theorems 2.1 and 3.1–3.3 can be reformulated in terms of $L_{\{0,1\}}$ and closure under inverse homomorphism. In this form Theorem 2.1 yields as corollary the fact that $L_{\{0,1\}}$ is a full semi-AFL generator (and also a cylinder generator) of the class of all recursively enumerable sets.

Let us point out a combinatorial property of sets of the form L_Σ. The proof of this result is in the Appendix.

LEMMA 4.1. *For any finite alphabet Σ with at least two elements, the language L_Σ cannot be represented as the equality set of two homomorphisms if either homomorphism is nonerasing; i.e., if $L_\Sigma = \mathrm{Eq}\,(h_1, h_2)$, then both h_1 and h_2 are erasing.*

In contrast to Lemma 4.1, notice that for homomorphisms h_1, h_2 from Σ^* to $\{a\}^*$, where a is a single symbol, the set $\mathrm{Eq}\,(h_1, h_2)$ is a context-free language [11]; in fact, it is a deterministic one-counter language. Further, for every two homomorphisms h_1, h_2 mapping Σ^* to $\{a\}^*$, there exist two nonerasing homomorphisms h_1', h_2' such that $\mathrm{Eq}\,(h_1', h_2') = \mathrm{Eq}\,(h_1, h_2)$—for every $b \in \Sigma$ define $h_1'(b) = h_1(b)a$ and $h_2'(b) = h_2(b)a$.

Let us return to Theorem 2.1 and its proof. The homomorphisms h_1 and h_2 are defined in such a way that they do perform erasing. Lemma 4.1 suggests that this must be the case. However, we will show (Theorem 4.6) that a result similar to Theorem 2.1 can be obtained when one restricts attention to equality sets of pairs of nonerasing homomorphisms. We proceed to develop the machinery necessary for the proof of this result.

Consider two homomorphisms h_1, h_2 from Σ^* to Δ^* and a word x in Σ^*. Then, the *balance of* x is defined by $\mathrm{bal}\,(x) = \mathrm{abs}\,(|h_1(x)| - |h_2(x)|)$, where for any integer i, $\mathrm{abs}\,(i)$ is the absolute value of i.

Let $f: N \to N$ be a bounding function. *Two homomorphisms h_1, h_2* have *f-bounded balance on a language L* if for every x in L and each prefix y of x, $\mathrm{bal}\,(y) \leqq f(|x|)$.

If the pair (h_1, h_2) has f-bounded balance on $\mathrm{Eq}\,(h_1, h_2)$, then (h_1, h_2) is said to have f-bounded balance and $\mathrm{Eq}\,(h_1, h_2)$ is called an *equality set with f-bounded balance.*

Let $\widehat{\mathrm{EQ}}(f)$ $(\mathrm{EQ}(f))$ denote the *family of equality sets* $\mathrm{Eq}\,(h_1, h_2)$ *with f-bounded balance* (where h_1, h_2 are nonerasing homomorphisms), and for a set $\mathcal{F}$ of bounding functions, let $\widehat{\mathrm{EQ}}(\mathcal{F}) = \bigcup_{f\in\mathcal{F}} \widehat{\mathrm{EQ}}(f)$ and $\mathrm{EQ}(\mathcal{F}) = \bigcup_{f\in\mathcal{F}} \mathrm{EQ}(f)$.

For every $x \in \Sigma^*$ and every two homomorphisms h_1, h_2 on Σ^*, $\mathrm{bal}\,(x) \leqq k|x|$ where $k = \max\{\mathrm{abs}\,(|h_1(a)| - |h_2(a)|) \mid a \in \Sigma\}$. Thus every pair of homomorphisms has linear-bounded balance on every language in the domain of the homomorphisms. Also, notice that for every f, $\mathrm{Eq}\,(f) \subseteq \widehat{\mathrm{Eq}}(f)$, and if f and g are functions such that for all $n \geqq 0$, $f(n) \leqq g(n)$, then $\widehat{\mathrm{Eq}}(f) \subseteq \widehat{\mathrm{Eq}}(g)$ and $\mathrm{Eq}\,(f) \subseteq \mathrm{Eq}\,(g)$.

Now we briefly describe results on machines which correspond to equality sets with f-bounded balance.

A *Post machine M* is a deterministic acceptor with a one-way, read-only input tape, finite-state control, and one work tape. The work tape operates as a Post tape or a queue, with two heads W and R: W is a write only head and R is read only. Both heads are restricted to move only from left to right and M is restricted to operate in quasi-realtime. Initially, the work tape of M is empty and W and R scan the same cell. M accepts in a final state when W and R scan the same cell. (For examples of Post machines operating without time bounds, see [10], and for more complex Post machines, see [2].)

If c is a configuration of M, then the *distance* of c, $d(c)$, is defined by the number of cells from R to W. Thus, if c is an initial or an accepting configuration, then $d(c) = 0$.

A Post machine M has *f-bounded distance* if for every x in $L(M)$ there is an accepting computation of M on x such that for each configuration c in the computation $d(c) \leqq f(|x|)$.

The following facts follow readily from the definitions.

LEMMA 4.2. *For every equality set* $\mathrm{Eq}\,(h_1, h_2)$ *with f-bounded balance, there exists a Post machine M with f-bounded distance such that $L(M) = \mathrm{Eq}\,(h_1, h_2)$.*

LEMMA 4.3. (i) *For every Post machine M_1 with f-bounded distance, there is a deterministic Turing machine M_2 with a one-way input tape and one work tape with one* *head such that* (a) M_2 *is f-space-bounded,* (b) M_2 *operates in time* $knf(n)$ *for some* $k \geqq 1$, *and* (c) $L(M_2) = L(M_1)$.

(ii) *For every Post machine* M_1 *with f-bounded distance, there is a deterministic single-tape Turing machine* M_2 *operating in time* $cnf(n)$ *for some* $c > 0$ *such that* $L(M_2) = L(M_1)$.

It is known [3], [4], [11] that for every two homomorphisms h_1, h_2 the set Eq (h_1, h_2) is regular if and only if Eq (h_1, h_2) has f-bounded balance for some function f such that there exists k with the property that for all $n > 0$, $f(n) \leqq k$. In [13] it is shown that if M is a deterministic Turing machine that reads its input in only one direction and is f-space bounded, then $L(M)$ nonregular implies that $\liminf (f(n)/\log n) > 0$. Thus, from Lemma 4.2 and Lemma 4.3(i), we obtain the following fact.

THEOREM 4.4. *If L is an equality set with f-bounded balance and L* is *not regular, then* $\liminf (f(n)/\log n) > 0$.

Bounds on the balance can also be obtained for equality sets that are of the form L_Σ.

THEOREM 4.5. *For every alphabet* Σ *containing at least two elements, if* h_1, h_2 *are homomorphisms such that* Eq $(h_1, h_2) = L_\Sigma$, *then* (h_1, h_2) *has f-bounded balance where f is such that* $\limsup (f(n)/n) > 0$.

Proof. Let $\{0, 1\} \subseteq \Sigma$. Since Eq $(h_1, h_2) = L_\Sigma$ and (h_1, h_2) has f-bounded balance, from Lemma 4.2 we see that there exists a Post machine M with f-bounded distance such that $L(M) = L_\Sigma$. Let COPY $= \{w\bar{w} \mid w \in \{0, 1\}^*\}$. Notice that COPY $= L_\Sigma \cap \{0, 1\}^*\{\bar{0}, \bar{1}\}^*$ so that COPY is also accepted by a Post machine with f-bounded distance. From Lemma 4.3(ii) we see that this means that COPY is recognized by a deterministic one-tape Turing machine operating in time $cnf(n)$ for some $c > 0$. However, a single-tape Turing machine operating in time g cannot accept COPY unless $\limsup (g(n)/n^2) > 0$ (just as such a machine cannot accept $\{ww^R \mid w \in \{0, 1\}^*\}$ in less time [8]). Thus $\limsup (nf(n)/n^2) > 0$ so that $\limsup (f(n)/n) > 0$. □

Let us summarize what we have learned so far. Consider an alphabet Σ with at least two elements and consider the set L_Σ. Let h_1 and h_2 be homomorphisms such that Eq $(h_1, h_2) = L_\Sigma$. By Lemma 4.1 both h_1 and h_2 must be erasing. From Theorem 4.5 we see that the pair (h_1, h_2) has balance that is bounded below by a linear function. But every pair of homomorphisms has balance that is bounded above by a linear function. Thus the pair (h_1, h_2) has balance that is bounded both above and below by linear functions.

Now we turn to the task of representing complexity classes by using pairs of *nonerasing* homomorphisms instead of the representation of Theorem 2.1 that used erasing homomorphisms.

THEOREM 4.6. *Let M be a Turing machine, let T be the function that measures M's running time, and let S be the function that measures how much space M uses. Then there exist two nonerasing homomorphisms* α *and* β, *a homomorphism* γ, *and a regular set R such that* $\gamma(\text{Eq}\,(\alpha, \beta) \cap R) = L(M)$, γ *is g-erasing on* Eq $(\alpha, \beta) \cap R$, *where* $g(n) = S(n)T(n)$, *and* (α, β) *has square-root-bounded balance on* Eq $(\alpha, \beta) \cap R$.

Proof. We do not provide a formal proof but rather point to certain modifications of the proof of Theorem 2.1 which arise from the condition that α and β are nonerasing. It is advisable for the reader to review that proof at this time.

In the proof of Theorem 2.1 there are two situations where the homomorphisms h_1, h_2 are erasing. First, when the Turing machine extends its work space and reads a blank, and second at the beginning and at the end of the encoding of accepting computations. To solve the first case we consider a new machine that uses all and only the tape squares where its input originally appears, and for the second case we join an appropriate head and tail to the encoding of accepting computations.

Let M be the given Turing machine and let Σ be the input alphabet of M. Consider a new machine M' that processes input strings of the form $wd \cdots d$ where $w \in \Sigma^*$ and d is a new symbol. We want M' to simulate M's computations on w by using all the tape squares where $wd \cdots d$ originally appears and without using any additional work space. If the function S that measures how much space M uses is honest, then M' can determine that the number of d's is correct before simulating M's computations on w; if not, M' can nondeterministically "guess" the correct number of d's and accept the input only if that guess is correct. We restrict attention to M'.

In the proof of Theorem 2.1, we defined homomorphisms h_1, h_2 such that if $z \in \mathrm{Eq}\,(h_1, h_2)$, then $h_1(z) = h_2(z)$ had the form $\mathrm{ID}_0\#\mathrm{ID}_1\# \cdots \mathrm{ID}_{t-1}\#\mathrm{ID}_t\#$ where $\mathrm{ID}_0, \mathrm{ID}_1, \cdots, \mathrm{ID}_t$ was a sequence of instantaneous descriptions making up an accepting computation of M', and z itself was of the form

$$(*) \qquad [\overline{Bw\#}, \mathrm{ID}_0\#][\mathrm{ID}_0\#, \mathrm{ID}_1\#] \cdots [\mathrm{ID}_{t-1}\#, \mathrm{ID}_t\#][\mathrm{ID}_t\#, \overline{\mathrm{ID}_t\#}].$$

While the homomorphisms h_1, h_2 erased certain symbols, here we wish to consider only *nonerasing* homomorphisms. The need to erase occurrences of the blank symbol B is eliminated by considering the machine M'. Thus we need only deal with the erasing of barred symbols.

Let $\Delta' = \Delta \cup \{d\}$. Let g_1 (g_2) be the projection on the first (second) coordinate of $\Delta' \times \Delta'$. Then $g_1(z) = Bw\#\mathrm{ID}_0\# \cdots \#\mathrm{ID}_t\#$ and $g_2(z) = \mathrm{ID}_0\# \cdots \mathrm{ID}_t\#\mathrm{ID}_t\#$. Thus z is not in $\mathrm{Eq}\,(g_1, g_2)$ but the strings $g_1(z)$ and $g_2(z)$ are "almost" the same. (In fact there is no pair (f_1, f_2) of nonerasing homomorphisms such that $z \in \mathrm{Eq}\,(f_1, f_2)$ since the use of the equality set machinery to match consecutive pairs leaves the strings $Bw\#$ and $\mathrm{ID}_t\#$ to be handled in some way.) We will modify the homomorphisms g_1 and g_2 so that a string of the form uzv will be in the equality set of the new pair of homomorphisms.

Let $p, q, r, s, \$, \text{¢}$ be six new symbols, and let the barred copies of $p, q, r, s, \$, \text{¢}$ be new symbols. Let α and β be the homomorphisms determined by defining

$$\alpha(a, b) = \begin{cases} a & \text{if } a \text{ is not barred,} \\ p & \text{if } a \text{ is barred,} \end{cases}$$

$$\alpha(p) = \alpha(\bar{p}) = p, \qquad \alpha(q) = \alpha(\bar{q}) = q,$$

$$\alpha(r) = s, \quad \alpha(\bar{r}) = ss,$$

$$\alpha(s) = r, \quad \alpha(\bar{s}) = rr,$$

$$\alpha(\text{¢}) = \text{¢}, \quad \alpha(\$) = r\$, \quad \alpha(\bar{\text{¢}}) = \text{¢}, \quad \alpha(\bar{\$}) = s\$;$$

$$\beta(a, b) = \begin{cases} b & \text{if } b \text{ is not barred,} \\ s & \text{if } b \text{ is barred,} \end{cases}$$

$$\beta(p) = q, \qquad \beta(\bar{p}) = qq,$$

$$\beta(q) = p, \qquad \beta(\bar{q}) = pp,$$

$$\beta(r) = \beta(\bar{r}) = r, \qquad \beta(s) = \beta(\bar{s}) = s,$$

$$\beta(\text{¢}) = \text{¢}q, \quad \beta(\$) = \$, \quad \beta(\bar{\text{¢}}) = \text{¢}p, \quad \beta(\bar{\$}) = \$.$$

For a string z of form $(*)$, let $n = |[\overline{Bw\#}, \mathrm{ID}_0\#]|$ so that $n = |w| + 2$. If n is even, then consider strings of the form

$$(**) \quad \text{¢}\,\bar{q}\,p\,\bar{p}\,q^2\,\bar{q}\,p^3\,\bar{p}\,q^4\,\bar{q} \cdots p^{n-3}\,\bar{p}\,q^{n-2}\,\bar{q}\,z\,r^{n-2}\,\bar{r}\,s^{n-3}\,\bar{s}\,r^{n-4}\,\bar{r} \cdots \bar{r}\,r^2\,s\,\bar{s}\,\bar{r}\,\$.$$

Notice that $\alpha(\text{¢}\bar{q}p\bar{p} \cdots q^{n-2}\bar{q}zr^{n-2}\bar{r} \cdots s\bar{s}\bar{r}\$) = \text{¢}qp^2 \cdots q^{n-1}\alpha(z)s^n \cdots r^3s^2r\$$ and $\alpha(z) = p^n\mathrm{ID}_0\# \cdots \#\mathrm{ID}_t\#$. On the other hand, $\beta(\text{¢}\bar{q}p\bar{p} \cdots q^{n-2}\bar{q}zr^{n-2}\bar{r} \cdots s\bar{s}\bar{r}\$) =$

$¢qp^2q^3\cdots p^n\beta(z)r^{n-1}\cdots s^2r\$$ and $\beta(z)=\mathrm{ID}_0\#\cdots\#\mathrm{ID}_t\#s^n$. Thus $\alpha(¢\bar{q}p\bar{p}\cdots q^{n-2}\bar{q}zr^{n-2}\bar{r}\cdots s\bar{s}\bar{r}\$)=¢qp^2\cdots q^{n-1}p^n\mathrm{ID}_0\#\cdots\#\mathrm{ID}_t\#s^n\cdots r^3s^2r\$=\beta(¢\bar{q}p\bar{p}\cdots q^{n-2}\bar{q}zr^{n-2}\bar{r}\cdots s\bar{s}\bar{r}\$)$, and so for each string z of the form $(*)$ with n even, the string $¢\bar{q}p\bar{p}\cdots q^{n-2}\bar{q}zr^{n-2}\bar{r}\cdots s\bar{s}\bar{r}\$$ is in Eq (α,β). Similarly, for each string z of the form $(*)$ with n odd, the string

$$(***)\qquad \bar{¢}\,\bar{p}\,q\,\bar{q}\cdots q^{n-2}\,\bar{q}\,z\,r^{n-2}\,\bar{r}\cdots r\,\bar{r}\,\bar{s}\,\bar{\$}$$

is in Eq (α,β) and $\alpha(\bar{¢}\bar{p}q\bar{q}\cdots q^{n-2}\bar{q}zr^{n-2}\bar{r}\cdots r\bar{r}\bar{s}\bar{\$})=¢pq^2p^3\cdots p^n\mathrm{ID}_0\#\cdots\#\mathrm{ID}_t\#s^n\cdots s^3r^2s\$$.

Let $\gamma_0:\{(\Delta'\times\Delta')\cup\{p,q,r,s,\$,¢,\bar{p},\bar{q},\bar{r},\bar{s},\bar{\$},\bar{¢}\}\}^*\rightarrow(\Sigma\cup\{d\})^*$ be the homomorphism determined by defining $\gamma_0(\bar{a},b)=a$ for all $\bar{a}\in\Sigma\cup\{d\}$ and all $b\in\Delta'$ and $\gamma_0(c)=e$ for all $c\notin\Sigma\cup\{d\}\times\Delta'$. If $w\in L(M')$ and z is a string of form $(*)$ encoding an accepting computation of M' on w, then there exist strings u and v such that $uzv\in\mathrm{Eq}\,(\alpha,\beta)$, $\gamma_0(uzv)=w$, and uzv is either of the form $(**)$ or of the form $(***)$. Thus, $L(M')\subseteq\{\gamma_0(uzv)\mid uzv\in\mathrm{Eq}\,(\alpha,\beta)\}$.

To obtain precisely $L(M')$, construct a finite-state machine D' that checks whether the input string uzv begins with $¢$ or $\bar{¢}$, checks the order of alternation of strings in $\{p\}*\{\bar{p}\}\cup\{q\}*\{\bar{q}\}\cup\{r\}*\{\bar{r}\}\cup\{s\}*\{\bar{s}\}$, behaves on z in a manner similar to the machine D in the proof of Theorem 2.1, matches the parity of $|z|$ with $¢$ or $\bar{¢}$, and checks that the final symbol is the correct choice of $\$$ or $\bar{\$}$. If R is the regular set of strings accepted by D', then it is clear that $\{\gamma_0(uzv)\mid uzv\in\mathrm{Eq}\,(\alpha,\beta)\cap R\}=L(M')$.

To obtain $L(M)$ we compose the homomorphism which erases d and is the identity on symbols in Σ with γ_0. Let γ be the resulting homomorphism so that $\{\gamma(uzv)\mid uzv\in\mathrm{Eq}\,(\alpha,\beta)\cap R\}=L(M)$.

At this point the reader should notice that both of the homomorphisms α and β are nonerasing and that $\gamma(\mathrm{Eq}\,(\alpha,\beta)\cap R)=L(M)$. What remains is to show that the amount of erasing that γ performs on strings in Eq $(\alpha,\beta)\cap R$ is properly bounded and that the pair (α,β) of homomorphisms has $n^{1/2}$-bounded balance on Eq $(\alpha,\beta)\cap R$.

First consider the homomorphisms γ_0 and γ. The homomorphism γ_0 erases certain symbols and any string in Eq $(\alpha,\beta)\cap R$ contains some occurrences of those symbols. Let us consider how much erasing γ_0 must perform on strings in Eq $(\alpha,\beta)\cap R$. Any string in Eq $(\alpha,\beta)\cap R$ is of the form uzv where $u\in\{p,q,¢\}^*$, $v\in\{r,s,¢\}^*$ and z is a string of the form $(*)$ encoding an accepting computation of M'. On an input string w, an accepting computation of M uses at most $S(|w|)$ tape squares and runs for at most $T(|w|)$ steps. Thus γ_0 must erase at most $S(|w|)\cdot T(|w|)$ symbols in z. Further γ must erase at most $S(|w|)-|w|$ d's. Both the strings u and v must be erased and the length of u as well as that of v is bounded by $S(n)\cdot(S(n)-1)/2$, where $n=|w|+2$, since the length of any one instantaneous description is bounded by $S(n)$. Thus γ must erase at most $S(|w|)\cdot T(|w|)+(S(|w|)-|w|)+S(n)\cdot(S(n)-1)$ symbols from uzv in order to retrieve the input string w accepted by M in the computation encoded in this way. Since $S(m)\leqq T(m)$ for all m, this means that γ is g-erasing on Eq $(\alpha,\beta)\cap R$, where $g(m)=k\cdot S(m)\cdot T(m)$ for some $k\geqq 1$.

Now let us consider the question of bounded balance. For any string w in $L(M)$ and any string uzv in Eq $(\alpha,\beta)\cap R$ such that $\gamma(uzv)=w$, it is the case that $k\cdot S(|w|)\cdot S(|w|)\leqq|uzv|\leqq c\cdot S(|w|)\cdot T(|w|)$ for some $k\geqq 1$ and $c\geqq 1$ that are independent of $|w|$ (since the length of an instantaneous description in the corresponding computation of M' is bounded by $S(|w|)$). Recall that both of the homomorphisms α and β are nonerasing. The structure of the string uzv and the definitions of α and β are such that for any prefix y of uzv, bal (y) is bounded by the length of an instantaneous description in the corresponding computation of M', i.e., by $S(|w|)$. Since

$S(|w|) \cdot S(|w|) \leqq |uzv|$, this means that $\mathrm{bal}(y) \leqq (|uzv|)^{1/2}$. Hence the pair (α, β) has $n^{1/2}$-bounded balance on $\mathrm{Eq}(\alpha, \beta) \cap R$. □

As in Theorem 2.1, we have considered only single-tape Turing machines. However, results similar to both Theorem 2.1 and also Theorem 4.6 can be obtained for multi-tape Turing machines by using a parallel encoding.

Remark. Notice that in Theorem 4.6 the pair (α, β) of homomorphisms have $n^{1/2}$-bounded balance. If M is a Turing machine which operates within space $S(n)$ and has running time $k^{S(n)}$ for some $k > 1$, then the balance of (α, β) on $\mathrm{Eq}(\alpha, \beta)$ is bounded by $f(n) = c \log n$ for some $c > 0$.

Clearly the representation technique of Theorem 4.6 can be extended in order to represent complexity classes, and when doing so the balance bounds of pairs of homomorphisms on equality sets and on intersections of equality sets and regular sets must range between "log-bounded balance," i.e., $(c \log n)$-bounded balance for some $c > 0$, and "root-bounded balance," i.e., $n^{1/2}$-bounded balance. Specifically, the next results follow from Theorem 4.6, the above remark and the results in § 3.

THEOREM 4.7. *For every L accepted in real time by a nondeterministic multitape Turing machine, there is a pair of nonerasing homomorphisms (h_1, h_2) with root-bounded balance, a regular set R and a homomorphism h, such that $L = h(\mathrm{Eq}(h_1, h_2) \cap R)$, and h is n^3-erasing on* $\mathrm{Eq}(h_1, h_2) \cap R$.

In the last result the bound of n^3 on the amount of erasing allowed can be reduced to n^2 when the version of Theorem 4.6 for multi-tape machines is used.

THEOREM 4.8. *If $\mathcal{B}$ is a good set of time bounds, then* $\mathrm{NTIME}(\mathcal{B})$ *is the smallest class containing all equality sets of nonerasing homomorphisms with root-bounded balance and closed under intersection with regular sets and under $\mathcal{B}$-erasing homomorphisms.*

THEOREM 4.9. *For every (deterministic) context-sensitive language L, there is a regular set R, two nonerasing homomorphisms h_1, h_2 which have log-bounded balance on* $\mathrm{Eq}(\alpha,\beta) \cap R$, *and a homomorphism h such that $L = h(\mathrm{Eq}(h_1,h_2) \cap R)$ and h is exponential erasing on* $\mathrm{Eq}(h_1,h_2) \cap R$.

THEOREM 4.10. *For every language $L \in$* PSPACE, *there is a pair of nonerasing homomorphisms (h_1, h_2) with log-bounded balance, a regular set R and a homomorphism h such that $L = h(\mathrm{Eq}(h_1, h_2) \cap R)$ and for some constants $c > 1$, $k > 0$, h is c^{n^k}-erasing on* $\mathrm{Eq}(h_1, h_2) \cap R$.

Recall that L_Σ is neither an equality set of a pair of nonerasing homomorphisms nor an equality set with f-bounded balance for any function f with $\lim\sup (f(n)/n) = 0$. On the other hand, the class $\{L \cap R \mid L \in \mathrm{Eq}(\log),\ R \text{ is regular}\}$ is a basis for the class of recursively enumerable sets, as well as many complexity classes.

Note that results similar to those in this section can be obtained by considering fixed-point languages and DGSM mappings.

Theorems 4.7–4.10 are similar to some results of Culik [4].

To see the relationship between the bounds on the balance and the bounds on the amount of erasing allowed, consider the following facts.

(1) A language L is regular if and only if there exist a (nonerasing) homomorphism h, an integer k, a language $L_0 \in \mathrm{Eq}(k)$, and a regular set R such that $h(L_0 \cap R) = L$.

(2) A language L is in NP if and only if there exist a homomorphism h, a language $L_0 \in \mathrm{Eq}$ (root), and a regular set R such that $h(L_0 \cap R) = L$ and h is polynomial erasing on $L_0 \cap R$.

(3) A language L is in PSPACE if and only if there exist a homomorphism h, a language $L_0 \in \mathrm{Eq}(\log)$, and a regular set R such that $h(L_0 \cap R) = L$ and h is f-erasing on $L_0 \cap R$, where for some $c > 1$, $k \geqq 1$, and all n, $f(n) = c^{n^k}$.

(4) A language L is recursively enumerable if and only if there exist a homomorphism h, a language $L_0 \in \mathrm{Eq}$ (log), and a regular set R such that $h(L_0 \cap R) = L$.

One of the referees has noted that cases (2) and (3) require two different proofs.

5. Additional results. We close by sketching some additional results based on a variation on the notion of balance.

Let h_1, h_2 be two homomorphisms from Σ^* to Δ^* and let g be a mapping from Σ^* to Γ^*. Let f be a bounding function. We say that (h_1, h_2) has *f-output bounded balance on a language* $L \subseteq \Sigma^*$ *with respect to* g, if for every x in L and each prefix y of x, abs $(\beta(y)) \leqq f(|g(x)|)$. If $L = \mathrm{Eq}\,(h_1, h_2)$, then we say that the pair (h_1, h_2) has *f-output bounded balance with respect to* g [3], [4].

In this case, the balance bound depends on the length of the image of $x \in L$ under g and not on the length of x. The notion of f-output bounded balance on L with respect to g can be defined for fixed points of DGSM mappings and on languages of the form $L = \mathrm{Eq}\,(h_1, h_2) \cap R$, where R is a regular set and h_1, h_2 are homomorphisms.

LEMMA 5.1. *For any language* L, $L \in \mathrm{NSPACE}(n)$ *if and only if* $L = g(\mathrm{Eq}(h_1, h_2))$, *where* h_1, h_2 *are homomorphisms, which may be chosen to be nonerasing such that* (h_1, h_2) *has linear output balance with respect to g and g is an exponential erasing* DGSM *mapping.*

The proof is an application of the arguments used in § 4.

THEOREM 5.2. *Let* $\mathscr{F}_1$ *and* $\mathscr{F}_2$ *be good sets of bounding functions. For every* DGSM *mapping g and every two homomorphisms* h_1, h_2 *where g is* $\mathscr{F}_1$*-erasing on* $\mathrm{Eq}\,(h_1, h_2)$ *and* (h_1, h_2) *has* $\mathscr{F}_2$*-output bounded balance with respect to g, the language* $g(\mathrm{Eq}(h_1, h_2))$ *is accepted by a nondeterministic Turing machine, which operates in time* f_1, *with* $f_1 \in \mathscr{F}_1$, *and space* f_2, *with* $f_2 \in \mathscr{F}_2$. *Conversely, if L is a language accepted by such a machine, then L can be represented as* $g(\mathrm{Eq}\,(h_1, h_2))$, *where g,* h_1 *and* h_2 *have the properties stated above.*

Proof. Let g be a DGSM mapping, let h_1, h_2 be two homomorphisms satisfying the hypothesis, and let $L = g(\mathrm{Eq}\,(h_1, h_2))$. For $w \in L$, a nondeterministic Turing machine M with one work tape guesses symbol-by-symbol a string x such that $x \in \mathrm{Eq}\,(h_1, h_2)$ and $g(x) = w$. On each guessed symbol a, M evaluates $h_1(a)$ and $h_2(a)$ and stores the difference between the images under h_1 and h_2 of the word guessed so far on its work tape. Simultaneously, M simulates g in its finite control, and if the symbol a causes a nonempty output of g, M compares this output with the input under the input head and moves this head right. The machine M accepts input w if and only if w is completely scanned, the work tape is empty, and the DGSM g, simulated by M, is in an accepting state. Now $L = L(M)$. Since (h_1, h_2) has $\mathscr{F}_2$-output bounded balance with respect to g, the distance between h_1 and h_2 on any prefix of x is bounded by $f_2(|w|)$ with $f_2 \in \mathscr{F}_2$, where $w = g(x)$. Thus, M is f_2-space bounded. On input $w \in L$, M makes $c \cdot |x|$ steps with $c > 0$ and $g(x) = w$. Since g is $\mathscr{F}_1$-erasing on $\mathrm{Eq}\,(h_1, h_2)$, the running time of M is bounded by some $f_1 \in \mathscr{F}_1$.

Notice that h_1, h_2 may be chosen to be nonerasing.

The proof of the converse is straightforward. □

COROLLARY 5.3. *Let* $\mathscr{F}_1$ *and* $\mathscr{F}_2$ *be classes of good bounding functions. For every* DGSM *mapping g and every two homomorphisms* h_1, h_2, *if g is* $\mathscr{F}_1$*-erasing on* $\mathrm{Eq}\,(h_1, h_2)$ *and* (h_1, h_2) *has* $\mathscr{F}_2$*-output bounded balance with respect to g, then*

(i) *if* $\mathscr{F}_1 \subseteq \mathscr{F}_2$, *then* $L = g(\mathrm{Eq}\,(h_1, h_2))$ *if and only if* $L \in \mathrm{NTIME}(\mathscr{F}_1)$;

(ii) *if* $\mathscr{F}_1 \supseteq \exp\,(\mathscr{F}_2)$, *then* $L = g(\mathrm{Eq}\,(h_1, h_2))$ *if and only if* $L \in \mathrm{NSPACE}(\mathscr{F}_2)$;

(iii) *if* $\mathscr{F}_2 \subsetneq \mathscr{F}_1 \subseteq \exp(\mathscr{F}_2)$, *then* $L \in \mathrm{NTIME}(\mathscr{F}_2)$ *implies that* $L = g(\mathrm{Eq}\,(h_1, h_2))$ *and* $L \in g(\mathrm{Eq}\,(h_1, h_2))$ *implies that* $L \in \mathrm{NSPACE}(\mathscr{F}_2)$.

(Here, $\exp\,(\mathscr{F}_2) = \{k^f \mid k > 0, f \in \mathscr{F}_2\}$.)

Notice that these statements also hold for the set of bounded functions $\mathcal{B}$, where NTIME($\mathcal{B}$), NSPACE($\mathcal{B}$) are the regular sets and for unbounded (partial recursive) functions and the r.e. sets.

COROLLARY 5.4. (i) *A language L is in* NP *if and only if there exists a polynomial erasing* DGSM *mapping g and two homomorphisms h_1, h_2 with polynomial output balance with respect to g such that $L = g(\mathrm{Eq}\,(h_1, h_2))$.*

(ii) *A language L is in* PSPACE *if and only if there exist c, $k > 0$, a c^{n^k}-erasing* DGSM *mapping g and two homomorphisms h_1, h_2 with polynomial output bounded balance with respect to g such that $L = g(\mathrm{Eq}\,(h_1, h_2))$.*

COROLLARY 5.5. *Let $\mathcal{F}$ be a class of good bounding functions properly between the polynomials and the Grzegorzcyk class ξ^3, and let* $\mathrm{Eq}\,(\mathcal{F}) = \{L \mid L = g(\mathrm{Eq}\,(h_1, h_2))$, *$g$ is an $\mathcal{F}$-erasing* DGSM *on* $\mathrm{Eq}\,(h_1, h_2)$ *and the pair of homomorphisms (h_1, h_2) has polynomial output bounded balance with respect to $g\}$. Then* $\mathrm{NP} \subseteq \mathrm{Eq}\,(\mathcal{F}) \subseteq \mathrm{PSPACE}$.

Appendix. The purpose of this Appendix is to provide a proof of Lemma 4.1. Recall that for any alphabet Σ, the language L_Σ is defined to be $L_\Sigma = \bigcup\{\text{shuffle}\,(w, \bar{w}) \mid w \in \Sigma^*\}$, ·and there exist homomorphisms h_1 and h_2 such that $\mathrm{Eq}\,(h_1, h_2) = L$.

LEMMA 4.1. *For any finite alphabet Σ with at least two elements, the language L_Σ cannot be represented as the equality set of two homomorphisms if either homomorphism is nonerasing; i.e., if $L = \mathrm{Eq}\,(h_1, h_2)$, then both h_1 and h_2 are erasing.*

To prove this result, we use certain combinatorial properties of strings.

Facts. (1) If two *nonempty* strings a, b have the property that $ab = ba$, then we say that a and b *commute.* If a and b commute, then there exist a unique nonempty shortest string c and unique largest integers p, $q > 0$ such that $a = c^p$ and $b = c^q$ [9].

(2) From (1), it is clear that if a, b, and c are three strings such that a commutes with b and c commutes with b, then a commutes with c.

(3) From (2), it is clear that if S_1 and S_2 are two nonempty sets of (nonempty) strings such that every two strings from S_1 commute and every two strings from S_2 commute, then every two strings from $S_1 \cup S_2$ commute if and only if there exist $a \in S_1^*$ and $b \in S_2^*$ such tht $a \neq e$, $b \neq e$ and a and b commute.

Proof of Lemma 4.1. It is sufficient to prove the result for an alphabet with two elements, say $\Sigma = \{0, 1\}$. Let us assume that there exist homomorphisms h_1 and h_2 such that $L_{\{0,1\}} = \mathrm{Eq}\,(h_1, h_2)$ and either h_1 or h_2 is nonerasing. We will write L for $L_{\{0,1\}}$.

Since $L \subseteq \{0, 1, \bar{0}, \bar{1}\}^*$, h_1 and h_2 must be defined on $\{0, 1, \bar{0}, \bar{1}\}$. Thus we can represent h_1 and h_2 by Table A.1.

TABLE A.1

	0	$\bar{0}$	1	$\bar{1}$
h_1	x_0	$\hat{x}_0$	x_1	$\hat{x}_1$
h_2	y_0	$\hat{y}_0$	y_1	$\hat{y}_1$

(i) For each $a \in \{0, 1, \bar{0}, \bar{1}\}$, $h_1(a) \neq h_2(a)$; otherwise, a is in $\mathrm{Eq}\,(h_1, h_2)$ and $a \notin \bigcup\{\text{shuffle}(w, \bar{w}) \mid w \in \{0, 1\}^*\} = L$, contradicting $L = \mathrm{Eq}\,(h_1, h_2)$.

(ii) Since $0\bar{0} \in L$ and $L = \mathrm{Eq}\,(h_1, h_2)$, $x_0\hat{x}_0 = h_1(0\bar{0}) = h_2(0\bar{0}) = y_0\hat{y}_0$. Either $|x_0| < |y_0|$ or $|x_0| > |y_0|$, since $|x_0| = |y_0|$ implies $x_0 = y_0$, contradicting (i). Assume that $|x_0| < |y_0|$ so that there exists $u_0 \neq e$ with $y_0 = x_0u_0$ and $\hat{x}_0 = u_0\hat{y}_0$. Since $00\bar{0}\bar{0} \in L$ and $L = \mathrm{Eq}\,(h_1, h_2)$, $x_0x_0\hat{x}_0\hat{x}_0 = h_1(00\bar{0}\bar{0}) = h_2(00\bar{0}\bar{0}) = y_0y_0\hat{y}_0\hat{y}_0$ so that $y_0 = x_0u_0$ and $\hat{x}_0 = u_0\hat{y}_0$

imply that $x_0x_0u_0\hat{y}_0u_0\hat{y}_0 = x_0u_0x_0u_0\hat{y}_0\hat{y}_0$. Hence $x_0u_0\hat{y}_0u_0 = u_0x_0u_0\hat{y}_0$ implying that $x_0u_0 = u_0x_0$ and $\hat{y}_0u_0 = u_0\hat{y}_0$. Thus any two strings from $\{x_0, \hat{x}_0, y_0, \hat{y}_0, u_0\}$ commute.

(iii) Since $1\bar{1} \in L$, an analysis similar to that of part (iii) shows that either there exists $u_1 \neq e$ with $y_1 = x_1u_1$ and $\hat{x}_1 = u_1\hat{y}_1$ and any two strings from $\{x_1, \hat{x}_1, y_1, \hat{y}_1, u_1\}$ commute, or there exists $v_1 \neq e$ with $x_1 = y_1v_1$ and $\hat{y}_1 = v_1\hat{x}_1$ and any two strings from $\{x_1, \hat{x}_1, y_1, \hat{y}_1, v_1\}$ commute. Thus we have two possible tables.

TABLE A.2

	0	$\bar{0}$	1	$\bar{1}$
h_1	x_0	$u_0\hat{y}_0$	x_1	$u_1\hat{y}_1$
h_2	x_0u_0	$\hat{y}_0$	x_1u_1	$\hat{y}_1$

TABLE A.3

	0	$\bar{0}$	1	$\bar{1}$
h_1	x_1	$u_0\hat{y}_0$	y_1v_1	$\hat{x}_1$
h_2	x_0u_0	$\hat{y}_0$	y_1	$v_1\hat{x}_1$

(iv) Consider the case of Table A.2. The string $01\bar{0}\bar{1}$ is in L so that $L = \text{Eq}(h_1, h_2)$ implies that $h_1(01\bar{0}\bar{1}) = h_2(01\bar{0}\bar{1})$. Using Table A.2, this means that $x_0x_1u_0\hat{y}_0u_1\hat{y}_1 = x_0u_0x_1u_1\hat{y}_0\hat{y}_1$ so that $x_1u_0\hat{y}_0u_1 = u_0x_1u_1\hat{y}_0$ and thus $x_1u_0 = u_0x_1$ and $\hat{y}_0u_1$ and $\hat{y}_0u_1 = u_1\hat{y}_0$. Hence every two strings from $S = \{x_0, \hat{x}_0, x_1, \hat{x}_1, y_0, \hat{y}_0, y_1, \hat{y}_1, u_0, u_1\}$ commute. Now consider the string $01\bar{1}\bar{0}$ which is not in L. Notice that $h_1(01\bar{1}\bar{0}) = x_0x_1u_1\hat{y}_1u_0\hat{y}_0 = u_0u_1x_0x_1\hat{y}_0\hat{y}_1$ by commutativity, and that $h_2(01\bar{1}\bar{0}) = x_0u_0x_1u_1\hat{y}_1\hat{y}_0 = u_0u_1x_0x_1\hat{y}_0y_1$ by commutativity. Thus $h_1(01\bar{1}\bar{0}) = h_2(01\bar{1}\bar{0})$ so that $01\bar{1}\bar{0}$ in is $\text{Eq}(h_1, h_2)$, contradicting the choice of h_1 and h_2 such that $\text{Eq}(h_1, h_2) = L$. Thus it cannot be the case that every two strings from the set S commute.

(v) The conclusion that any two strings from the set S commute was based on the fact that either x_1 and u_0 are nonempty or $\hat{y}_0$ and u_1 are nonempty. By choice of cases, u_0 is nonempty and u_1 is nonempty. Hence $x = h_1(1) = e$ and $\hat{y}_0 = h_2(\bar{0}) = e$, so that neither h_1 nor h_2 is nonerasing.

(vi) Consider the case of Table A.3. The string $01\bar{0}\bar{1}$ is in L so that $L = \text{Eq}(h_1, h_2)$ implies that $h_1(01\bar{0}\bar{1}) = h_2(01\bar{0}\bar{1})$. Using Table A.3 this means that $x_0y_1v_1u_0\hat{y}_0\hat{x}_1 = h_1(01\bar{0}\bar{1})h_2(01\bar{0}\bar{1}) = x_0u_0y_1\hat{y}_0v_1\hat{x}_1$ so that $y_1v_1u_0\hat{y}_0 = u_0y_1\hat{y}_0v_1$. Since $y_1v_1 = v_1y_1$, we have $v_1y_1u_0\hat{y}_0 = u_0y_1\hat{y}_0v_1$. Thus, either $u_0 = v_1$ or $u_0 = v_1w$ for some $w \neq e$ or $v_1 = u_0w$ for some $w \neq e$.

(a) Suppose $u_0 = v_1$. Since $u_0 \neq e$ and $v_1 \neq e$, $u_0v_1 = v_1u_0$ so that any two strings from $\{x_0, \hat{x}_0, x_1, \hat{x}_1, y_0, \hat{y}_0, y_1, \hat{y}_1, u_0, v_1\}$ commute and the analysis proceeds as in parts (iv) and (v).

(b) Suppose that $u_0 = v_1w$ for some $w \neq e$. The string $10\bar{1}\bar{0}$ is in L so that $L = \text{Eq}(h_1, h_2)$ implies that $h_1(10\bar{1}\bar{0}) = h_2(10\bar{1}\bar{0})$. Using Table A.3 this means that $y_1v_1x_0\hat{x}_1u_0\hat{y}_0 = y_1x_0u_0v_1\hat{x}_1\hat{y}_0$ so that $v_1x_0\hat{x}_1u_0 = x_0u_0v_1\hat{x}_1$. Since $u_0 = v_1w$, we have $v_1x_0\hat{x}_1u_0 = x_0v_1wv_1\hat{x}_1$ so that $v_1x_0 = x_0v_1$. Thus, either $x_0 \neq e$ so that any two strings from $\{x_0, \hat{x}_0, x_1, \hat{x}_1, y_0, \hat{y}_0, y_1, \hat{y}_1, u_0, v_1\}$ commute and the analysis proceeds as in parts (v) and (vi), or $x_0 = e$. If $x_0 = e$, then $v_1x_0\hat{x}_1u_0 = x_0u_0v_1\hat{x}_1$ implies $(v_1\hat{x}_1)u_0 = u_0(v_1\hat{x}_1)$ so that any two strings from $\{\hat{x}_0, x_1, \hat{x}_1, y_0, \hat{y}_0, y_1, u_0, v_1\}$ commute and the analysis proceeds as in parts (iv) and (v).

(c) Suppose that $v_1 = u_0 w$ for some $w \neq e$. As above, since $h_1(010\bar{1}) = h_2(010\bar{1})$, $y_1 v_1 u_0 \hat{y}_0 = u_0 y_1 \hat{y}_0 v_1$. Since $v_1 = u_0 w$, $y_1 u_0 w u_0 \hat{y}_0 = u_0 y_1 \hat{y}_0 v_1$ so that $y_1 u_0 = u_0 y_1$. Now the analysis is as in case (b), depending on whether $y_1 = e$ or $y_1 \neq e$.

(vii) In part (ii) it was assumed that $|x_0| < |y_0|$. The argument for the case of $|x_0| > |y_0|$ is exactly parallel to that given in parts (ii)–(vi). □

REFERENCES

[1] R. BOOK, *Polynomial space and transitive closure*, this Journal, 8 (1979), pp. 434–439.

[2] F.-J. BRANDENBURG, *Multiple equality sets and Post machines*, J. Comput. System Sci., to appear.

[3] K. CULIK, II, *A purely homomorphic characterization of recursively enumerable sets*, J. Assoc. Comput. Mach., 26 (1979), pp. 345–350.

[4] ———, *On homomorphic characterization of families of languages*, Automata, Languages, and Programming, Lecture Notes in Computer Science 71, Springer-Verlag, New York, 1979, pp. 161–170.

[5] K. CULÍK II AND H. A. MAURER, *On simple representations of language families*, RAIRO Informat. Théor., 13 (1979), pp. 241–250.

[6] K. CULIK II AND A. SALOMAA, *On the decidability of homomorphism equivalence for languages*, J. Comput. System Sci., 17 (1978), pp. 163–175.

[7] J. ENGELFRIET AND G. ROZENBERG, *Fixed point languages, equality languages, and representation of recursively enumerable languages*, J. Assoc. Comput. Mach., to appear. An extended abstract appears in the Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 123–126.

[8] J. HOPCROFT AND J. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

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

[10] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[11] A. SALOMAA, *Equality sets for homomorphisms of free monoids*, Acta Cybernet., 4 (1978), pp. 127–139.

[12] R. SMULLYAN, *Theory of Formal Systems*, Annals of Mathematics Studies, No. 47, Princeton University Press, Princeton, NJ, 1961.

[13] R. STEARNS, J. HARTMANIS AND P. LEWIS, *Hierarchies for memory limited computations*, Conference Record, 6th IEEE Symp. on Switching Circuit Theory and Logical Design, 1965, pp. 179–190.

[14] C. WRATHALL, *Remarks on languages and relations*, in preparation.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0009 $01.00/0

FINDING CONNECTED COMPONENTS AND CONNECTED ONES ON A MESH-CONNECTED PARALLEL COMPUTER*

DAVID NASSIMI† AND SARTAJ SAHNI‡

Abstract. Let $G=(V, E)$ be an undirected graph in which no vertex has degree more than d. Let $|V|=n^q=2^p$. In this paper we present an $O(q^3(q+d)n \log n)$ algorithm to find the connected components of G on a q-dimensional $n \times n \times \cdots \times n$ mesh-connected parallel computer. When $d=2$, the connected components can be found in $O(q^4 n)$ time. We also show that the connected ones problem can be solved in $O(q^6 n)$ time.

Key words. connected components, connected ones, mesh-connected computer, parallel algorithm, complexity

1. Introduction. A *mesh-connected parallel computer* (MCC) is an SIMD (Single Instruction Stream, Multiple Data Stream) computer. It consists of $N=2^p$ processing elements (PEs). In a q-dimensional $n \times n \times \cdots \times n$ MCC, the PEs may be thought of as logically arranged in a q-dimensional $n \times n \times \cdots \times n$ array. The PE at location $(i_{q-1}, \cdots, i_0)$ of the array is *connected* to the PEs at locations $(i_{q-1}, \cdots, i_j \pm 1, \cdots, i_0)$ $0 \leqq j < q$, provided they exist. Two PEs are *adjacent* iff they are connected. A PE may transmit data to an adjacent PE in a *unit-route*. In an MCC, each PE has some local memory. Each word and register of local memory has a storage capacity of $\log N$ bits (all logarithms in this paper are base 2). The PEs are synchronized and operate under the control of a single instruction stream which is determined by the *control unit.* An *enable mask* may be used to select a subset of PEs that will perform the instruction to be executed at any given time. All enabled PEs perform the same instruction.

Parallel algorithms for MCCs and closely related models (such as cellular automatons and parallel processing arrays) have been studied by several researchers. Efficient sorting algorithms can be found in [11] and [15]. Algorithms for certain graph and matrix problems appear in [1] to [4], [8] to [10], and [16]. General routing problems are considered in [12], [13], and [14]. [6] and [7] cover language recognition type problems. Several other references to work on MCCs exist. Most of these can be found by tracing through the references of the papers cited above.

In this paper we shall address the following problems:

(i) Let $G=(V, E)$ be an undirected graph with $N=n^q=2^p$ vertices. Let the degree of each vertex be at most d. Find the connected components of G. We shall call this problem: *connected components for degree d graphs.* The initial configuration for this problem has the adjacency list for vertex i, $A(i, 0\colon d-1)$, stored in d registers of PE(i).

(ii) *Connected components for degree* 2 *graphs*: This is a special case of (i) (i.e. with $d=2$). It arises in the design of a parallel algorithm to set-up the Benes permutation network (Nassimi and Sahni [14]).

(iii) *Connected ones*: In this problem each PE has a register called A. Initially $A(i)=1$ or 0, $0 \leqq i < N$. Two ones are said to be *adjacent* if they are in the A registers of two adjacent PEs. The transitive closure, R*, of this adjacency relation defines the connectivity of the ones. A one in PE(i) is connected to a one in PE(j) iff $R^*(i, j)=1$. The connected ones problem is to determine whether all ones are connected.

Hirschberg [5] has obtained an $O(\log^2 N)$ parallel algorithm to find the connected components of any N vertex undirected graph using an SIMD machine with N^2PEs.

* Received by the editors October 5, 1979, and in final form February 20, 1980. This research was supported in part by the National Science Foundation under grant MCS 78-15455.

† Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

The parallel machine model used by him is different from the one we are using. In his model, all PEs share a common memory. So, data transfers between PEs may be made in $O(1)$ time via this memory. We obtain an $O(q^3(q+d)n \log n)$ algorithm to find the connected components of an n^q-vertex graph on an n^q PE MCC. Our algorithm can, however, be used only on graphs with vertex degree at most d. The basic idea behind our algorithm is quite similar to the idea underlying Hirschberg's algorithm. The implementation of this scheme on an MCC requires the use of novel strategies. Our algorithm for problem (ii) runs in $O(q^4n)$ time.

The connected ones problem has been studied earlier by several researchers ([1], [8], and [9]). The problem is of importance in pattern recognition. The parallel computer model used in [1], [8], and [9] is called a parallel processing array (PPA). The essential difference between a PPA and an MCC is that each PE in a PPA is a *finite state automaton* capable of storing only a fixed number of bits of information. Each PE in an MCC can store $O(\log N)$ bits of information. While other differences between the two models exist, these are not so crucial. For the PPA model, Levialdi [9] has an $O(n)$ algorithm to solve the connected ones problem for an $n \times n$ PPA. This algorithm can be easily run on an $n \times n$ MCC in $O(n)$ time. Kosaraju [8] has extended this result for the case of $n \times n \times d$ PPAs. Kosaraju's algorithm solves the connected ones problem for $n \times n \times d$ PPAs in $O(n)$ time. It is not known if the connected ones problem for $n \times n \times n$ PPAs can be solved in $O(n)$ time. We are able to solve the connected ones problem for q-dimensional $n \times n \times \cdots \times n$ MCCs in $O(q^6n)$ time. The underlying idea behind our algorithm is the same as that for our algorithm for problems (i) and (ii).

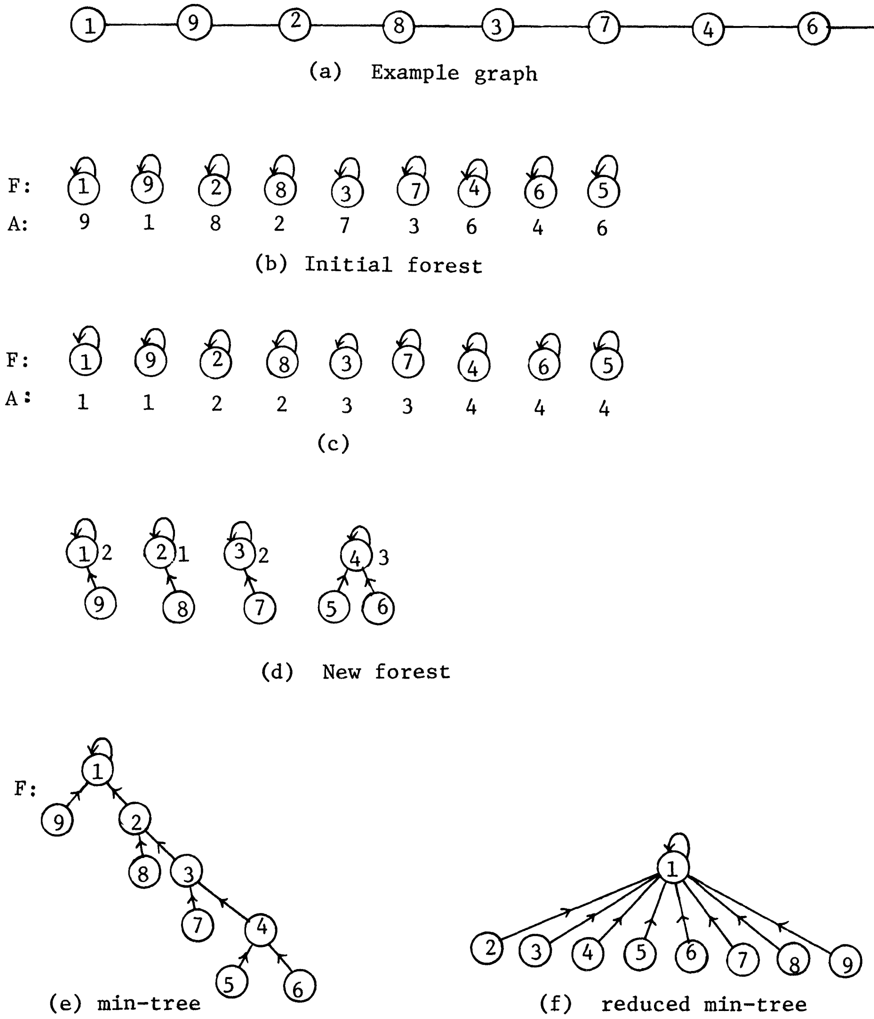
In § 2 we discuss the solution strategy to be used in obtaining our algorithms for the three problems cited above. In § 3 we introduce the notation and terminology to be used in later sections. In this section we also introduce the subalgorithms that have been developed in [13] and [15] and which will be used to arrive at the algorithms of this paper. Section 3 also contains a new MCC algorithm. This algorithm generates reduced min-trees (this term will be defined in § 2). Sections 4, 5, and 6 respectively contain the algorithms for problems (i), (ii) and (iii).

2. Solution strategy.

The solution strategy used to arrive at the algorithms of §§ 4, 5 and 6 is quite similar to that used by Hirschberg [5]. First, let us define two terms. A *min-tree* is a tree in which the index of each node is less than the index of each of its children. A *reduced min-tree* is a min-tree of height at most two. (The height of a tree is the maximum level in the tree. The root is at level 1.) The algorithms developed in this paper will partition the set of vertices in a graph (or the set of ones) into a set of reduced min-trees such that two vertices (or two ones) will be in the same reduced min-tree iff they are in the same connected component (or in the same set of connected ones).

While finding the connected components of a graph, we shall maintain several sets of vertices. These sets will represent a partition of the vertex set of the given graph, G. All vertices in the same set will be in the same connected component of G. Each set will be represented as a reduced min-tree with $R(i)$ pointing to the root of the tree. The basic strategy in our connected components algorithm is to combine together sets of vertices while retaining the property that all vertices in the same set are in the same component of G. This is continued until no more set combination is possible. It is a trivial matter to see that at this point, each set of vertices defines a connected component of G.

We illustrate the preceding strategy by an example. Consider the 9-vertex graph of Fig. 1(a). Initially, no two vertices are known to be in the same component. So, we begin with 9 sets of vertices. These are given by the 9-tree forest of Fig. 1(b). The arrows give



(a) Example graph

(b) Initial forest

(c)

(d) New forest

(e) min-tree

(f) reduced min-tree

FIG. 1

the R values. Observe that $R(i) = i$ for all nodes (as stated earlier, $R(i)$ gives the root of the tree). Two trees with roots i and j, $i \neq j$, are *adjacent* iff tree i contains a node i' and tree j contains a node j' such that i' and j' are adjacent in G. The tree with root j is the *min-adjacent tree* to the tree with root i iff the tree with root i has no adjacent tree with root less than j. To determine whether two sets should be combined, we first determine the min-adjacent tree $A(i)$ for each root i. The A values are given below each root of Fig. 1(b). As can be seen, for some i, $A(i) > i$. This is undesirable, since changing $R(i)$ to $A(i)$ to effect a set combination would result in nontrees (i.e. graphs with cycles). This is remedied by changing all $R(i)$ with $A(i) > i$ to $A(A(i))$.

LEMMA 1. $A(A(i)) \leqq i$.

Proof. Let $A(i) = j$. Since trees i and j are adjacent, $A(j) \leqslant i$. □

From Lemma 1 it follows that if we replace $R(i)$ by $A(i)$ if $A(i) \leqq i$, and by $A(A(i))$ otherwise, then we will be combining together min-adjacent trees and will be left behind with min-trees. Fig. 1(c) gives the updated A values and Figure 1(d) shows the min-trees resulting from the combination just described. All min-trees of Fig. 1(d) are also reduced min-trees. Repeating the above combination process, we first find that

$A(1)=2$, $A(2)=1$, $A(3)=2$ and $A(4)=3$. Updating $A(i)$ for $A(i)>i$, we get $A(1)=1$. The resulting min-tree is given in Fig. 1(e). The corresponding reduced min-tree is given in Fig. 1(f). The next lemma shows that the combination process described above need be repeated at most $\log N$ times for an N-vertex graph.

LEMMA 2. *If the vertices of a connected component are in r distinct trees, $r>1$, then following a tree combination as described above, they are in at most $\lfloor r/2 \rfloor$ trees.*

Proof. Let the roots before the combination be $I=\{i_1, i_2, \cdots, i_r\}$. Let $A(i)$, $i \in I$, be the root of the min-adjacent tree for root i. And, let $I=I^G U I^L$ where $I^G=\{i|i \in I$ and $A(i)>i\}$, and $I^L=\{i|i \in I$ and $A(i)<i\}$. The only candidates for root nodes following the combination are nodes I^G (since for any node $i \in I^L$, $R(i)$ is changed to $A(i)<i$). For a node $i \in I^G$, $R(i)$ becomes $A(A(i))=A(j)$, where $j=A(i)$ and $A(j) \leqq i<j$. Thus, the number of new roots, r', is also no more than the number of nodes j of I with $A(j)<j$. That is, $r' \leqq |I^L|$. And since $r' \leqq |I^G|$, we conclude $r' \leqq \lfloor r/2 \rfloor$. □

3. Terminology, notation and subalgorithms. Throughout, we shall assume that we are dealing with an $n \times n \times n \cdots \times n$ q-dimensional MCC with $N=n^q=2^p$ PEs. For any integer i, $(i)_b$ will denote bit b of i, and $(i)_{s:l}$, $s \geqq l$, will denote the number with binary representation $(i)_s \cdots (i)_l$. Bit 0 is the least significant bit. By a 2^k-*block* of PEs we shall mean a block of 2^k consecutively indexed PEs whose indices differ only in the least significant k bits. We shall index the PEs in a q-dimensional MCC in *shuffled row-major order* (see [13] or [15]). When this indexing scheme is used, the index of the PE in position $(i_{q-1}, \cdots, i_0)$ of the q-dimensional PE array is obtained by merging together the binary representations of $i_{q-1}, \cdots, i_0$. Let $t=\log n$. When shuffled row-major indexing is used, the binary representation of the index of the PE in position $(i_{q-1}, i_{q-2}, \cdots, i_0)$ is $(i_{q-1})_{t-1}(i_{q-2})_{t-1} \cdots (i_0)_{t-1}(i_{q-1})_{t-2}(i_{q-2})_{t-2} \cdots (i_0)_{t-2} \cdots (i_{q-1})_0(i_{q-2})_0 \cdots (i_0)_0$. (Recall that $(i_{q-1})_{t-1}$ denotes bit $t-1$ of i_{q-1}.) Fig. 2 gives the PE indices resulting when the shuffled row-major indexing scheme is used on a 4×4 MCC.

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

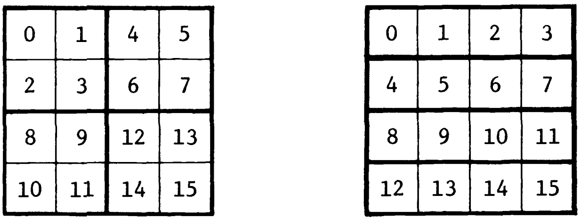
FIG. 2. *Shuffled row-major indexing.*

If we have a q-dimensional MCC with $N=n^q=2^p$ PEs, then each 2^{p-1}-block of PEs will form an $n/2 \times n \times n \times \cdots \times n$ array, each 2^{p-2}-block will form an $n/2 \times n/2 \times n \times \cdots \times n$ array, and each 2^{p-q}-block will form an $n/2 \times n/2 \cdots \times n/2$ array when shuffled row-major indexing is used. In general, a 2^k-block forms an $m_{q-1} \times m_{q-2} \times \cdots \times m_0$ array, where

$$(1) \qquad m_i=\begin{cases} 2^{\lceil k/q \rceil}, & 0 \leqq i<d, \; d=k \textbf{ mod } q, \\ 2^{\lfloor k/q \rfloor}, & d \leqq i<q. \end{cases}$$

To see this, note that the most significant bit of a PE index comes from dimension $q-1$, the next from dimension $q-2$, and so on. In general, bit i of a PE index is the $\lfloor i/q \rfloor$th bit of dimension i **mod** q.

In words, (1) states that when shuffled row-major indexing is used, then each 2^k-block forms an $m_{q-1} \times m_{q-2} \times \cdots \times m_0$ array such that $\sum_{i=0}^{q-1} m_i$ is minimized. As an example, consider 2^2-blocks for a 4×4 MCC using row-major and shuffled row-major indexing (Fig. 3). The quantity m_0+m_1 is 4 for the shuffled row-major indexing but 5 for the row-major indexing.



0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

(a) Shuffled row-major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(b) Row-major

FIG 3. *2^2-blocks.*

Our algorithms will make use of two kinds of assignment statements. The first kind is an assignment requiring no data routing. This will be denoted by the use of the symbol ':='. The second kind will require data routing among the PEs and will be denoted by '$\leftarrow$'. PE selectivity will be done by providing a masking function. For example, the statement

$$X(i) := Y(i), ((i)_b = 0)$$

has selectivity function $(i)_b = 0$, and the assignment $X := Y$ is to be carried out only on PEs with bit $b = 0$. When a selectivity function is provided for a **for** loop, the instructions in the **for** loop body are to be performed only on PEs satisfying the selectivity function.

The discussion of the next three sections will make use of the following algorithms.

(i) SORT.

This algorithm sorts data items $G(i)$, $0 \leqq i < N = n^q$, into nondecreasing order assuming a shuffled row-major indexing scheme. Its complexity is $O(q^2 n)$ (see [15]).

(ii) RAR (Random Access Read).

In an RAR, each PE specifies a PE index from which it wishes to receive data. The RAR algorithm described in [13] routes data from the source PEs to the PEs desiring to receive the data. Each PE is allowed to request data from at most one PE. Several PEs can request data from the same PE. The algorithm of [13] has complexity $O(q^2 n)$. An RAR in a 2^k-block takes $O(q^2 2^{\lceil k/q \rceil})$ time.

(iii) RAW (Random Access Write).

In an RAW, each PE specifies the PE to which its data is to be sent. If two or more PEs specify the same destination PE, then the RAW algorithm of [13] can be set either to send all pieces of data to the destination or to send a selected one (say, the one with minimum key). In the former case, if data from at most d PEs is to go to one PE, then the time needed is $O(qn(q+d))$. In the latter case, the time needed is $O(q^2 n)$. An RAW in a 2^k-block takes $O(q 2^{\lceil k/q \rceil}(q+d))$ or $O(q^2 2^{\lceil k/q \rceil})$ time, respectively.

(iv) RANK.

The rank of a selected record in a PE is the number of PEs of smaller index which contain a selected record. For example, assume we have 8 PEs each containing one record. Let the key values of these 8 records be (6, 4, 2, 2*, 6, 6*, 3*, 4*) where an asterisk over a key value denotes a flagged or selected record. The ranks of the flagged records are (–, –, –, 0, –, 1, 2, 3). Nassimi and Sahni [13] present an algorithm to rank records in each 2^k-block of PEs. Let $s = \lfloor k/q \rfloor$, $d = k$ **mod** q, and let $m_i s$ be as given in (1). The number of unit routes needed by the ranking algorithm of [13] is

$2\sum_{i=0}^{q-1}(m_i-1)=(q+d)2^{s+1}-2q=O(q2^{\lceil k/q\rceil})$. When $k=\log N=q\log n$, the number of unit-routes becomes $2q(n-1)$.

(v) CONCENTRATE.

Let $G(i_r)$, $0\leqq r\leqq j$, be a set of records with $G(i_r)$ initially in PE(i_r). Assume that the records have been ranked so that the rank $H(i_r)$ of record $G(i_r)$ is r. A concentrate results in record $G(i_r)$ being moved to PE(r), $0\leqq r\leqq j$. The algorithm given in [13] carries out this function using the same number of unit-routes as used by the ranking algorithm. This algorithm permutes records so that no record is destroyed.

(vi) REDUCE.

Let $R(0{:}\,N-1)$ define a set of *min-trees* (see § 2). $R(i)=i$ iff i is a root of a min-tree and $R(i)<i$ otherwise. Procedure REDUCE(k) generates the set of reduced min-trees when each of the original min-trees is confined to a 2^k-block of PEs. A 2^k-block may contain more than one min-tree. Note that $R(i)$ denotes a register in PE(i). In line 3, we assume that the condition $((R(i))_{k-1:b}=(i)_{k-1:b})$ will be true for all i when $b>k-1$.

```
line    procedure REDUCE($k$)
1         global $R$
2         for $b$ := 1 to $k$ do
3           $R(i)\leftarrow R(R(i)),\ ((R(i))_{k-1:b}=(i)_{k-1:b})$
4         end
5       end REDUCE
```

ALGORITHM 1

The correctness of REDUCE may be established by induction on b. We shall show that following iteration $b=r$ of the **for** loop, the following condition holds:

C^r: For any i, $R(i)=j\leqq i$. Furthermore, either j is a root,
or i and j are in different 2^r-blocks.

Observe that initially C^0 holds. For the induction step, assume that C^{r-1} holds following iteration $b=r-1$. During iteration $b=r$, PE(i) is enabled to update its $R(i)$ iff i and $R(i)=j$ are in the same 2^r-block. If j is a root, then the update does not alter $R(i)$. If j is not a root, then from C^{r-1}, it follows that i and j are in different 2^{r-1}-blocks such that $(i)_{r-1}=1$ and $(j)_{r-1}=0$. Let $R(j)=l$. Following the update, we get $R(i)=l<i$. If l is not a root, then it must be in a lower 2^{r-1}-block with respect to j; this means that i and l are in different 2^r-blocks. Hence, C^r will hold after iteration $b=r$. The correctness of REDUCE(k) will follow from C^k since each of the original min-trees was restricted to a 2^k-block.

Line 3 of REDUCE is an RAR in a 2^b-block, and requires $O(q^2 2^{\lceil b/q\rceil})$ time. The complexity of REDUCE(k) is therefore $O(q^3 2^{k/q})$. When $k=p=\log N$, the complexity becomes $O(q^3 n)$.

4. Connected components for degree *d* graphs. Let G be a graph with 2^k vertices. No vertex has degree more than d. Let ADJ(i,j), $0\leqq j<d$, be the adjacency list for vertex i, $0\leqq i<2^k$. If vertex i has degree $d_i<d$ then we assume that ADJ$(i,j)=\infty$, $d_i\leqq j<d$. We also assume that ADJ(i,j), $0\leqq j<d$, denotes memory cells/registers associated with PE(i), $0\leqq i<2^k$. A shuffled row-major indexing of PEs is assumed. The graph resides in a 2^k-block of PEs.

Procedure CONNECT(k,d) is a direct implementation of the strategy outlined in § 2. Line 1 initializes the vertex sets to contain one vertex each. Each set is represented as a min-tree. The **for** loop of lines 2–14 iterates the set combination process $k=\log 2^k$

times. Lines 3–13 implement the set combination process. In lines 3–9 we determine for each vertex j, its min-adjacent tree. This is defined as:

$$\text{CANDID}(j) = \mathbf{min}\{R(t) | t = \text{ADJ}(j, l) \text{ for some } l \text{ and } R(t) \neq R(j)\}.$$

Line 10 then finds the min-adjacent tree for each root i. Line 11 takes care of trees that have no adjacent trees, and line 12 updates R according to Lemma 1. From the discussion of § 2, it follows that after line 12, R defines a set of min-trees. Line 13 produces a reduced min-tree from each tree.

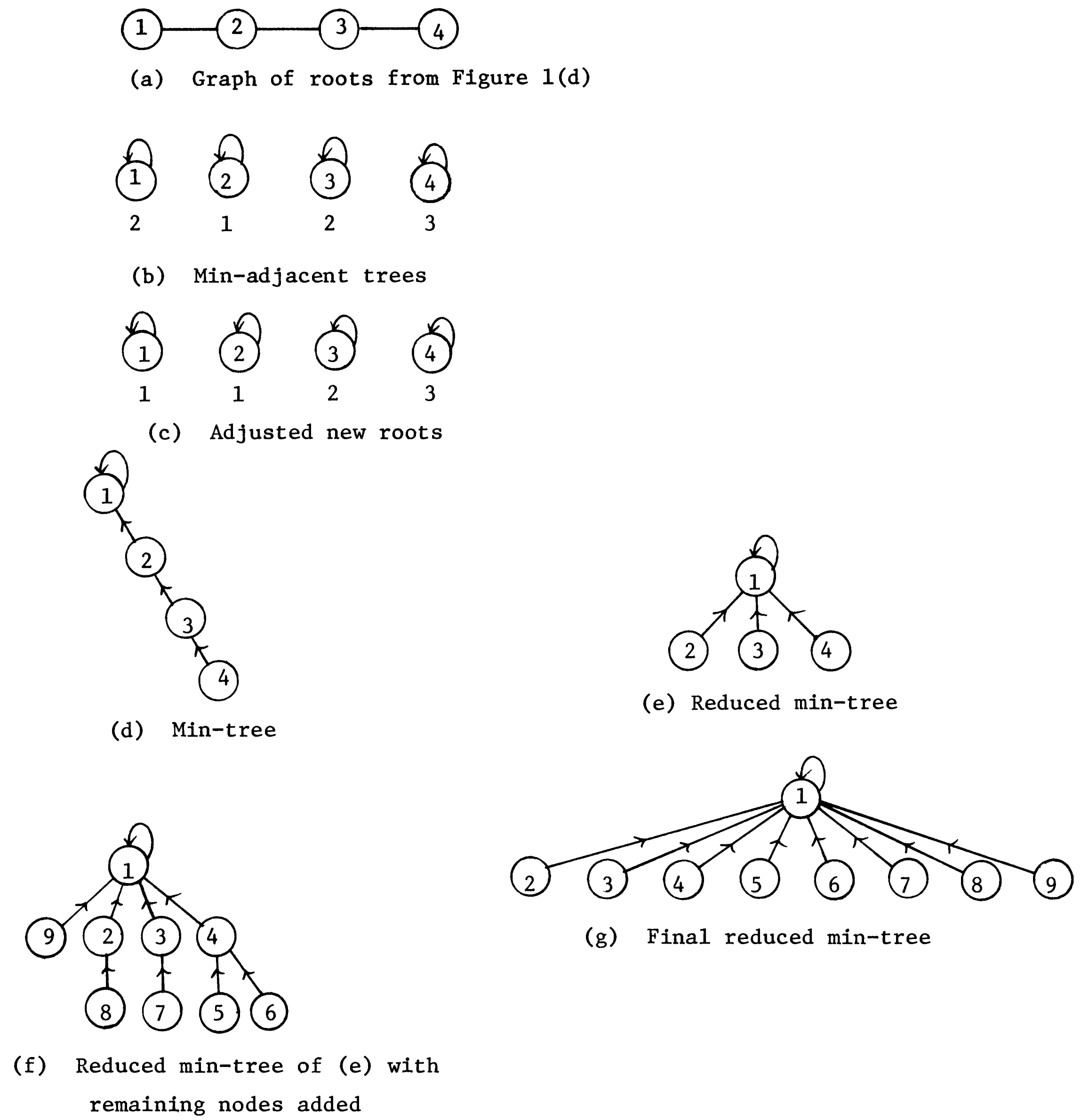
As far as the complexity of CONNECT is concerned, we observe that lines 6 and 12 require RARs while line 10 is an RAW. Hence, each of these three lines requires $O(q^2 2^{k/q})$ time. Line 13 requires $O(q^3 2^{k/q})$ time. The overall complexity of CONNECT is therefore $O(k(q^3 2^{k/q} + dq^2 2^{k/q})) = O(kq^2(q+d)2^{k/q})$. Note that in this much time we can find the connected components of several 2^k-vertex graphs by using each 2^k-block of PEs for a different graph. Also note that when $k = \log N$, the complexity of CONNECT becomes $O(q^2(q+d)N^{1/q} \log N) = O(q^3(q+d)n \log n)$.

```
line   procedure CONNECT($k$, $d$)
         //ADJ($i$, 0: $d-1$) gives the adjacency list for PE($i$).//
         //$d$ is the degree of the graph. $2^k$ is the number of PEs//
 1       $R(i)$ := $i$//start with single-node trees//
 2       for $b$ := 0 to $k-1$ do //merge trees//
 3         CANDID($j$) := $\infty$
 4         for $e$ := 0 to $d-1$ do //get smallest neighboring root//
 5           TEMP($j$) := $\infty$
 6           TEMP($j$) ← $R$(ADJ($j$, $e$)) //get root from neighbor//
 7           TEMP($j$) := $\infty$, (TEMP($j$) = $R(j)$) //discard if your own root//
 8           CANDID($j$) := min{CANDID($j$), TEMP($j$)}
 9         end
           //find min-adjacent tree//
10         $R(i)$ ← min{CANDID($j$)|$R(j) = i$}
11         $R(i)$ := $i$, ($R(i) = \infty$) //this tree cannot grow//
12         $R(i)$ ← $R(R(i))$, ($R(i) > i$) //convert to min-tree//
13         call REDUCE($k$) //reduce min-tree//
14       end
15     end CONNECT
```

ALGORITHM 2.

5. Connected components for degree 2 graphs. As stated earlier, this special case arises in setting up the Benes permutation network [14]. We shall show that this special case can be solved in $O(q^4 n)$ time on a q-dimensional MCC with $N = n^q$ PEs. As in the previous section we begin with the adjacency list of vertex i in PE(i). The indices of the (at most) two vertices adjacent to vertex i are stored in ADJ(i, 0) and ADJ(i, 1). If vertex i is of degree 1 then ADJ(i, 1) = ∞; if the vertex has degree 0 then ADJ(i, 0) = ADJ(i, 1) = ∞. Our connected components algorithm will produce reduced min-trees with the property that $R(i) = R(j)$ iff i and j are in the same connected component.

Our algorithm for degree 2 graphs uses a strategy slightly different from that described in § 2. Consider the example graph of Fig. 1(a). As before, we begin with each vertex in a set of its own. Vertices in a set are represented by a min-tree. Fig. 1(b) gives the initial forest denoting the sets. Each tree determines its min-adjacent tree. These



(a) Graph of roots from Figure 1(d)

(b) Min-adjacent trees

(c) Adjusted new roots

(d) Min-tree

(e) Reduced min-tree

(f) Reduced min-tree of (e) with remaining nodes added

(g) Final reduced min-tree

FIG. 4

are given outside the nodes in Fig. 1(b). Fig. 1(d) depicts the min-trees following a tree combination step making use of Lemma 1. At this point, rather than repeat the tree-combination step and get the min-tree of Fig. 1(e), we form a new graph which contains only the root nodes of Fig. 1(d), i.e., nodes 1, 2, 3, and 4. This graph is called the *root graph.* In the new root graph, vertices i and j are adjacent iff the trees with roots i and j are adjacent. The resulting graph is given in Figure 4(a). Note that, in general, if vertices $i_1, i_2, \cdots, i_t$ are in the same min-tree following the tree-combination step, then at most two of the vertices $i_1, i_2, \cdots, i_t$ can be adjacent to vertices not in the tree. To see this, observe that the graph we began with was of degree 2. Each vertex in a min-tree is adjacent to at least one other vertex in the tree (provided the tree has more than one vertex). If every vertex of $i_1, i_2, \cdots, i_t$ is adjacent to two other vertices in the same min-tree, then none of $i_1, i_2, \cdots, i_t$ can be adjacent to a vertex not in the min-tree. So, assume there is a vertex (say i_1) which is adjacent to exactly one of $i_2, \cdots, i_t$. Without loss of generality, we may assume this vertex to be i_2. i_2 must be adjacent to one of $i_3, \cdots, i_t$ otherwise $i_3, \cdots, i_t$ cannot be in the same min-tree as i_1 and i_2. Repeating this argument, we see that the vertices in a tree can be relabeled so that i_j is adjacent to i_{j+1}, $1 \leqq j < t$. So vertices i_1 and i_t are the only vertices that can be adjacent to a vertex not in the tree. Let the root of this tree be i_r, and let u and v be the two possible vertices adjacent to i_1 and i_t and not in this tree. Let $R(u)$ and $R(v)$ be the roots of the trees containing u and v respectively. In the new graph of root nodes, only $R(u)$ and $R(v)$ can be adjacent to vertex i_r. So, the new root graph is also of degree 2.

The tree combination step of § 2 is repeated on the root graph of Fig. 4(a). This yields Fig. 4(b), 4(c) and 4(d). The min-tree of Fig. 4(d) is reduced to get Fig. 4(e). At this point, if more than one reduced min-tree existed, we would form a new root graph and re-apply the tree combination step. The tree combination-root graph formation process has to be repeated at most log N times if we start with an N node graph (see Lemma 2). Our algorithm will repeat this basic step exactly log N times. Once we are finished with this step, we will be left with one min-tree for each component (as in Fig. 4(f)). The min-tree for each component is then reduced as in Fig. (4(g).

```
line   procedure CONNECT2($p$)
          //ADJ($i$, 0: 1) gives the adjacency list for PE($i$)//
          //$2^p = N$ is the number of PEs//
 1        ORG($i$) := $R(i)$ := $i$ //original address of a node//
 2        LIVE($i$) := 1
 3        LIVE($i$) := 0, (ADJ($i$, 0) = ADJ($i$, 1) = $\infty$)
 4        $b$ := $p$ //initial graph is in a $2^p$-block//
 5        loop, (LIVE($i$) = 1 and $i < 2^b$) //active set of PEs//
 6           $R(i)$ := min{ADJ($i$, 0), ADJ($i$, 1)}
 7           $R(i) \leftarrow R(R(i))$, ($R(i) > i$)' //convert to min-trees//
 8           call REDUCE($b$)
 9           if $b = 1$ then[$R(i) \leftarrow ORG(R(i))$
10              exit from loop] // go to line 25//
11           for $e$ := 0, 1 do
12              ADJ($i$, $e$) $\leftarrow$ $R$(ADJ($i$, $e$))
13              ADJ($i$, $e$) := $\infty$, (ADJ($i$, $e$) = $R(i)$)
14           end
15           CANDID($i$) := min{ADJ($i$, 0), ADJ($i$, 1)}
16           {ADJ($i$, 0), ADJ($i$, 1)} $\leftarrow$ {CANDID($j$)|$R(j) = i$}
17           $R(i) \leftarrow$ ORG($R(i)$) //cut off all nodes from roots//
18           LIVE($i$) := 0, ($R(i) \neq$ ORG($i$) or ADJ($i$, 0) = ADJ($i$, 1) = $\infty$)
             //update adjacency lists of live nodes before moving://
19           call RANK($b$) //rank live nodes. Let $H$ be the rank.//
20           ADJ($i$, 0) $\leftarrow$ $H$(ADJ($i$, 0))
21           ADJ($i$, 1) $\leftarrow$ $H$(ADJ($i$, 1))
22           call CONCENTRATE($b$) //concentrate live nodes. The records//
                // are $G$ = (ORG, LIVE, ADJ, R)//
23           $b$ := $b - 1$
24        repeat //go to line 5//
25        $R$(ORG(i)) $\leftarrow$ R(i) //send all nodes back to origin; use SORT.//
26        call REDUCE($p$)
27     end CONNECT2
```

ALGORITHM 3

The reason the addition of the root graph formation step leads to an asymptotically faster algorithm than procedure CONNECT (Algorithm 2) is that a root graph cannot contain more than $N/2$ nodes of degree one or two if the original graph contained N nodes. This follows from Lemma 2. As a result, the root graph can be concentrated into a 2^{p-1}-block of PEs before tree combination. The next root graph formed will have at most $N/4$ nodes and so can be concentrated into a 2^{p-2}-block of PEs. Thus, following each tree combination step, we can localize the next root graph to a smaller block of PEs. Tree combination in a smaller block of PEs takes less time than in a bigger block and so the resulting algorithm is faster.

Now, let us look at the details of the algorithm. Since our algorithm will be concentrating root graphs into smaller blocks of PEs, we will have a need to know the originating PE for each vertex in a root graph. ORG(i) will be used to denote the originating PE for the vertex currently in PE(i). The variable LIVE will be used to distinguish between nodes in the current root graph and other nodes: LIVE$(i) = 1$ iff the vertex currently in PE(i) is in the root graph and has degree more than zero; LIVE$(i) = 0$ for all other PEs. We may regard the initial graph as a root graph. Procedure CONNECT2 (Algorithm 3) is a formal specification of our algorithm. Lines 1 to 3 initialize ORG, R, and LIVE. The variable b is used to denote the current block size. It is initialized to p (i.e. the block size is $2^p = N$) in line 4. Lines 5 to 24 define the basic tree combination-root graph formation step. The PE selectivity function specified in line 5 requires that the statements within the loop body be executed only on "live" PEs that are in the "first" 2^b-block (the "first" 2^b-block contains PEs $0, 1, \cdots, 2^b - 1$). Additional selectivity functions provided within the loop further restrict the PEs on which certain statements are to be executed.

At the start of each iteration of the loop of lines 5–24, we have a new root graph. Each vertex in this graph is in a different set. So, to find the min-adjacent tree for any single-node tree T, we need only find the least indexed vertex adjacent to the sole vertex in T. This is done in line 6. Line 7 updates R according to Lemma 2. The min-trees created in lines 6 and 7 are reduced in line 8. If $b = 1$ then only two vertices could be present in the root graph. So, only one min-tree can result following lines 6 and 7. Hence, no further iterations are needed, and the loop is exited from line 10. If $b \neq 1$, then further iterations of the tree combination step may be needed. So, we proceed to set up the new root graph. In lines 11 to 15 each live vertex determines whether it is adjacent to another live vertex in a different min-tree. Following line 15, CANDID$(i) \neq \infty$ iff the vertex in PE(i) is adjacent to a live vertex in a different tree. If CANDID$(i) \neq \infty$, then CANDID(i) equals the index of the PE containing the adjacent live vertex's root. In line 16 the (at most) two vertices that will be adjacent to root i in the new root graph are recorded in ADJ$(i, 0)$ and ADJ$(i, 1)$. Note that one or both of these values may be ∞. At this point the R value of each live PE(i) is updated to be the originating PE of the vertex in PE$(R(i))$ (line 17). Following this, $R(i) = $ ORG(i) only for root nodes. Line 18 "kills" nodes that are not to be in the new root graph as well as nodes that would have a degree of zero in the new root graph.

Following the creation of the new root graph, the new root graph is to be concentrated into a smaller block of PEs. Procedure RANK ranks all the nodes in the new root graph. The rank, H, of a node gives the PE to which it is to be routed during the concentration. ADJ$(i, 0:1)$ is updated in lines 20 and 21 to reflect the PE indices of the adjacent nodes following the concentration. Line 22 actually concentrates the nodes in the new root graph. The records being concentrated are $\langle$ORG(i), LIVE(i), ADJ$(i, 0{:}\ 1)$, $R(i)\rangle$ for LIVE$(i) = 1$ and $i < 2^b$. Note that if there are j live PEs, $j \leqq 2^{b-1}$, then following a concentration the live records occupy PEs $0, 1, \cdots, j-1$ and the remaining records are in PEs $j, \cdots, N-1$. No record is destroyed during concentration. Also, CONCENTRATE *does not* change the relative order of live records since for two live records i and j, $H(i) > H(j)$ iff $i > j$. As a result of this, we can use ADJ(i, 0: 1) rather than ORG(ADJ(i, 0)) and ORG(ADJ(i, 1)) when finding min-adjacent trees (line 6). Note that for $R(i)$ as given in line 6, we have ORG($R(i)$) = **min**{ORG(ADJ(i, 0)), ORG(ADJ(i, 1))}; assume ORG(∞) = ∞. Thus, in lines 6, 7, 12, 13, 15, 16, 20, 21 R and ADJ are really PE indexes and not vertex indexes. In lines 9 and 17 R is reset to be a vertex index. So, on exit from the loop, all R values are vertex indexes. Line 25 sends every vertex back to its originating PE. It is easy to see that following this, we shall have $R(i) \leqq i$ for every i, $0 \leqq i < N$. The min-trees are finally reduced in line 26.

To obtain the complexity of procedure CONNECT2, we need be concerned only with lines 7, 8, 9, 12, 16, 17, 19, 20, 21, 22, 25 and 26. The remaining lines contribute a total of $O(p)$ time. Lines 7, 9, 12, 17, 20 and 21 are RARs in a 2^b-block. Line 16 is an RAW in a 2^b-block. During this RAW, only the smallest two values destined for a given PE are to reach the destination PE. Using the complexity figures given in § 3 for RARs, RAWs, REDUCE, CONCENTRATE and RANK in a 2^b-block, we see that each iteration of the loop of lines 5 to 24 takes $O(q^3 2^{\lceil b/q \rceil})$ time. So, the overall time spent in this loop is $O(q^4 n)$ where $N = n^q = 2^p$. Line 25 requires a sort on the field ORG. This takes only $O(q^2 n)$ time. Line 26 takes $O(q^3 n)$ time. So, CONNECT2 has time-complexity $O(q^4 n)$.

6. Connected ones. Before describing our algorithm to solve the connected ones problem, we introduce some terminology. Two 2^b-blocks are *siblings* iff they together form a 2^{b+1}-block. A 2^b-block is a *left* 2^b*-block* if it contains only PEs with bit b equal to zero. It is a *right* 2^b*-block* if all PEs have bit b equal to one. A PE in a 2^b-block is a *boundary* PE iff it is adjacent to a PE in its sibling 2^b-block. Let $d = b \textbf{ mod } q$. From the discussion in § 3, we know that bit b of a PE index is bit $\lfloor b/q \rfloor$ of dimension d when shuffled row-major indexing is used. So, a 2^{b+1}-block results from combining two sibling 2^b-blocks along dimension d. Also, a 2^b-block defines a PE array of size $m_{q-1} \times m_{q-2} \times \cdots \times m_0$, where the m_is are as given by (1). In particular, $m_d = 2^{\lceil b/q \rceil}$. The number of boundary PEs, t, in each 2^b-block is therefore $t = 2^{b - \lfloor b/q \rfloor}$.

Our algorithm for the connected ones problem actually partitions all the PEs into sets such that PE(i) and PE(j) are in the same set iff $A(i) = A(j) = 1$ and these two ones are connected. On termination of the algorithm, each partition is represented by a reduced min-tree. We shall have $R(i) = R(j) \neq \infty$ iff $A(i) = A(j) = 1$ and these two ones are connected. $R(i) = \infty$ iff $A(i) = 0$. To determine if all the ones are connected we need only check if the number of distinct R values (not counting ∞) is more than one. This is easy to do.

Our algorithm begins by considering each 2^0-block. For the lone PE in a 2^0-block, $R(i) = i$ if $A(i) = 1$, and $R(i) = \infty$ if $A(i) = 0$. From the sets of connected ones in each 2^b-block, we construct the sets of connected ones in each 2^{b+1}-block, $0 \leqq b < p$. (Recall that the MCC has $N = 2^p$ PEs.) The sets of connected ones in a 2^{b+1}-block are obtained by combining together the sets for the two 2^b-blocks contained in the 2^{b+1}-block. Sets are combined in the same manner as before, i.e., min-adjacent trees are combined together. In defining the set adjacencies for purposes of this combination, it is sufficient to consider only boundary node adjacency. Let i be a boundary PE in a left 2^b-block and let j be a boundary PE in the corresponding right 2^b-block. PE(i) is a *live boundary* PE iff it is adjacent to a PE(j) in its sibling 2^b-block and $A(i) = A(j) = 1$. Note that each live boundary node is adjacent to exactly one other live boundary node in its 2^{b+1}-block. A

PE is a *live root* PE if it is the root of a min-tree containing a live boundary PE (a live root PE can also be a live boundary PE). Thus, to obtain the sets of connected ones in a 2^{b+1}-block, we need only attempt to combine those 2^b-block sets with a live root. This combination can be carried out by considering only the adjacencies of the live boundary nodes.

With this introduction, we are ready to look at the details of procedure CONNECT ONES (Algorithm 4). This procedure uses a sub-procedure ROUTE(E, d, i) which transmits the data in the E register of each PE to the E register of a PE that is i units away along dimension d. i maybe positive or negative depending on the direction along dimension d that the route is to be performed. Lines 1 and 2 set-up the reduced min-trees corresponding to 2^0-blocks. Lines 3–30 build the reduced min-trees for each 2^{b+1}-block, $0 \leqq b < p$. Lines 4–11 determine the live boundary PEs. This is done by first determining the dimension, d, along which the member 2^b-blocks are combined (line 4). Following line 7, $E(i) = 1$ iff PE(i) is a live boundary PE in a left 2^b-block and following line 10, LIVE(i) = 1 iff PE(i) is a live boundary PE. Line 12 identifies the live root-PEs. Since the number, t, of boundary PEs in a 2^b-block is $2^{b-\lfloor b/q \rfloor}$, the number of live PEs (including live root PEs) in a 2^{b+1} block is no more than $\mathbf{min}\{2^{b+1}, 4t\}$. Live min-trees are combined by first concentrating the live nodes in each 2^{b+1}-block into a "corner" of that block. This requires us first to rank the live nodes (line 13) and then to set-up the adjacency list for each live boundary node. As remarked earlier, each live boundary node is adjacent to exactly one live boundary node. Lines 16–22 set-up ADJ(i) = j for each live boundary node i. j is the PE index to which i's adjacent live boundary node will be moved. The concentration of live nodes is performed in line 24. The records being concentrated are $G(i) = \langle R(i)$, ADJ(i), LIVE(i), ORG(i)$\rangle$. (As before, CONCENTRATE *permutes* the records in each 2^{b+1}-block so no record is destroyed.) CONNECT′ is the same as procedure CONNECT of § 4 except that line 1 is omitted and only PEs with LIVE = 1 are involved in any computation. Following line 26, live nodes i and j have the property that $R(i) = R(j)$ iff ORG(i) and ORG(j) are in the same set of connected ones for the 2^{b+1}-block containing PEs i and j. Lines 27 and 28 move the reduced min-trees created in line 26 back to the originating PEs. The reduced min-trees together with the PEs that were not live before the concentrate (line 24) form min-trees of height at most 3. Line 29 reduces these min-trees (note that $R(i) = i$ for a root). Line 29 may be restricted to PEs which had LIVE(i) = 0 before line 24 was executed.

Let $m = 2^{\lceil b/q \rceil}$ and $m' = 2^{\lceil k/q \rceil}$, where k is as given in line 25 of the algorithm. Lines 6, 9, 18, and 21 represent unit-distance routing and so have a complexity of $O(1)$. Line 12 is an RAW in 2^b-blocks and requires $O(q^2 m)$ unit-routes. Line 13 takes $O(qm)$ time. Lines 15 and 29 are RARs in 2^b-blocks and take $O(q^2 m)$ time. Line 24 has complexity $O(qm)$. Lines 26 and 27 have complexity $O(q^4 m' \log m')$ and $O(q^2 m')$ respectively. Finally, line 28 is a sort, and has complexity $O(q^2 m)$. Since

$$q^4 m' \log m' \simeq q^2 m \frac{bq-b}{2^{b/q^2}} = O(q^5 m),$$

each iteration of the **for** loop takes $O(q^5 m)$ time. The overall complexity of CONNECT ONES is therefore $O(q^6 n)$.

An $O(n)$ algorithm for the connected ones problem can also be arrived at using other block combination strategies. For example, if we have found the sets of connected ones in each 2^{b-q}-block then the sets for each 2^b-block may be found by combining together the sets in the $2^q 2^{b-q}$-blocks that make up the 2^b-block. A boundary PE of a

2^{b-q}-block can be adjacent to at most q boundary PEs in the remaining 2^q-1 blocks making up a 2^b-block. If we set up ADJ(i, 0: $q-1$) in a manner similar to the setting up of ADJ(i) in procedure CONNECT ONES then we can make a call to CONNECT′ (as in line 26) with k replaced by k' and 1 replaced by q. k' is the maximum number of live nodes in a 2^b-block.

Another possibility is to reduce the number of block combination steps to a constant. For example, if $q=3$ then we can consider $n^{3/4}\times n^{3/4}\times n^{3/4}$ blocks. Each PE in such an $n^{9/4}$-block is ones-adjacent to at most 6 other PEs in that block (two PEs are ones-adjacent iff they are adjacent and they both contain a one). Applying procedure CONNECT to each $n^{9/4}$-block in parallel takes $O(n^{3/4}\log n)$ time. The $n^{9/4}$-blocks can be combined together as before. The total number of boundary nodes in a $n^{3/4}\times n^{3/4}\times n^{3/4}$ block is 6 $n^{3/2}$. The number of blocks is $n^{3/4}$. So, the total number of live nodes (including roots) is at most $12n^{9/4}$. The time to concentrate these nodes is $O(n)$, and the time to run CONNECT′ is $O(n^{3/4}\log n)$. So, the overall time is $O(n)$.

```
line   procedure CONNECT ONES ($p$)
         //Find connected 1's. $2^p = n^q = N$ is the number of PEs.//
 1.      $R(i) := \infty$, $(A(i)=0)$ //$A$ is 0/1 pattern//
 2.      $R(i) := i$, $(A(i)=1)$
 3.      for $b := 0$ to $p-1$ do //combine pairs of $2^b$-blocks.//
 4.         $d := b$ mod $q$
 5.         $E(i) := (i)_b * A(i)$
 6.         call ROUTE($E$, $d$, $-1$) //unit distance route to left half//
 7.         $E(i) := (1-(i)_b) * A(i) * E(i)$
 8.         LIVE($i$) := $E(i)$ //live PEs in left block//
 9.         call ROUTE($E$, $d$, $+1$) //unit-distance route to right//
10.         LIVE($i$) := 1, $(E(i)=1)$ //live PEs in right block//
11.         BORDER($i$) := LIVE($i$) //mark live border PEs//
12.         LIVE($R(i)$) $\leftarrow$ 1, (LIVE($i$) = 1)
13.         call RANK($b+1$) //rank live nodes. Let $H$ = rank.//
14.         $H(i) := H(i) + 2^{b+1} * (i)_{p-1:b+1}$ //Add block bias//
15.         $R(i) \leftarrow H(R(i))$, (LIVE($i$) = 1) //update before move//
16.         ADJ($i$) := $\infty$ //build ADJ for live boundary PEs://
17.         $E(i) := H(i)$
18.         call ROUTE($E$, $d$, $-1$)
19.         ADJ($i$) := $E(i)$, ($(i)_b = 0$ and BORDER($i$) = 1)
20.         $E(i) := H(i)$
21.         call ROUTE($E$, $d$, $+1$)
22.         ADJ($i$) := $E(i)$, ($(i)_b = 1$ and BORDER($i$) = 1)
23.         ORG($i$) := $i$
            //Let record $G(i) = \langle R(i)$, ADJ($i$), LIVE($i$), ORG($i$)$\rangle$//
24.         call CONCENTRATE($b+1$)
25.         $k :=$ min$\{b+1, b - \lfloor b/q \rfloor + 2\}$ //live nodes in a block $\leqq 2^k$.//
26.         call CONNECT′($k$, 1) //same as CONNECT without line 1//
27.         $R(i) \leftarrow$ ORG($R(i)$), (LIVE($i$) = 1)
28.         $R$(ORG($i$)) $\leftarrow R(i)$ //move back//
29.         $R(i) \leftarrow R(R(i))$ //reduce the min-trees//
30.      end
31.    end CONNECT ONES
```

ALGORITHM 4

REFERENCES

[1] C. ARCELLI AND S. LEVIALDI, *Parallel shrinking in three dimensions*, Computer Graphics and Image Processing, 4(1972), pp. 21–30.

[2] L. E. CANNON, *A cellular computer to implement the Kalman filter*, Ph.D. Thesis, Montana State University, 1969.

[3] E. DEKEL, D. NASSIMI, AND S. SAHNI, *Parallel Matrix and Graph Algorithms*, University of Minnesota Technical Report #79-10, 1979.

[4] M. J. FLYNN AND S. R. KOSARAJU, *Processes and their interactions*, Kybernetes, 5 (1976), pp. 159–163.

[5] D. HIRSCHBERG, *Parallel algorithms for the transitive closure and the connected components problems*, Proceedings ACM 8th Annual Symposium on Theory of Computing, 1976, pp. 55–57.

[6] S. R. KOSARAJU, *On some open problems in the theory of cellular automata*, IEEE Trans. Comput., 23 (1974), pp. 561–565.

[7] S. R. KOSARAJU, *Speed of recognition of context-free languages by array automata, this Journal*, 4 (1975), pp. 331–340.

[8] ———, *Fast parallel processing array algorithms for some graph problems*, Proceedings ACM 11th Annual Symposium on Theory of Computing, 1979, pp. 231–236.

[9] S. LEVIALDI, *On shrinking binary picture patterns*, Comm. ACM, 15 (1972), pp. 2–10.

[10] K. LEVITT AND W. KAUTZ, *Cellular arrays for the solution of graph problems*, Comm. ACM, 15 (1972), pp. 789–801.

[11] D. NASSIMI AND S. SAHNI, *Bitonic sort on a mesh-connected parallel computer*, IEEE Trans. Comput., 28 (1979), pp. 2–7.

[12] ———, *An optimal routing algorithm for mesh-connected parallel computers*, J. Assoc. Comput. Mach., 27 (1980), pp. 6–29.

[13] ———, *Data Broadcasting in* SIMD *Computers*, University of Minnesota, Technical Report #79-17, 1979. IEEE Trans Comput., to appear.

[14] ———, *Parallel algorithms to set-up the Benes permutation network*, University of Minnesota Technical Report #79-19, 1979.

[15] C. THOMPSON AND H. KUNG, *Sorting on a mesh-connected parallel computer*, Comm. ACM, 20 (1977), pp. 263–271.

[16] F. L. VAN SCOY, *Parallel algorithms in cellular spaces*, Ph.D. Thesis, University of Virginia, 1976.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0010 $01.00/0

FACTORIZATION OF SYMMETRIC MATRICES AND TRACE-ORTHOGONAL BASES IN FINITE FIELDS*

GADIEL SEROUSSI† AND ABRAHAM LEMPEL†

Abstract. It is shown that every symmetric matrix A, with entries from a finite field F, can be factored over F into $A = BB'$, where the number of columns of B is bounded from below by either the rank $\rho(A)$ of A, or by $1+\rho(A)$, depending on A and on the characteristic of F. This result is applied to show that every finite extension Φ of a finite field F has a trace-orthogonal basis over F. Necessary and sufficient conditions for the existence of a trace-orthonormal basis are also given. All proofs are constructive, and can be utilized to formulate procedures for minimal factorization and basis construction.

Key words. matrix factorization, finite fields, trace, trace-orthogonal basis

1. Statement of the main results. Throughout this paper all matrix operations and concepts such as rank, linear dependence, etc., are taken over a finite field F whose order, p^r, and characteristic, p, will be specified, when so required, by writing $F = GF(p^r)$.

Consider a symmetric matrix A. A matrix B is called a *factor* of A if $A = BB'$, where B' is the transpose of B. Our basic result, as established by the existence parts of Theorems 1 and 2, is that over a finite field, every symmetric matrix has a factor.

A factor B of a symmetric matrix A is called a *minimal factor* if no factor of A has fewer columns than B. The number of columns of a minimal factor of A will be denoted by $\mu(A)$.

THEOREM 1. *Every symmetric matrix $A = (A_{ij})$ over $F = GF(2^r)$ has a factor, and $\mu(A) = \rho(A) + \delta(A)$, where $\rho(A)$ is the rank of A and*

$$\delta(A) = \begin{cases} 1 & \text{if } A_{ii} = 0 \text{ for all } i, \\ 0 & \text{otherwise.} \end{cases}$$

This result generalizes the main result of [1], where the theorem is shown to hold for the special case of $r = 1$, i.e. for matrices over the binary field $GF(2)$. The proof of Theorem 1 is presented in § 2; it is constructive and follows, essentially, along the lines of the proof in [1].

A square submatrix M of A is called a *principal submatrix* of A if either $M = A$ or if there exists a permutation matrix P such that M occupies the upper left corner of PAP'. Note that if A is symmetric, then so is M. Also, every symmetric matrix A, $A \neq 0$, has a principal submatrix M^* whose order and rank are both equal to $\rho(A)$. The determinant $|M^*|$ of such a submatrix of A will be referred to as a *major* of A. By Lemma 5 of § 3, every symmetric matrix $A \neq 0$ over a finite field F has the property that either all of its majors are quadratic residues in F or none of them are. For fields of odd characteristic we obtain the following result.

THEOREM 2. *Let p be an odd prime. Every symmetric matrix A over $F = GF(p^r)$ has a factor, and if $A \neq 0$ $\mu(A) = \rho(A) + \varepsilon(A)$, where*

$$\varepsilon(A) = \begin{cases} 0 & \text{if the majors of } A \text{ are quadratic residues in } F, \\ 1 & \text{otherwise.} \end{cases}$$

* Received by the editors May 13, 1979, and in final revised form February 25, 1980.

† Department of Computer Science, Technion–Israel Institute of Technology, Haifa, Israel.

The proof of this theorem is also constructive and it is presented in § 3. It should be noted that despite their formal resemblance, there is an essential difference between Theorems 1 and 2, as in finite fields of characteristic 2 every element is a quadratic residue. Thus, in such fields $\varepsilon(A)$ is always equal to zero while $\delta(A)$ depends on A.

A matrix B is called a *semifactor* of the symmetric matrix A if there exists a diagonal matrix Λ such that $A=B\Lambda B'$. In particular, every factor of A is also a semifactor of A, with $\Lambda=I$, the identity matrix. B is called a *minimal semifactor* of A if no semifactor of A has fewer columns than B. The number of columns of a minimal semifactor of A will be denoted by $\mu_s(A)$. In § 3 we also prove the following result.

THEOREM 3. *Let p be an odd prime. For every symmetric matrix $A\neq 0$ over $F=GF(p^r)$, $\mu_s(A)=\rho(A)$.*

By omission of the case $p=2$, Theorem 3 points out another peculiarity of $GF(2^r)$ which is due to the fact that every element of such a field is a quadratic residue in it. If $A=B\Lambda B'$ over $GF(2^r)$, then $A=B\Gamma(B\Gamma)'$ where $\Gamma=\Lambda^{2^{r-1}}$. Hence, over finite fields of characteristic 2, $\mu_s(A)=\mu(A)$ and, in view of Theorem 1, Theorem 3 does not extend to the case $p=2$ when $\delta(A)=1$.

An interesting application of the foregoing results is a method of constructing trace-orthogonal bases over finite fields. Consider a finite field $F=GF(q)$ and a finite extension $\Phi=GF(q^n)$ thereof. For $\alpha\in\Phi$, the *trace* of α over F is defined by

$$T(\alpha)=\sum_{i=0}^{n-1}\alpha^{q^i}.$$

A basis $\Omega=\{\omega_1,\omega_2,\cdots,\omega_n\}$ of Φ over F is called *a trace-orthogonal basis* (in short, TOB) if

$$T(\omega_i\omega_j)\neq 0 \text{ if and only if } i=j;$$

Ω is called a *trace-orthonormal basis* (in short, TONB) if Ω is a TOB and if $T(\omega_i^2)=1$ for all $i=1,2,\cdots,n$.

In § 4 we prove, constructively, the following result.

THEOREM 4. *Every finite extension $\Phi=GF(q^n)$ of a finite field $F=GF(q)$ has a* TOB *over F; Φ has a* TONB *over F if and only if either q is even or both q and n are odd.*

The validity of this theorem for the special case of $F=GF(2)$ has been established in [1]. A TONB is referred to in [2, Chapt. 4] as a self-complementary basis. Theorem 4 thus specifies the necessary and sufficient conditions for the existence of such a basis in an arbitrary finite field.

In the sequel we shall make free use of some fundamental properties of finite fields. Chapter 4 of [2] can serve as a suitable reference for the required background on finite fields.

2. Factorization over $GF(2^r)$. Following [1], we prove the existence part of Theorem 1 by describing the construction of a so called elementary factorization.

Consider a symmetric matrix $A=(A_{ij})$ of order n, and let $N=\{1,2,\cdots,n\}$. We define a set $N_1\subseteq N$ and a set of ordered pairs $N_2\subseteq N\times N$ as follows:

$$N_1=\left\{k\in N\,\middle|\,\sum_{j\neq k}A_{kj}\neq A_{kk}\right\},$$

$$N_2=\{(i,j)|i,j\in N,\, i<j, \text{ and } A_{ij}\neq 0\}.$$

A *k-column*, $k\in N_1$, is a column of n rows with a 1 in row k and zeros elsewhere; an *(i,j)-column*, $(i,j)\in N_2$, is a column of n rows with a 1 in rows i and j and zeros elsewhere. Let $E=[E_1E_2\cdots E_m]$ be a matrix of n rows and $m=|N_1|+|N_2|$ columns such that E contains one k-column for each $k \in N_1$ and one (i, j)-column for each $(i, j) \in N_2$, and let $\Lambda = (\Lambda_{ij})$ be a diagonal matrix of order m with

$$\Lambda_{ii} = \begin{cases} A_{kk} - \sum_{j \neq k} A_{kj} & \text{if } E_i \text{ is a } k\text{-column}, \\ A_{kj} & \text{if } E_i \text{ is a } (k, j)\text{-column}. \end{cases}$$

LEMMA 1. *Let E and Λ be as defined above. Then $A = E\Lambda E'$.*

The proof of this lemma is a straightforward generalization of the proof of Lemma 1 in [1].

Note that the construction leading to Lemma 1 is valid over every field. However, over $GF(2^r)$ it also produces a factor $B = E\Lambda^{2^{r-1}}$ of A, and thus establishes the existence part of Theorem 1. We proceed now to derive a procedure for reducing a given factor to a minimal one. We shall use the notation $c(M)$ for the number of columns of a matrix M.

LEMMA 2. *If B is a factor of A, then $c(B) \geqq \rho(A) + \delta(A)$.*

Proof. Clearly, $c(B) \geqq \rho(B) \geqq \rho(A)$. Hence, for $\delta(A) = 0$ the lemma is trivial. Assume $\delta(A) = 1$. Then $A_{kk} = 0$ for all k and, recalling that we work in $GF(2^r)$, we obtain

$$0 = A_{kk} = \sum_{j=1}^{n} B_{kj}^2 = \left(\sum_{j=1}^{n} B_{kj} \right)^2,$$

which implies $\sum_{j=1}^{n} B_{kj} = 0$ for all k. Hence, the columns of B are linearly dependent and, therefore, $c(B) \geqq \rho(B) + 1 \geqq \rho(A) + 1$. QED.

Lemma 2 shows that $\rho(A) + \delta(A)$ is a lower bound on $\mu(A)$. The following lemmas show how to achieve this bound.

LEMMA 3. *If $A = BB'$ is nonsingular and if a proper subset of columns of B are linearly dependent, then there exists a factor $\tilde{B}$ of A such that $c(\tilde{B}) < c(B)$.*

Proof. Since BB' is invariant under a permutation of the columns of B, we may assume that $B = [G \quad H]$, where $1 \leqq c(G) < c(B)$ and the columns of G are linearly dependent. Hence, there exists a nonzero vector $u = (u_i)$ such that $Gu = 0$. Without loss of generality, we may assume that $u_1 = 1$. Let $s = \sum u_i$ and let $B^* = [Z \quad H]$, where

$$Z = \begin{cases} G & \text{if } s = 0, \\ [G \quad 0] & \text{if } s \neq 0, \end{cases}$$

and where $[G \quad 0]$ is the matrix obtained by adjoining an all-zero column to G. Since $ZZ' = GG'$, B^* is a factor of A. Let Z_1 and H_1 be the first columns of Z and H respectively, and let $x = Z_1 + H_1$ and $Y = Z + xv'$, where

$$v = \begin{cases} u & \text{if } s = 0, \\ \begin{bmatrix} u \\ s \end{bmatrix} & \text{if } s \neq 0. \end{cases}$$

We have $Zv = Gu = 0$ and, since we compute in a field of characteristic 2, we also have

$$v'v = s^2 + u'u = s^2 + \sum u_i^2 = s^2 + \left(\sum u_i \right)^2 = 2s^2 = 0.$$

Therefore,

$$YY' = ZZ' + Zvx' + xv'Z' + xv'vx' = ZZ',$$

and $\hat{B} = [Y \quad H]$ is a factor of A. Now, recalling that $u_1 = 1$ and that $H_1 = Z_1 + x$, we obtain

$$Y_1 = Z_1 + xv_1 = Z_1 + xu_1 = Z_1 + x = H_1.$$

Thus, the joint contribution of Y_1 and H_1 to the product $\hat{B}\hat{B}'$ is null and, hence, the matrix $\tilde{B}$ obtained by deleting Y_1 and H_1 from $\hat{B}$ is also a factor of A. Consequently, we have

$$c(\tilde{B})=c(Y)+c(H)-2=c(Z)+c(H)-2\leqq c(G)+c(H)-1<c(B). \qquad \text{QED.}$$

LEMMA 4. *If $A=BB'$ is nonsingular and, if $c(B)>\rho(A)+\delta(A)$, then there exists a factor $\tilde{B}$ of A with $c(\tilde{B})<c(B)$.*

Proof. Nonsingularity of $A=BB'$ implies $\rho(A)=\rho(B)$, and if $c(B)>\rho(A)+\delta(A)$, then $c(B)>\rho(B)+\delta(A)$. Hence, if $\delta(A)=1$, then any $\rho(B)+1$ columns of B form a proper subset of linearly dependent columns of B. By Lemma 3, this implies the existence of a factor $\tilde{B}$ with $c(\tilde{B})<c(B)$, which validates the lemma when $\delta(A)=1$.

Assume now that $\delta(A)=0$. We still have $c(B)>\rho(B)$ and, therefore, there exists a nonzero vector $u=(u_i)$ such that $Bu=0$. If $u_i=0$ for some i, then again B has a proper subset of linearly dependent columns, and we are back at the case covered by Lemma 3. Thus, we can assume that $u_i\neq 0$ for all i. Since $\delta(A)=0$, the columns of B do not sum to an all-zero column. Therefore, given any constant c, there is at least one i such that $u_i\neq c$. In particular, there exists some i such that $u_i\neq\sum u_j$. Without loss of generality, we may assume that $1=u_1\neq\sum u_j$. Consequently, if B_j denotes the jth column of B and $s=\sum_{j=2}^m u_j$, where $m=c(B)$, we obtain $B_1=\sum_{j=2}^m u_jB_j$ and $s\neq 0$. Let $\hat{B}=B+B_1v'$, where $v'=(1\quad s^{-1}u_2\quad s^{-1}u_3\quad\cdots\quad s^{-1}u_m)$. Since

$$BvB_1'=(B_1+s^{-1}\sum_{j=2}^m u_jB_j)B_1'=(1+s^{-1})B_1B_1',$$

we have $BvB_1'+B_1v'B'=0$. Also, since $s^{-1}\sum_{j=2}^m u_j=1$, we have

$$v'v=1+s^{-2}\sum_{j=2}^m u_j^2=1+\left(s^{-1}\sum_{j=2}^m u_j\right)^2=0.$$

Therefore,

$$\hat{B}\hat{B}'=BB'+BvB_1'+B_1v'B'+B_1v'vB_1'=BB',$$

and $\hat{B}$ is a factor of A. Observing that $\hat{B}_1=B_1+B_1=0$, it follows that the matrix $\tilde{B}$ obtained by deleting the first column from $\hat{B}$ is also a factor of A with $c(\tilde{B})=c(B)-1$. QED.

This concludes the proof of Theorem 1 for the nonsingular case. The existence part is covered by Lemma 1, and the minimality part by Lemmas 2 and 4. Using the constructions of the proofs of Lemmas 1, 3, and 4, one can readily establish a simple procedure for obtaining a minimal factor.

When A is singular it contains a principal submatrix M^* whose rank and order are both equal to $\rho(A)$. (We may assume $\rho(A)>0$, for the theorem is trivial when $A=0$.) One can readily adapt the argument given in [1] to show that $\delta(M^*)=\delta(A)$ over every field of characteristic 2, and that there exists a nonsingular matrix R such that

$$RAR'=\begin{bmatrix} M^* & 0\\ 0 & 0\end{bmatrix}.$$

Since M^* is nonsingular, it has a minimal factor H with $c(H)=\rho(M^*)+\delta(M^*)=\rho(A)+\delta(A)$. Hence

$$RAR'=\begin{bmatrix} H\\ 0\end{bmatrix}[H'\quad 0],$$

and $B = R^{-1}\left[\begin{smallmatrix}H\\0\end{smallmatrix}\right]$ is a factor of A with $c(B) = \rho(A) + \delta(A)$. This validates Theorem 1 for the singular case as well.

3. Factorization over finite fields of odd characteristic. Let p be an odd prime and let A be a symmetric matrix of order n over $F = GF(p^r)$. On our way to prove Theorem 2 we first prove Theorem 3. Since both theorems are void when $A = 0$, we shall assume throughout that $\rho(A) > 0$.

Proof of Theorem 3. We prove the theorem by induction on the order n of A. The theorem being trivially true for $n = 1$, we assume it to hold for $1 \leqq n \leqq m - 1$, and we consider a symmetric matrix A of order m and rank $\rho(A) > 0$ over F. Since the number of columns of a semifactor of A cannot be less than $\rho(A)$, it suffices to show that there exists one with no more columns.

Case I: $\rho(A) < m$. In this case, A has a principal submatrix M^* whose rank and order are both equal to $\rho(A)$. As in the singular case of § 2, there exists a nonsingular matrix R such that

$$RAR' = \begin{bmatrix} M^* & 0 \\ 0 & 0 \end{bmatrix}.$$

Since the order of M^* is less than m, it follows from the inductive hypothesis that M^* has a semifactor B^* with $c(B^*) = \rho(A)$. Therefore, $B = R^{-1}\left[\begin{smallmatrix}B^*\\0\end{smallmatrix}\right]$ is a semifactor of A with $c(B) = \rho(A)$.

Case II: $\rho(A) = m$ *and* $A_{kk} \neq 0$ *for some* k. Let A_k denote the kth column of A, and let $\tilde{A} = A - A_{kk}^{-1}A_kA_k'$. Since $\tilde{A}_k = A_k - A_{kk}^{-1}A_kA_{kk} = 0$, we have $\rho(\tilde{A}) < m$. We cannot have $\tilde{A} = 0$, because then $\rho(A) = 1 < m$, contrary to the definition of Case II. Thus, $\tilde{A} \neq 0$ and, by Case I, $\tilde{A}$ has a semifactor $\tilde{B}$ with $c(\tilde{B}) = \rho(\tilde{A})$. Therefore, $B = [\tilde{B} \quad A_k]$ is a semifactor of A with $c(B) = \rho(\tilde{A}) + 1 \leqq m$.

Case III: $\rho(A) = m$ *and* $A_{kk} = 0$ *for all* k. Since A is nonsingular and the field characteristic is odd, $2A_{1k} \neq 0$ for some $k \neq 1$. Let $A^* = -(2A_{1k})^{-1}(A_k - A_1)(A_k - A_1)'$ and let $\tilde{A} = A - A^*$. Since $A_{11} = A_{kk} = 0$ and $A_{1k} = A_{k1}$, we have

$$\begin{aligned}\tilde{A}_1 - \tilde{A}_k &= A_1 - A_k - (A_1^* - A_k^*) \\ &= A_1 - A_k + (2A_{1k})^{-1}(A_k - A_1)(A_{k1} - A_{11} - A_{kk} + A_{1k}) \\ &= A_1 - A_k + (A_k - A_1) = 0.\end{aligned}$$

Thus, $\rho(\tilde{A}) < m$ and, since in this case too $\tilde{A} \neq 0$, by Case I, A has a semifactor $\tilde{B}$ with $c(\tilde{B}) = \rho(\tilde{A})$. Therefore, as in Case II, $B = [\tilde{B} \quad x]$, where $x = A_k - A_1$, is a semifactor of A with $c(B) = \rho(\tilde{A}) + 1 \leqq m$. This completes the proof of Theorem 3. QED.

The following lemma guarantees that the definition of $\varepsilon(A)$ in Theorem 2 is unambiguous.

LEMMA 5. *Let A be a symmetric matrix over F. Then, either every major of A is a quadratic residue in F, or none of them is.*

Proof. The lemma is obvious when A is nonsingular. If A is singular, let M and $\tilde{M}$ be nonsingular principal submatrices of order $\rho(A)$ of A. Then, there exist nonsingular matrices T and $\tilde{T}$ such that

$$A = T\begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix}T' = \tilde{T}\begin{bmatrix} \tilde{M} & 0 \\ 0 & 0 \end{bmatrix}\tilde{T}'.$$

Hence,

$$\begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix} = S\begin{bmatrix} \tilde{M} & 0 \\ 0 & 0 \end{bmatrix}S',$$

where

$$S=\begin{bmatrix} S_{11} & S_{12}\\ S_{21} & S_{22}\end{bmatrix}=T^{-1}\tilde{T},$$

and S_{11} is a square matrix of order $\rho(A)$. It follows that

$$\begin{bmatrix} M & 0\\ 0 & 0\end{bmatrix}=\begin{bmatrix} S_{11}\\ S_{21}\end{bmatrix}\tilde{M}[S'_{11} \quad S'_{21}],$$

and $M=S_{11}\tilde{M}S'_{11}$. Therefore, $|M|=|S_{11}|^2|\tilde{M}|$ and $|M|$ is a quadratic residue if and only if $|\tilde{M}|$ is. QED.

The following lemma will be useful in constructing a factor from a given semifactor of a matrix A.

LEMMA 6. *If $\lambda\in F=GF(p^r)$ is not a quadratic residue in F, then λ is the sum of two quadratic residues in F.*

Proof. Let γ be a primitive element of F (i.e., the multiplicative order of γ is p^r-1). Then the set Q of quadratic residues of F is given by

$$Q=\left\{\gamma^{2i}|1\leqq i\leqq\frac{p^r-1}{2}\right\}\cup\{0\},$$

while the set $\bar{Q}$ of nonresidues is given by

$$\bar{Q}=\left\{\gamma^{2i-1}|1\leqq i\leqq\frac{p^r-1}{2}\right\}.$$

Since $|Q|=\frac{1}{2}(p^r+1)$ does not divide $|F|=p^r$, Q is not an additive subgroup of F. Therefore, Q is not closed under addition, and there exist q_1, $q_2\in Q$ such that $q_1+q_2=\gamma^{2t-1}$ for some t, $1\leqq t\leqq\frac{1}{2}(p^r-1)$. Hence, for each $i=1,2,\cdots,\frac{1}{2}(p^r-1)$ we have

$$\gamma^{2i-1}=q_1\gamma^{2(i-t)}+q_2\gamma^{2(i-t)}.$$

Since Q is closed under multiplication, it follows that every element of $\bar{Q}$ is the sum of two elements from Q. QED.

LEMMA 7. *If $A=BB'\neq 0$ then $c(B)\geqq\rho(A)+\varepsilon(A)$.*

Proof. Since we must have $c(B)\geqq\rho(A)$, it suffices to show that the equality $c(B)=\rho(A)$ implies $\varepsilon(A)=0$. Assume $c(B)=\rho(A)$, and let M^* be a principal submatrix of A whose order and rank are equal to $\rho(A)$. Then, if B^* is the corresponding submatrix of B, we have $M^*=B^*(B^*)'$ and $|M^*|=|B^*|^2$. Hence, if $c(B)=\rho(A)$, the majors of A are quadratic residues and $\varepsilon(A)=0$. QED.

Proof of Theorem 2. By Lemma 7, it suffices to show that every symmetric matrix $A\neq 0$ over F has a factor B with $c(B)=\rho(A)+\varepsilon(A)$. To this end, consider a minimal semifactor S of A. By Theorem 3, $c(S)=\rho(A)$, and there exists a diagonal matrix Λ such that

$$A=S\Lambda S'=\sum_{i=1}^{t}\lambda_iS_iS'_i,$$

where $t=\rho(A)$ and $\lambda_i=\Lambda_{ii}$, $1\leqq i\leqq t$.

Let γ, Q and $\bar{Q}$ be as defined in Lemma 6. Then, for each $\lambda_i\in Q$ there exists an element $a_i\in F$ such that $\lambda_i=a_i^2$ and $\lambda_iS_iS'_i=\tilde{S}_i\tilde{S}'_i$, where $\tilde{S}_i=a_iS_i$. Similarly, for each $\lambda_j\in\bar{Q}$ there exists an element $b_j\in F$ such that $\lambda_j=b_j^2\gamma$ and $\lambda_jS_jS'_j=\gamma\tilde{S}_j\tilde{S}'_j$, where $\tilde{S}_j = b_j S_j$. Thus, without loss of generality, we may assume that $\lambda_i = \gamma$ for $1 \leqq i \leqq s$, and that $\lambda_i = 1$ for $s < i \leqq t$.

Now, let M^* be a principal submatrix of A whose order and rank are equal to $\rho(A)$, and let S^* be the corresponding submatrix of S. Then $|M^*| = |\Lambda|\,|S^*|^2 = \gamma^s |S^*|^2$, and we have $\varepsilon(A) = 0$ if and only if s is even.

Case I: $s = 2k$. If $k = 0$, then $\Lambda = I$, and $B = S$ is a factor of A with $c(B) = \rho(A)$. If $k \geqq 1$, then, by Lemma 6, there exist $\alpha, \beta \in F$ such that $\gamma = \alpha^2 + \beta^2$. Consider the matrix B whose columns are defined by

$$B_{2i-1} = \alpha S_{2i-1} + \beta S_{2i},$$
$$B_{2i} = \beta S_{2i-1} - \alpha S_{2i},$$

for $1 \leqq i \leqq k$, and $B_i = S_i$ for $s < i \leqq t$. Since for $1 \leqq i \leqq k$ we obtain

$$\begin{aligned} B_{2i-1}B'_{2i-1} + B_{2i}B'_{2i} &= (\alpha^2 + \beta^2)(S_{2i-1}S'_{2i-1} + S_{2i}S'_{2i}) \\ &= \gamma(S_{2i-1}S'_{2i-1} + S_{2i}S'_{2i}), \end{aligned}$$

it follows that B is a factor of A with $c(B) = \rho(A)$.

Case II: $s = 2k + 1$. If $k > 0$, we can apply the construction of Case I to obtain a semifactor $[S_s \tilde{B}]$ of A, $c(\tilde{B}) = \rho(A) - 1$, such that

$$A = \gamma S_s S'_s + \tilde{B}\tilde{B}' = (\alpha^2 + \beta^2) S_s S'_s + \tilde{B}\tilde{B}'.$$

Thus, the matrix B whose columns are defined by $B_1 = \alpha S_s$, $B_2 = \beta S_s$, and $B_{2+i} = \tilde{B}_i$ for $1 \leqq i \leqq \rho(A) - 1$, is a factor of A with $c(B) = \rho(A) + 1$.

Since Case I corresponds to $\varepsilon(A) = 0$ and Case II corresponds to $\varepsilon(A) = 1$, this completes the proof of Theorem 2. QED.

4. Trace-orthogonal bases over finite fields. Consider a finite field $F = GF(q)$ and a finite extension $\Phi = GF(q^n)$ thereof. Recalling that $a \in F$ if and only if $a^q = a$, and that over F $(\alpha + \beta)^q = \alpha^q + \beta^q$, one can readily verify the following properties of the trace operator T:

(t.1) $T(\alpha) \in F$ for all $\alpha \in \Phi$,

(t.2) $T(\alpha^q) = T(\alpha)$ for all $\alpha \in \Phi$,

(t.3) $T(a\alpha + b\beta) = aT(\alpha) + bT(\beta)$ for all $\alpha, \beta \in \Phi$ and $a, b \in F$.

Let γ be a primitive element of Φ and let $V = (V_{ij})$ be a square matrix of order n whose entries are defined by

$$V_{ij} = \gamma^{iq^j}, \qquad 0 \leqq i, j \leqq n - 1.$$

It is easy to see that V is a Vandermonde matrix over Φ, and hence its determinant is given by

$$|V| = \prod_{0 \leqq i < j \leqq n-1} (\gamma^{q^j} - \gamma^{q^i}).$$

Consider the symmetric matrix $A = VV'$. Since V is nonsingular over Φ, so is A. Moreover, since

$$A_{ij} = \sum_{k=0}^{n-1} \gamma^{iq^k} \gamma^{jq^k} = T(\gamma^{i+j}), \qquad 0 \leqq i, j \leqq n - 1,$$

if follows, by (t.1), that the entries of A belong to F and that A is nonsingular over F.

LEMMA 8. *Over the field $F = GF(q)$, $\mu(A) = n$ if and only if either q is even or both q and n are odd.*

Proof. Since A is nonsingular, it suffices to show that when q is even, $A_{kk} \neq 0$ for at least one k, and that when q is odd, $|A|$ is a quadratic residue in F if and only if n is odd. Suppose q is even. Then, the characteristic of F is 2, and we have

$$A_{kk} = T(\gamma^{2k}) = (T(\gamma^k))^2 = A_{k0}^2.$$

Since A is nonsingular, there must be at least one k such that $A_{k0} \neq 0$ and, thus, $A_{kk} \neq 0$ for at least one k.

Assume now that q is odd. Since $|A| = |V|^2$ over Φ, it suffices to determine the condition under which $|V|$ belongs to F. To this end, recall that

$$|V|^q = \prod_{0 \leqq i < j \leqq n-1} (\gamma^{q^j} - \gamma^{q^i})^q.$$

Since $(\alpha - \beta)^q = \alpha^q - \beta^q$, and $\alpha^{q^n} = \alpha^{q^0} = \alpha$ for all $\alpha, \beta \in \Phi$, we obtain

$$|V|^q = \prod_{1 \leqq i < j \leqq n-1} (\gamma^{q^j} - \gamma^{q^i}) \cdot \prod_{i=1}^{n-1} (\gamma^{q^0} - \gamma^{q^i}) = (-1)^{n-1} |V|.$$

Hence, $|V| \in F$ if and only if n is odd. QED.

By Lemma 8 and Theorem 3, it follows that the matrix $A = VV'$ can always be factored *over F* into

$$A = S\Lambda S',$$

where Λ is a diagonal matrix and $c(S) = n$.

Now, let $m = q^n - 1$ and let $M = (M_{ij})$ be the $m \times m$ symmetric matrix with

$$M_{ij} = T(\gamma^{i+j}), \qquad 0 \leqq i, j \leqq m-1.$$

Note that A is a submatrix of M, occupying the $n \times n$ upper left corner of M. We now show that the rank of M is n. Since γ is primitive in Φ, the n powers γ^j, $0 \leqq j \leqq n-1$, form a basis of Φ over F. Therefore, for each $k = 0, 1, \cdots, m-1$ there exist n elements $L_{jk} \in F$, $0 \leqq j \leqq n-1$, such that $\gamma^k = \sum_{j=0}^{n-1} L_{jk}\gamma^j$. Consequently, we obtain

$$\sum_{j=0}^{n-1} M_{ij}L_{jk} = \sum_{j=0}^{n-1} L_{jk}T(\gamma^{i+j}),$$

which, by (t.3), yields

$$\sum_{j=0}^{n-1} M_{ij}L_{jk} = T\left(\sum L_{jk}\gamma^{i+j}\right) = T(\gamma^{i+k}).$$

This shows that every column M_k of M is spanned by its first n columns $M_0, M_1, \cdots, M_{n-1}$, and since $\rho(A) = n$ we also have $\rho(M) = n$. Moreover, if L denotes the $n \times m$ matrix formed by the L_{jk}, $0 \leqq j \leqq n-1$, $0 \leqq k \leqq m-1$, then

$$M = L'AL = L'S\Lambda S'L = B\Lambda B'$$

where $B = L'S$.

It is also easy to verify that the m columns M_k, $0 \leqq k \leqq m-1$, of M, and the all-zero column Z of length m, form a field which is isomorphic to Φ under componentwise addition as defined in F and multiplication defined by

$$M_kZ = ZM_k = Z, \qquad 0 \leqq k \leqq m-1,$$

$$ZZ = Z,$$

$$M_iM_j = M_{i+j}, \qquad 0 \leqq i, j \leqq m-1,$$

where all column indices are taken modulo m. The one-to-one correspondence between these columns and the elements of Φ expressed as powers of γ is

$$\begin{aligned} &M_k \leftrightarrow \gamma^k, \qquad 0 \leqq k \leqq m-1, \\ &Z \leftrightarrow 0. \end{aligned}$$

Note that Z and the m columns of M exhaust all linear combinations of the first n columns of M over F. We can prove now the following result.

LEMMA 9. *The columns* $B_0, B_1, \cdots, B_{n-1}$ *of a minimal semifactor* B *of* M *form a* TOB *of* Φ *over* F.

Proof. Since $M = B\Lambda B'$, it is clear that the columns of B span those of M. Moreover, since $c(B) = n$ and $\rho(M) = n$, we must have $\rho(B) = n$ and, hence, the columns of B form a basis of Φ over F. As such, it should be clear that the columns of B are a subset of those of M. It remains to be shown that this basis is actually a TOB. Let γ^{e_i}, $0 \leqq j \leqq n-1$, $0 \leqq e_j \leqq m-1$, be the power of γ corresponding to B_j. Without loss of generality, we may assume that $e_0 < e_1 < \cdots < e_{n-1}$. Substituting γ^k for M_k and γ^{e_i} for B_j we can replace $M = B\Lambda B'$ by

$$[\gamma^0 \gamma^1 \gamma^2 \cdots \gamma^{m-1}] = [\gamma^{e_0} \gamma^{e_1} \cdots \gamma^{e_{n-1}}]\Lambda B'.$$

It is clear from this relation that for each j such that $0 \leqq j \leqq n-1$, and for each $i = 0, 1, \cdots, n-1$,

$$(\Lambda B')_{ie_j} = \lambda_i B_{e_j i} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Since B_j corresponds to γ^{e_i} which, in turn, corresponds to M_{e_j}, we have

$$B = [M_{e_0} M_{e_1} \cdots M_{e_{n-1}}].$$

This implies

$$B_{ij} = M_{ie_j} = T(\gamma^{i+e_j})$$

and hence,

$$\lambda_i B_{e_j i} = \lambda_i T(\gamma^{e_i + e_j}) = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Since $|\Lambda| \neq 0$, $\lambda_i \neq 0$ for all i, and we have

$$T(\gamma^{e_i} \cdot \gamma^{e_j}) = \begin{cases} \lambda_i^{-1} & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

This proves, as claimed, that the columns of B form a TOB of Φ over F. QED.

Proof of Theorem 4. The existence part with respect to a TOB is covered by Lemma 9. When q is even or when both q and n are odd, it follows from Lemma 8, that A can be factored over F into $A = SS'$ (i.e., $\Lambda = I$) with $c(S) = n$. Consequently, $M = BB'$ over F, and the columns of B form a TONB of Φ over F. To prove that the condition of the theorem is necessary for the existence of a TONB, suppose Φ has such a basis over F. Then, the principal $n \times n$ submatrix of M whose columns (and rows) correspond to the TONB is the identity matrix I of order n. The determinant $|I| = 1$ of this principal submatrix is a major of M. Consequently, either q is even or q is odd and $\varepsilon(M) = 0$. Hence, either q is even or q is odd and $\mu(M) = \mu(A) = n$. By Lemma 8, this implies that either q is even or both q and n are odd. QED.

REFERENCES

[1] A. LEMPEL, *Matrix factorization over $GF(2)$ and trace-orthogonal bases of $GF(2^n)$*, this Journal. 4 (1975), pp. 175–186.

[2] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0011 $01.00/0

A MODEL AND PROOF TECHNIQUE FOR MESSAGE-BASED SYSTEMS*

JEROME A. FELDMAN† AND ANIL NIGAM†

Abstract. Distributed computing with widely separated machines is a subject of growing theoretical and practical interest. This paper attempts to present a framework for the analysis of message-based distributed computations. This is done in the context of the classical critical section problem and the high-level language, PLITS. The proof techniques described are based on the use of finite-state machines which characterize the external behavior of each module in the distributed computation.

Key words. distributed computing, proof techniques, critical section problem, communication processes, finite-state model

1. Introduction. Distributed computing, the execution of a single computation by a network of computers, is currently a very hot topic. As is usual in computing, applications are proceeding in advance of any systematic study of the subject. The technical and economic reasons for the great increase in distributed computing seem likely to remain in force for some time. In addition to its practical significance, distributed computing is a subject of inherent intellectual interest because every complex system, from a micro-organism to a society, does its computations with a number of fairly independent subsystems whose communications can be viewed as messages. We have previously worked on a message-based model for system programs [2] and on a high-level programming language (PLITS) for distributed computing [11]. This paper attempts to lay out an abstract framework for the analysis and synthesis of message-based distributed computation.

The direct motivation for the current effort came from a graduate seminar that was working on the refinement and applications of PLITS. The students had very little trouble *writing* message-based PLITS programs but had great difficulty *proving* them correct, even though they could prove ordinary programs. One difficulty was the absence of an agreed-upon formalism for demonstrating the correctness of such programs. An initial attempt at such a formalism is one of the three goals of this paper. A second goal is to extend the recent work on synchronization of modules which share memory to those which communicate only by messages. Our third goal is to contribute to the development of programming and proof techniques which will help in the production of demonstrably correct solutions to real distributed computing problems.

The choice of a model is often the crucial step in the formalization of a domain. Distributed computing presents particular modeling difficulties because an adequate treatment of it requires consideration of several poorly understood concepts including time-dependency, parallelism and unreliability. There also is much less accumulated intuition than is available for monoprocessing. The particular model chosen here assumes that reliable transmission (including preserving sequence) and the detection of dead modules is treated by an underlying system. This is the same level of model used in our PLITS work and seems to be a clean abstraction of a level to be found in most distributed systems.

* Received by the editors December 29, 1977, and in revised form August 1, 1979. This research was supported by the Alfred P. Sloan Foundation under grant 74-12-5 and by the National Science Foundation under grant MCS76-10825.

† Department of Computer Science, University of Rochester, Rochester, New York 14627.

This paper can also be viewed as part of the continuing effort to carefully define and study the problems of synchronizing multiple concurrent program modules (or processes). The most recent informal effort along these lines is a paper by Peterson and Fischer [23] which discusses the critical section problem for distributed systems. A more formal investigation has been reported in [7]. The critical section problem is not particularly interesting (cf. below), but is very simple and has a long history. There is currently a great deal of work, on the validation of communication protocols [26], which is also related to our work.

We will loosely follow the formulation of Peterson and Fischer, but our concern is with message-based coordination problems. Essentially all previous work on the problems has been based on models employing shared memory for system state variables (flags). This model makes sense in a single processor with multiple modules, but does not have an obvious realization in a system distributed across a network. One thing we will do later is explore direct extensions of the flag style of solution to message-based (realistic) distributed systems. The historical context of the critical section problem is in the design of resource-sharing operating systems. A resource (such as a printer or a disk unit) had to be used by only one process of the system at a time. Since all the processes were on a single machine, the reading of flags was perfectly straightforward. We are concerned with the allocation of resources by a widely distributed system. This paper explores the techniques available for synchronizing message-based systems and, especially, the proof methods which can be employed with them.

There is an additional set of problems involving the allocation of multiple resources among parallel processes with overlapping requirements [8]. This paper does not directly address these problems, but the results developed here can be used to extend the existing coordination algorithms to distributed systems. There is work in progress [21] on applying results of this paper to data base problems.

A typical resource to be allocated by a distributed processing system might be access to a distributed data base or a band of the electromagnetic spectrum or a volume of air space by an air traffic control system. Often the best solution to these shared-resource allocation problems is to have a specific module in charge of allocating the resource (this is analogous to replacing global variables with classes or modules). The major problem with having one controller per resource is that a failure in the controller totally precludes the use of the resource. If there are redundant controllers for a resource, they will have a synchronization problem like those discussed here.

One of our main goals is to develop a model for message-based computation which is both useful for proving correctness and sufficiently realistic to be implemented as a layer of practical distributed computing systems. The model will be defined in terms of specific assumptions (or axioms) such as:

(A1) Messages sent from a sending module to a receiving module will all arrive safely and in the order issued with no duplications.

(A2) There is enough space to queue all incoming messages for each module.

For each such assumption (there are two more), we will describe briefly how its validity can be achieved. Assumption A1 (reliable transmission) is at the heart of all message-based systems [27] and has been called the pipelining assumption [4]. It is usually achieved by using redundancy checks and sequence numbers in messages and requiring a positive acknowledgment from the receiver before the sender discards its copy of a message. Assumption A2 is usually accomplished by keeping the sender continuously informed of the available capacity of the receiver. If messages are all processed in the order received, it is equivalent (although less efficient) to have the sender retransmit messages rejected because of inadequate queue space. It is feasible to have essentially unbounded message queues by using secondary storage. If even this very large limit is exceeded, the system is deemed to be broken. These assumptions (and the two which follow) seem to be at a level appropriate for our goals. It is reasonable to ask that our methodology be adequate to prove assumptions A1–A4 given a more primitive model and we will carry out one such reduction below.

In order to be explicit, we will assume that modules are programmed in an Algol-like language which has been augmented with constructs dealing with messages ([11] contains a complete specification for a usable example of such a language). In the language used here, a module can transmit a message by executing a statement of the form

Send ⟨message⟩ *To* ⟨module⟩.

Messages are queued at the receiving end until requested. We also postulate a single primitive, *Receive* ⟨message⟩, by which the receiver process can request and wait for a message (from the queue provided by the underlying system). A module executing a *Receive* will be assumed to eventually be resumed with either the first incoming message or with the special message "dead". The treatment of dying and reincarnating modules is the basis for assumptions A3 and A4 and will be discussed in detail below. We also want to explicitly account for arbitrary differences in execution speeds among modules. Intuitively, one can imagine that a module has been swapped out of main memory between any two statements.

With this model, we can devise a very simple solution to the critical section problem that satisfies the three conditions of mutual exclusion, no lockout, and no deadlock assuming neither module dies and that each module attempts to enter the critical section infinitely often. There are two messages, "canI?" and "yes", in addition to the system message "dead". Neither module will enter its critical section until it receives a "yes" from the other one. In the interesting case when the modules exchange "canI?" requests, module P_0 will have precedence. The code for the two modules P_0 and P_1 is shown in Fig. 1.

```
$P_0$
NEUTRAL:
  ⟨non-critical section⟩
  Send canI? To $P_1$
TRYING: Receive (message)
  If message = canI? Then
  Begin
    Go to TRYING
  End

CRITICAL: ⟨critical section⟩
  Send yes To $P_1$
  Go to NEUTRAL
```

```
$P_1$
NEUTRAL:
  ⟨non-critical section⟩
  Send canI? To $P_0$
TRYING: Receive (message)
  If message = canI? Then
  Begin
    Send yes To $P_0$
    Go to TRYING
  End
CRITICAL: ⟨critical section⟩;
  Go to NEUTRAL
```

FIG. 1. *Programs for the solution of the oversimplified critical section problem.*

Intuitively, the solution works because there is a complete "canI"-"yes" handshake for each entry to a critical section. In this (much too simple) case, each module can only respond to messages at one place, TRYING. Independent of who sends the first "canI?" message, each will send and receive a "canI?". Module P_0 has precedence so it just waits for another message. P_1 yields precedence so it will send a "yes" and wait for a message. P_0 will get the "yes", enter its critical section, and complete it. Then it will send a "yes" to P_1. This enables P_1 to enter its critical section. We would like to be able to provide more rigorous proofs for solutions to this kind of problem. Our first attempt will be to follow the style of Peterson and Fischer [23].

Proof. We prove that the algorithms of Fig. 1 solve the critical section problem (recall that we are assuming for now that neither module ever dies). Notice first that each module receives all messages in one place (labeled TRYING). A module can enter its critical section only by receiving a "yes" message at this point. Suppose P_0 has sent a "yes" to P_1; then it is not in its critical section and cannot enter again without (going through its noncritical section and) sending a "canI?" and getting a "yes" response. Thus the sequence of messages sent by P_0 strictly alternates between "canI?" and "yes".

Now consider the module P_1 which will see this alternating sequence. Whenever it sees a "canI?" it responds "yes" and waits for the next message. But the next message it receives was shown above to always be "yes". This enables P_1 to enter its critical section. After P_1 finishes its critical section, it will (go to NEUTRAL, send a "canI?", and) receive the next "canI?" from P_0. Thus, the two modules will strictly alternate which one is in the critical section. Strict alternation obviously entails mutual exclusion and the absence of deadlock and of lockout.

The algorithms in Fig. 1 are a little more complex than is strictly necessary here, because we want similar algorithms to work when dying modules are considered. The proof becomes more involved and will be deferred until more machinery is developed in § 2. It is sometimes considered more elegant to have the programs for each module be identical. To convert our solution to one with this property, one adds tests in each module on whether it gets or yields lexicographic preference. The problem solved here is not identical to that faced in Peterson and Fischer [23], because we assume each module tries to enter its critical section infinitely often; later versions of our solution relax this requirement.

The proof presented above is in the informal style used in the literature [17], [25] on synchronization problems. There are some people who find it difficult to convince themselves that such proofs are correct (sometimes they are not) and under which circumstances. In the following section, we attempt to lay out a more formal approach to proving properties of message-based systems. Subsequent sections elaborate on the problem, the model, and the solution presented above. The central question is how to produce efficient and demonstrably correct solutions to realistic problems in distributed computing. One approach we will follow is layering: a given model of the "system" is defined and a simple solution to the (e.g., critical section) problem proved relative to that model. Then we consider how to realize the model in terms of a more primitive "system". Then the simple solution is used to develop more realistic solutions. Finally, we consider how to synthesize larger systems from demonstrably correct parts.

2. An abstract formulation. We will present informally a formal model which has proved useful in the design and proof of programs like those of Fig. 1 and more realistic message-module systems. Flow table methods from sequential circuit theory have been used [6] to design control programs for parallel processes. The basic idea is to describe the external behavior of each module as a finite state machine, with a small number of states. The possible messages in the system are also divided into a small number of classes, but this trivializes in the current example because there are only two possible messages (still assuming neither module dies). Now, if we can define all the assumptions carefully, and if the modules can be described (externally) in such a simple way, proofs can be made much clearer. Intuitively, the modules P_0 and P_1 of Fig. 1 each have three distinct states: noncritical, trying, and critical. Fig. 2 presents the programs of Fig. 1 and the finite state table that characterizes each one.

```
$P_0$
NEUTRAL:
  ⟨non-critical section⟩
  Send canI? To $P_1$
TRYING: Receive (message)
  If message = canI? Then
    Begin
      Go to TRYING
    End

CRITICAL: ⟨critical section⟩
  Send yes To $P_1$
  Go to NEUTRAL
```

```
$P_1$
NEUTRAL:
  ⟨non-critical section⟩
  Send canI? To $P_0$
TRYING: Receive (message)
  If message = canI? Then
    Begin
      Send yes To $P_0$
      Go to TRYING
    End
CRITICAL: ⟨critical section⟩
  Go to NEUTRAL
```

message state	?	y	—
T	*T*	*C*	*T*
C			*N*/*y*
N	$/Q_e$		*T*/?

message state	?	y	—
T	*T*/*y*	*C*	*T*
C	$/Q_e$		*N*
N	$/Q_e$		*T*/?

where T = *TRYING*, C = *CRITICAL*, N = *NEUTRAL*, ? = "*canI?*", y = "*yes*", Q_e is the action of queuing the message, to be processed later.

FIG. 2

The plan is to use the tables as an intermediate representation in establishing the properties of message systems. A proof will consist of showing that each table captures the external behavior of the related program (cf. axiomatic semantics) and that the collection of finite state tables satisfies the required global conditions. The aim is to avoid global predicates based on the values of internal or auxiliary variables [22] because these are generally not observable. Some progress has been reported [5] in establishing correctness via a combination of state behavior and global predicates.

State tables describe the actions and state transitions that occur in a program either by the program receiving a message (columns ?, y) or by internal actions of the program (column -). The tables in Fig. 2 are particularly simple. Only one state (T) has message reception and it changes only on message reception. The other two states (C and N) queue messages and have the additional nice property that they each have a unique successor and the transition is triggered by an internal action. The notation "T/?" in an entry means that the program (simultaneously) sends the message "?" and goes into state "T". The two blank entries indicate that there is no legal way that a "y" message can appear while a module is in states C or N. The semantics of a blank entry could be to discard the message, to signal a control module, etc. We leave it undefined here.

Looking ahead, we will want to develop the mathematics of systems of simple state-message machines and use these to produce demonstrably correct solutions to real distributed computing problems. It should already be apparent that this will have a major influence on how one *synthesizes* these solutions. In particular, the programs P_0

Note. The entries in the tables have the format ⟨NextState⟩/⟨Action⟩. If the ⟨NextState⟩ part is omitted, it means that the process executes the specified Action but stays in the same state.

and P_1 are direct encodings of the accompanying state tables. It also turns out that this formalism is of great value in performance monitoring [12].

There are some notational conventions that we will follow throughout. In both programs and state tables, the names of states will be in all upper case letters. The names of message classes will be in lower case and special symbols. Other constructs will be capitalized with reserved words in italics. We will abbreviate state and message names frequently.

It is easy to characterize the critical section problem in terms of pairs of states ($\langle$state of $P_0\rangle$, $\langle$state of $P_1\rangle$). The mutual exclusion condition is:

$$\neg(C, C). \tag{1}$$

Let $(X, Y) \Rightarrow (Z, V)$ mean that if the system is in state (X, Y), it will necessarily reach state (Z, V) in finite time. Then the no deadlock condition becomes:

$$(T, T) \Rightarrow (C, \bar{C}) \vee (\bar{C}, C) \tag{2}$$

where $\bar{X}_i$ is the complement of X_i relative to the states of module P_i. Finally, the condition of no lockout is

$$(T, X) \Rightarrow (C, Y) \wedge (X, T) \Rightarrow (Y, C). \tag{3}$$

In this formulation, the desired properties of our solutions are the reachability or nonreachability of certain system states. For the simple example here, one can easily write down the diagram of all reachable states and read off the desired results. It is also easy to write a general program to test for similar properties of a small system of state-message machines [1]. Despite the fact that the example is so simple as to be almost degenerate, much of this discussion will carry over to realistic problems.

For more complex problems, we will need to look carefully at sequences of system states. One critical notion is that of a discrete *event* [13]. For the fixed systems of modules considered here, an event will be the sending or receipt of a message or a change in the internal state of a module. More generally, the creation and destruction of module instances are also events. We will use the notation $(X \leftarrow a, Z)$ to denote the event where P_0 receives a message a while in state X with module P_1 remaining in state Z. An important point is that we make no attempt to impose a uniform time frame on a system of modules. If P_1 could be changing state or receiving a message in parallel with $X \leftarrow a$, we must account for all possible sequences of these events. We will try to develop solutions which can be characterized without including the messages en route (and in queues) as part of the state of the system, but there is no inherent difficulty in including these as well.

With the formulation, we can present an alternative proof of the solution to the critical selection problem illustrated in Figs. 1 and 2.

Alternative proof (based on the state tables). Notice that each module sends a ? only in state T and remains in T until it gets a "yes" response.

No deadlock. If P_0 is in T, it has sent a ?. If P_1 is not in T when it receives the ?, it will queue it. As soon as P_1 enters T, it will send a "yes" response, enabling P_0 to enter C.

No lockout. The argument above shows that P_0 is not locked out. If P_1 is in T, it will remain there until it receives a "yes". But P_0 sends a "yes" when it completes its critical section allowing P_1 to enter C.

Mutual exclusion. If P_1 is in C, it has received a "yes" from P_0 while in T. P_0 sends "yes" only on entering state N. For P_0 to get to C, it must go through T and send a ?. But P_1 will receive the "yes" first and not respend to the ? until its next cycle at TRYING. In addition, there could be no "yes" in P_0's queue because P_1 responds "yes" only once to each ? and P_0 uses that "yes" to enter C.

The general situation here is an elaboration of the coordination problem originally laid out by Petri (cf. [24]). It is fundamentally impractical (because of the finite speed of signals and finite size of elements) to model an arbitrarily large system as changing its global state. The fine structure of how the "state-change" is coordinated must be accounted for. In our model, and in real networks, it is usually impractical to attempt to move the system deterministically through a fixed sequence of states. Although each module (and its characterizing table) is strictly deterministic, the overall system is not. Modules cannot, in general, keep track of the internal state of their correspondents. We can only verify (and understand) systems that have some stable transitions of at least a subset of the modules. The notation $(S, T) \Rightarrow (U, V)$ and its extension to events provides a way of talking about successions of stable points.

Let us now consider the situation where one module can die at any point in the dialogue. We assume that the module is eventually restarted at its beginning and would like the system to continue to perform correctly. Problems like this frequently arise in practice.

There are two assumptions about dying modules, the more intuitive one being:

(A3) When a module dies, its death is reported (by a "dead" message) to all its correspondents. The dead message remains enqueued until a message from the reincarnation of the deceased is received.[1] A module that restarts and attempts to send a message to a module that is dead at that time, receives a "dead" message.

The common way to get the effect of A3 is to have the communication system maintain a continuous "Hello—I Heard You" dialogue at a low level. If some module fails to respond in time to a "Hello", it is declared dead and appropriate actions are taken. There are some delicate questions involving the choice of time-outs and the possibility of a module restarting in the middle, which we will not address here.

(A4) When a module restarts, it receives accurate information on which other modules are alive or dead. Any message to a previous incarnation of a module will be flushed from the system.

Assumption A4 must be met at some level by any correct system, but its inclusion as an underlying system feature is debatable. One common way of discarding messages to previous incarnations is to include a birthtime stamp or incarnation number as part of the module name. This works, but adds overhead to every message. Another way to achieve A4 is to have each module go through a special "starting" procedure which carries out the work needed (for A4). We will first extend the solution of Fig. 2 to dying modules, under the assumption that A4 holds. Later we describe some details of an implementation of A4.

We will also need a variation of the *Receive* (⟨message⟩) which does not suspend the module which executes it. The primitive

Pending (⟨message⟩)

will return *False* if no message is in the queue. If the queue is nonempty, a copy of its

[1] Notice that the RECEIVE primitive when applied to a "dead" message returns a copy of it, and the message remains enqueued (similar to PENDING introduced later).

first element is put into ⟨message⟩ and *True* returned. In the event that *Pending* (⟨message⟩) returns *True* we might be content to proceed with the computation using the copy put into ⟨message⟩. However, to dequeue the original message the explicit primitive *Flush* (⟨message⟩) is provided. Once again following PLITS syntax, we extend both *Receive* and *Pending* to have an optional additional part, "*From* ⟨module⟩", which causes the message reception to be conditional on the Sender.

Modules will have to make assumptions about whether others are dead or not depending on whether or not there is a "dead" from that module in its queue. Of course, this information could be outdated, and we will take this into account. For convenience in writing examples, we will introduce the notation:

$$\textit{Down}\ (\text{Pk}) \equiv (\textit{Pending}\ (\text{message})\ \textit{From}\ \text{Pk}) \wedge (\text{message} = \text{dead}).$$

That is, *Down* (Pk) is true iff there is a dead message from Pk in the queue.

Fig. 3 presents the programs and state-tables for the critical section problem with dying modules under the assumption that A4 holds. The additional program statements are enclosed in set brackets. The additions to the state tables are simply the columns labeled "*d*".

P_0

```
NEUTRAL: ⟨non-critical section⟩
  {If Down (P1) Then Go to CRITICAL}
  Send canI? To P1
TRYING: Receive (message)
  If message = canI? Then
    Begin
      Go to TRYING
    End

CRITICAL: ⟨critical section⟩
  {Unless Down (P1) Then
    Send yes To P1}
    Go to NEUTRAL
```

P_1

```
NEUTRAL: ⟨non-critical section⟩
  {If Down (P0) Then Go to CRITICAL}
  Send canI? To P0
TRYING: Receive (message)
  If message = canI? Then
    Begin
      Send yes To P0
      Go to TRYING
    End
CRITICAL: ⟨critical section⟩
  Go to NEUTRAL
```

P_0

state \ message	?	*y*	*d*	—
T	*T*	*C*	*C*	*T*
C			*N*	*N*/*y*
N	/Q_e		*C*	*T*/?

P_1

state \ message	?	*y*	*d*	—
T	*T*/*y*	*C*	*C*	*T*
C	/Q_e		*N*	*N*
	/Q_e		*C*	*T*/?

FIG. 3. *Extension of Fig. 2 to include "dead" messages.*

We now must show that the solution in Fig. 3 is correct. First observe that the "*d*" column accurately reflects the actions of the two programs. In *N*, a "dead" causes a transition to *C*; in *T*, a "dead" message causes control to skip to *C* and in *C* (for P_0 only) a "dead" message suppresses the sending of a "yes" which would otherwise accompany the transition to *N*.

If both modules remain alive, the situation is unchanged and our previous proof holds. While one module remains dead, the other cycles merrily through its critical and noncritical section. Suppose module P_1 has restarted and sends a "canI?" which eventually arrives at P_0. This will be seen by P_0 in state *N* or *C* (P_0 doesn't enter *T* while P_1 is down). If P_0 is in *N* when it sees P_1's "canI?", it will queue it and go to TRYING in the old way. If P_0 is in *C* when it gets the message, it will simultaneously send a "yes" to P_1 and go to *N*. When P_0 gets to *N*, P_1 will not be Down and so the normal dialogue will be resumed. The situation when P_0 dies is even simpler, because P_1 will always see its first "canI?" while in N and resume the normal dialogue. This shows that Fig. 3 is a correct solution.

Suppose we did not insist that A4 holds. Then P_0 could send a "yes" to P_1 which took a very long time to get to P_1. Meanwhile, P_1 could die and the "dead" message get to P_0's queue. P_0 will, of course, cycle through C and N until it gets a new message from P_1. Suppose P_1 is reborn, sends a "canI?", and waits for a "yes" response. The "yes" sent by P_0 to the previous incarnation could arrive at P_1 before P_1's "canI?" got to P_0. In this case, both P_0 and P_1 would simultaneously enter their critical sections. The problem here is the specious "yes" message to a previous incarnation of P_1. We can outlaw this kind of problem with A4, or account for it with a somewhat more complex solution which doesn't require A4. In this solution, each module has an extra STARTING state. A reborn module sends a "canI? start" and waits for a "yes start" before starting its main body. All non-starting messages are ignored in STARTING state.

In the table below these messages have been abbreviated as "st?" and "ok" respectively. While in the STARTING state "canI?" and "yes" messages are ignored (denoted by /Flush), as these are messages that were intended for an earlier incarnation of the process. A "dead" message may be obtained as a result of trying to send the "canI? start" message. This is interpreted as an accurate account of the other process being down.

It should also be noted that startup activity for both processes is identical. This follows from the fact that no state is saved across failures; i.e., a restarting process has an empty message queue prior to executing startup actions. In the case of distributed databases [21] we shall be dealing with cases where some state is carefully saved across failure. The objective of this example is to demonstrate that the startup actions, coupled with assumptions A1–A3, provide the effect of A4.

state \ message	?	y	d	—	st?	ok
T	▨	▨	▨	▨	T/ok	
C	▨	▨	▨	▨	$/Q_e$	
N	▨	▨	▨	▨	$/Q_e$	
S	/Flush	/Flush	N	$S/st?$	S/ok	N

```
START:
  Send canI? start to P1
  Repeat Forever
  begin
    Receive (message);
    If message = dead then goto NEUTRAL
    else if message = yes start
      then goto NEUTRAL
  end Repeat
```

3. More realistic examples. One problem with all the previous examples is that modules P_0 and P_1 wait much more than they should. Suppose P_0 sends a "canI?" to P_1 while P_1 is in its critical section. P_0 will wait while P_1 finishes its critical section, does its NEUTRAL section, and sends its own message. Recall that we introduced a Boolean construct *Pending* (⟨message⟩) which does no waiting but simply returns *True* if there is a pending message and *False* otherwise.

Now consider the following extra statement for P_1:

```
If Pending (message) then
  If message = canI? then
    begin
      Send yes to P0;
      Flush (message);
    end
  else ⟨ignore it⟩
```

This statement could be put after the critical section or somewhere in the NEUTRAL section of P_1 to enable P_0 to proceed. In fact, one would like to be able to have P_0 proceed *whenever* P_1 is not in its critical section.

More generally, one wants to have a module always capable of receiving a (e.g., emergency) message and providing a simple reply. This behavior could be modeled by a polling mechanism, by interrupts, or in a number of other ways. The fundamental model we will adopt is that each module is a pair of cooperating processes, sharing storage and communicating by a standard *Signal* and *Wait* discipline. Signal, Wait, and the reading and writing of shared variables are assumed to be indivisible. The processes will be called the *Main* processes and the *Receiver* process for obvious reasons. Some similar mechanism will be required for any model in which modules both respond promptly to messages and to independent computation.

Using the Main-Receiver model, we can develop a solution to the critical section problem which is much better than our earlier ones and which will extend to more complex problems. For this solution, each module will have a Main process and a Receiver process which share one variable, *State*, and one "semaphore", *Ok*. The two Receiver processes will each have a variable of type message which can receive either of the two permissible messages, "yes" and "canI?", as well as the system message "dead". The allowable values of *State* are NEUTRAL, TRYING, and CRITICAL, just as in the simple case. In this case, however, the state is stored explicitly in a variable rather than being given implicitly by the program counter. There are two major differences between this and the earlier solution. Each module's Receiver will answer "yes" to "canI?" in NEUTRAL and each module will use a Flag to remember a request that it couldn't honor and then reply "yes" when leaving CRITICAL. Fig. 3 gives the programs and tables.

The extended notation (y|Flag) means that a "yes" will be sent if Flag is *True*. Flag ← True is the operation which does this. Since the tables are to be used as auxiliary devices, one should not be rigid about the choice of notation for entries. It should be noted that Flag is a simple instance of information being sent to Main by the Receiver. It is obvious from the tables and the code that the behavior of P_0 and P_1 differ only upon receipt of a "canI?" message in TRYING state.

A complete proof of the correctness of the solution would include showing that the tables characterize the programs, but we will omit this step. It is not trivial, requiring proofs involving *Signal* and *Wait*, but is straightforward. We will demonstrate that the two state tables satisfy the three conditions of mutual exclusion, no deadlock, and no lockout. The programs presented only work on the assumption that the conditional statements in the main and receiver processes are indivisible with respect to one another; otherwise one would have to introduce more signal–wait pairs to protect the shared variables. (Gary Peterson pointed this out.)

Proof. Notice that each module sends a ? only upon entering *State T* and it remains in *State T* until receiving a "yes" response.

No deadlock: If P_0 is in T, it sent a ?. P_1 responds "yes" to ? unless it is already in its critical section.

No lockout: Both P_0 and P_1 respond to a ? by either an immediate "yes" or by setting Flag. If Flag is set, then they both respond "yes" at the end of their critical section. Thus, neither will enter its critical section twice after receiving a ? without first responding "yes".

Mutual exclusion: Assume that the system has entered the forbidden state (C, C). The immediate preceding event must have been one of the three cases: (1) $(C, T \leftarrow y)$; (2) $(T \leftarrow y, C)$; (3) $(T \leftarrow y, T \leftarrow y)$. We show that none of these events is possible.

```
Main (same for both $P_0$ and $P_1$)
  Begin
    State ← NEUTRAL
  Top: ⟨non-critical code⟩
    Flag ← False
    State ← TRYING
    Send canI? To $P_1$ ($P_0$ respectively)
    Wait ok
      ⟨critical code⟩
      State ← Neutral
  If Flag Then Send yes To $P_1$ ($P_0$ respectively)
  Go to Top
```

```
Receiver of $P_0$

Repeat Forever
Begin
  Receive (message)
  Case State of
  NEUTRAL:
    If message = canI? then
    Send yes to $P_1$
  TRYING:
    If message = canI? then
      Flag ← True
    else
    begin
      State ← CRITICAL
      Signal ok
    end
  CRITICAL:
    If message = canI? then
      Flag ← True
  EndCase
End Repeat
```

```
Receiver of $P_1$

Repeat Forever
Begin
  Receive (message)
  Case State of
  NEUTRAL:
    If message = canI? then
    Send yes to $P_0$
  TRYING
    If message = canI? then
      Send yes to $P_0$
    else
    begin
      State ← CRITICAL
      Signal ok
    end
  CRITICAL:
    If message = canI? then
      Flag ← True
  EndCase
End Repeat
```

state \ message	?	y	—
T	Set Flag	C	T
C	Set Flag		N/ (y|flag)
N	/y		T/?

state \ message	?	y	—
T	/y	C	T
C	Set Flag		N/ (y|flag)
N	/y		T/?

FIG. 4. *Programs and tables for general critical section solution.*

Case (1). P_0 is in C so that it has sent a ? and received a "yes". But P_0 cannot send a "yes" (that P_1 is supposedly receiving) after its ? before leaving C.

Case (2). P_1 is in C so it received a "yes" from P_0 while in T. At the time the "yes" was sent, P_0 was in N. In going to T, P_0 sent a ?. Since the "yes" arrived first, P_1 was in C when it saw the ? and couldn't have sent the "yes" reply.

Case (3). If P_0 sent "yes", it was in N. But if P_0 is in N, it has no pending request to P_1 which could cause P_1 to respond "yes". Once again, the ? sent by P_0 in going to T would follow the "yes" it sent and thus P_1 could not have sent the "yes" that P_0 is supposedly receiving.

There are several additional points to be made here. Although we excluded the consideration of dying modules from the proof, the solution in Fig. 3 can be extended to dying and restarting modules. Both receivers, upon receipt of a "dead" message, treat it as "yes" if in T, set Flag ← False in C, and ignore the message in N. Notice the form of the solution almost totally isolates the Receiver from the Main program; either could be changed significantly without much effect on the other. In particular, one could code Receivers which modeled the state variables of Peterson and Fischer [23]. The messages would be of the form "what is your state?", "it's T", and "it's F", and "it's $\emptyset$".

The solution and proof techniques described above for the two-module critical section problem can be extended to the purely synchronization issues for a wide range of two-module problems. It is not obvious, however, how to handle larger collections of modules or how to prove the correctness of computations that actually do something. Let us first consider the N-module critical section problem.

Several researchers have tackled the distributed mutual exclusion problem, with different objectives. A circulating control token algorithm [19] has been developed for handling failures in a ring, Ellis [10] has reported a multiple copy update algorithm that uses a ring topology; evaluation nets are used as a specification formalism. A distributed algorithm using time-stamps (and distributed clocks) [18] has been presented for the ordered mutual exclusion problem, i.e., requests are granted in the order in which they were initiated.

Peterson and Fischer [23] present a tournament model for their flag-based solution and their ideas carry over to our solution of Fig. 1 or Fig. 4. One can image the N modules in a binary tree of log N levels. At each level, k, pairs of processes compete as P_0 or P_1 depending on ordering induced by process number (or module identifier). For example, process P_i competes as P_1 at level k if the kth bit of the binary representation of i is 1. The winner at each level goes on to the next level and the champion gets to enter its critical section. We will only sketch the message form of this algorithm because it isn't very informative.

The easiest way to extend the solution of Fig. 1 to N modules is to add a new kind of arbiter module for the tournament; the winner of the pair-wise competitions at the bottom level passes its name to the second level arbiter and waits for a "yes". Arbiter modules choose their winners by alternation with lexical preference, just as in Example 1. The top arbiter sends a "go" to the winner. When the winner finishes, it sends a message to its arbiter and the whole arbitration process recycles. With a little more trouble, one can extend the modules of Fig. 1 or Fig. 4 to do the arbitration themselves. The main trick is that you must have the same module always be the arbiter at level l (independent of which won at level $l-1$) because modules must know whom to communicate with. Extending this solution to dying modules isn't too hard, but a module can get locked out if it is always the case that one of its superiors is dead. A solution which avoids this difficulty would involve redundant communication links and would be considerably more complex.

There is an alternative solution to the N-module critical section problem for message systems which can be considered. In this scheme, the modules are assumed to be organized in a ring, with each module normally sending messages to its right. A request by some module to enter its critical section must be approved by all the other modules who might compete for the resource controlled by the critical section. Each module will have a Main-Receiver pair as in Fig. 4, but the Receivers will be more sophisticated. Each will maintain a queue of requests and will forward only the highest priority requests to the next module. We will first present a detailed solution assuming no module dies, but including some extra information for that case, and then extend the solution. From the foregoing discussion, it should be sufficient to describe and prove the solution in terms of state-message tables without explicitly laying out the code for the modules.

The state-message table for the immortal case is given in Fig. 5.

state \ message	?(x)	y	—
T	/Queue (x); Forward	C	T
C	/Queue (x)		N/Finish; Forward
N	/Queue (x); Forward		T/Queue (own)

where

?(x):	Request other than my own.
y:	Return of my request, i.e., ?(own).
Queue(x):	Puts the request x (possibly my own) in queue; duplicates are ignored.
Forward:	Sends all requests in queue that have priority higher than my own request (if any).
Finish:	Deletes own request from queue.
Blank:	Semantics unspecified (discard, report error, etc., are possible choices).

FIG. 5[2]

The critical data structure is the queue maintained by the Receiver of each module. Each element of the queues is a pair

$$\langle\text{Requester, Seq}\#\rangle$$

where Requester is the module number of the initiator of the request and Seq# is defined below. We assume that Seq# is initially zero for all modules. When a module initiates a request, it uses the Seq# one greater than any it has received and/or forwarded.[3] The priority of requests is by lowest Seq#, with ties resolved to favor the lower numbered (lefterly) initiator. Each module keeps a copy of its own request and waits for an input identical to its request ("y" in the table).

Proof for solution of Fig. 5. At each point in time, there is a unique request of highest priority in the system. The crucial fact is that no request of lower priority (higher numbers) will be satisfied before the highest priority request. This follows directly from three facts:

(1) A module initiates lower priority requests than it has passed on;

(2) Each request must pass through every module;

(3) The module that initiated the highest priority request will not pass on any request worse than its own.

It is not true that the instantaneously highest priority request will be the next one executed, because another module can start a better request. (Let us consider the case where modules P_1 through P_N have come up but no requests have been initiated. P_4 then initiates the request (P_4, 1). If P_1 initiates a request later, but before P_4's request reaches P_1, then P_1's request is (P_1, 1). As P_1 has higher priority than P_4, and both

[2] This specification is semiformal; it can be formalized further by providing a formal definition of data structures (including variables), functions, and predicates in base language for each table.

[3] Each module can be postulated to maintain a local integer MaxSoFar, which is initialized to zero. Upon receiving a request MaxSoFar is reset to greater of MaxSoFar and the Seq # of the request. Thus to initiate a request, the module increments its MaxSoFar by one and uses it as the Seq # for the request.

requests have the same Seq#, P_1's request (though temporally later) is granted first.) This can happen at most $N-1$ times, however. We now present a more formal correctness argument.

We define a total ordering $\ll$ on requests. A request is characterized by two pieces of information: the identity of the initiator and a Seq#. Priority decreases with increasing value of Seq#. To handle ties in Seq#, a static ordering is imposed on modules in the ring. Priority is given to the lefterly module, i.e.,

$$P_i > P_{i+1}, \quad \text{where } 1 \leqq i < N \text{ and } i \text{ increases clockwise along the ring.}$$

Now request $R_i(P_i, S_i)$ has a higher priority than $R_j(P_j, S_j)$; i.e., $R_i \gg R_j$ iff

$$S_i < S_j \quad or \quad (S_i = S_j \; and \; P_i > P_j).$$

LEMMA. *A module P_k forwards a request $R_i(P_i, S_i)$ iff one of the following holds:*

- *P_k does not have a request in the ring;*
- *$R_i \gg R_k$ where R_k is the request of P_k currently somewhere in the ring.*

Conversely P_k blocks a request $R_i(P_i, S_i)$ iff P_k has a request R_k in the ring and $R_k \gg R_i$.

LEMMA. *When P_i enters its critical section, i.e., R_i is satisfied, then $R_i \gg R_k$ for all requests R_k present in the ring at that instant.*

Proof. As R_i has successfully circulated through the ring, $R_i \gg R_k$, for all requests R_k of modules that forwarded R_i. However, requests might have been created while R_i was circulating in the ring. Any request initiated in the wake of R_i has a lower priority, as it will have a higher Seq#. A module downstream of R_i could initiate a request of higher priority. If such a request R_k had been initiated then R_i would have been blocked as $R_k \gg R_i$. However, R_i is not blocked so such an R_k was not present.

Mutual exclusion: Let two modules P_i and P_j be in the critical section; i.e., the associated requests R_i and R_j are satisfied simultaneously. This would require that $R_i \gg R_j$ and $R_i \ll R_j$; hence the contradiction.

No deadlock: Deadlock would arise if every request in the system was blocked. As there can be at most N requests in the system, this would require that

$$R_1 \gg R_2 \gg R_3 \gg \cdots \gg R_N \gg R_1.$$

This again implies at $R_1 \gg R_N$ and $R_1 \ll R_N$; hence the contradiction. So R_1 will go through thereby breaking the deadlock.

No lockout: This follows from the argument for deadlock freeness. A request can be blocked at $N-1$ points in the ring. However, upon exiting the critical section, all blocked messages are forwarded *before* the module can put a new request in the ring. Thus a request will have to wait at most for $N-1$ cycles, through critical section, of its superiors.

In order to extend this solution to the case where modules die at arbitrary places, we need one new state, START, and one new message, "Starting(x)", plus the system message "dead". The extended solution is abstracted in Fig. 6. (The upper left-hand $3*3$ array is identical to Fig. 5.)

There is one extra table (or function) needed for each module. This table, Next, is the module's internal model of who is to its right in the chain. This gets updated in the obvious way when a module receives a "starting(x)" or "dead(x)" message. When a living module receives a dead(x) message, it also flushes any request initiated by the deceased module, x. If a module receiving a "dead" has sent out any requests of its own, it sends the request again. Notice that its own request must be the best one in the queue of the initiating module.

	?(x)	y	—	Dead	Starting (x)
T	/Queue (x); Forward	C	T	/Update (x); Retransmit	/Update (x); Forward
C	/Queue (x)		N/Finish; Forward	/Update (x)	/Update (x); Forward
N	/Queue (x); Forward		T/Queue (own)	/Update (x)	/Update (x); Forward
START	/Forward	N	START	/Update (x); Retransmit	/Update (x); Forward

where

?(x): Request other than my own.

y: Return of my own request, i.e., ?(own) or starting(own).

Queue(x): Puts the request X (possibly my own) in queue; duplicates are ignored.

Update(x): Updates "Next" table and flushes requests (if any) for the dead module.

Forward: Sends all requests in queue that have priority higher than my own request (if any); also "Starting" request is provided with a Seq #.

Finish: Deletes own request from queue.

Retransmit: Marks own request as unsent and then Forward.

FIG. 6[4]

Assuming (for the moment) that no module restarts, we can show that the argument for Fig. 5 extends to dying modules. When any module dies, each other module which had sent a request sends it again. There are two cases to consider: the original request was either being held in the dead module or it was not. If a request was lost in the dead module, its copy will proceed as in Fig. 5. Otherwise an equivalent request is introduced in the ring.

These equivalent requests can be viewed as new requests with the same descriptors as that of an earlier request (i.e., the original). Consequently the duplicate elimination assumption in A1 does not rule out equivalent requests (later referred to as "repeats"). The "repeats" chase the original along the ring. If the original gets blocked en route, then the "repeats" catch up and are eliminated via the uniqueness requirement for queues. The other possible cases are:

(i) the initiator dies: "repeats" are flushed upon receipt of the pertinent dead message.

(ii) the initiator is in critical section: "repeats" are eliminated upon being enqueued on the initiator's queue.

(iii) the initiator has exited the critical section and is quiescent: "repeats" are detected, as ?(own) is received while there is no ?(own) in the queue, and discarded.

(iv) the initiator has introduced a new request: the "repeat" ?(own) has a Seq # smaller than that of the ?(own) in the queue, and is discarded.

Hence our solution handles the equivalent requests satisfactorily. Finally, the three properties used in establishing the correctness of Fig. 5 continue to hold.

The problem of reincarnated modules is a little more complex. We don't have to worry about requests from a previous incarnation of x because they are flushed from all queues upon receipt of a dead(x). We do have to ensure that the reincarnated module will get into synchronization with the others before initiating a request for the shared

[4] "Repeats" of ?(x), y and Starting(x) can be created as a consequence of deaths. These are flushed irrespective of the state in which they are received.

resource. This is accomplished by having the new module first circulate a "starting(x)" message. Each module that forwards this message puts in it the highest Seq # it has seen (including here). When this returns to x, the new incarnation is known and in place and can enter state N and beyond. Once again, it is not hard to see that the basic argument for the correctness of Fig. 5 carries over.

Several remarks are in order. Fig. 6 is even more abstracted than previous ones because we did not explicitly put in the initial sending of a "starting" message. To do so would have meant adding a PRESTARTING state which would never be re-entered during execution.

More interestingly, the style of solution used in Fig. 6 can be extended to make only those modules interested in a resource pass on the requests of others. One simply has a module circulate a "starting" message when it is interested in some resource and drop out when it is finished. This means that the requests must circulate only among modules actively seeking a shared resource.

Although we described the values of Seq # as increasing without bound, they need only cycle through the numbers 0 to $2N$. The comparison between Seq #'s would be

$$\text{if (larger} - \text{smaller}) < N, \quad \text{then smaller has priority.}$$

This accounts for the wrapping around and works because all of the Seq #'s in use are within a range of N (usually less) at a given instant. From this, we can see that the length of each message is about $3 \log N$ bits. There are, of course, a wide variety of other efficiency and elegance issues that could be investigated. Notice also that we introduced parameterized message classes in a natural way and that this caused no extra difficulty in the proofs.

This is clearly only a beginning in the study of the distributed computing problem. We have found, however, that even the level of formalization obtained so far has been useful in considering our more applied efforts in the area. The central question is the extent to which the methodology described here precludes good solutions to problems—it is simply too early to know. Judging from the past efforts of others and ourselves, there are probably cases for which this paper is wrong or underspecified.

Most recently, we have been applying these methods to proving correctness of distributed programs for problems like stable marriage [3] and eight queens. One interesting result is the appearance of a division that parallels the partial correctness-termination split in sequential code proofs. Our proofs have one part that shows that the collection of modules will *reach* the desired configuration and a separate argument (and often extra states and messages) for showing that the modules will *know* that the global configuration is correct. Our additional main concern is to integrate this work with conventional verification methods and with the relevant part of the PLITS effort, especially assertions and exception handling.

Acknowledgment. Mike Fischer, Jim Low, Jose Nabielsky, Gary Peterson, and Paul Rovner all found important flaws in earlier versions of this paper.

REFERENCES

[1] A. K. Arya and J. R. Low, SAIL-PLITS *Manual*, Internal Memo, Computer Science Dept., Univ. of Rochester, Rochester NY, January 1979.

[2] E. Ball, J. Feldman, J. Low, R. Rashid and P. Rovner, RIG, *Rochester's intelligent gateway: system overview*, TR5, Computer Science Dept., Univ. of Rochester, Rochester NY, April 1976. Also appeared in IEEE Trans. Software Engineering, vol. SE-2, No. 4 (1976).

[3] C. J. BARTER, *Communications between sequential processes*, TR34, Computer Science Dept., Univ. of Rochester, Rochester NY, November 1978.

[4] P. A. BERNSTEIN, J. B. ROTHNIE, N. GOODMAN AND C. A. PAPADIMITRIOU, *The concurrency control mechanism of SDD*-1, IEEE Trans. Software Engineering, 3 (1978), pp. 154–168.

[5] G. V. BOCHMANN AND J. GECSEI, *A unified method for the specification and verification of protocols*, Information Processing 77, B. Gilchrist, ed., 1977, pp. 229–234.

[6] T. H. BREDT AND J. MCCLUSKEY, *Analysis and synthesis of control mechanisms for parallel processes*, in Parallel Processor Systems, Technologies and Applications, L. C. Hobbs, ed., Spartan, Washington, DC, 1970, pp. 287–296.

[7] J. E. BURNS, M. J. FISCHER, N. A. LYNCH, P. JACKSON AND G. L. PETERSON, *Data requirements for implementation of N-process mutual exclusion using a single shared variable*, School of Information and Computer Science, Georgia Institute of Technology, GIT-ICS-79/02, May 1979.

[8] E. E. COFFMAN, M. J. ELPHICK AND A. SHOSHANI, *System deadlocks*, Computing Surveys, vol. 3, No. 2 (1971), pp. 67–78.

[9] E. W. DIJKSTRA, *Solution of a problem in concurrent programming control*, Comm. ACM 8 (1965), p. 569.

[10] C. A. ELLIS, *Consistency and correctness of duplicate database systems*, Proc. 6th ACM Symp. on Operating Systems Principles (1977), pp. 67–84.

[11] J. A. FELDMAN, *High level programming for distributed computing*, Comm. ACM, 22 (1979), pp. 353–368.

[12] I. GERTNER, *Performance evaluation of communicating processes*, Proc. Conf. on Simulation, Measurement, and Modelling of Computer Systems, Boulder, Colorado, August 1979.

[13] C. HEWITT AND H. BAKER, *Laws for communicating parallel processes*, Information Processing 77, B. Gilchrist (ed.), August 1977.

[14] C. A. R. HOARE, *Communicating sequential processes*, Comm. ACM, 21 (1978), pp. 666–677.

[15] R. E. KAHN AND W. R. CROWTHER, *Flow control in a resource-sharing computer network*, IEEE Trans. Comm., COM-20 (1972), pp. 539–546.

[16] G. KAHN AND D. B. MACQUEEN, *Coroutines and networks of parallel processes*, IFIP Congress Proceedings, 1977.

[17] L. LAMPORT, *A new solution of Dijkstra's concurrent programming problem*, Comm. ACM, 17 (1974).

[18] L. LAMPORT, *Time, clocks, and the ordering of events in a distributed system* Comm. ACM, 21 (1978), pp. 558–565.

[19] G. LELANN, *Distributed systems—towards a formal approach*, Information Processing 77, B. Gilchrist, ed. (1977), pp. 155–160.

[20] J. M. MCQUILLAN, W. R. CROWTHER, B. P. COSELL, D. C. WALDEN AND F. F. HEART, *Improvements in the design and performance of the ARPA network*, Proc. of Fall Joint Computer Conf., AFIPS Press, 1972, pp. 741–754.

[21] A. NIGAM, *Specification formalisms and proof methodologies for distributed database algorithms*, forthcoming Ph.D. thesis, Computer Science Dept., Univ. of Rochester, expected Nov. 1980.

[22] S. OWICKI AND D. GRIES, *Verifying properties of parallel programs: an axiomatic approach*, Comm. ACM., 19 (1976), pp. 279–285.

[23] G. L. PETERSON AND M. J. FISCHER, *Economical solutions for the critical section problem in a distributed system*, Proc. SIGACT Conf., Boulder, Colorado (1977).

[24] J. L. PETERSON, *Petri nets*, Computing Surveys, 9 (1977), pp. 223–252.

[25] R. L. RIVEST AND V. R. PRATT, *The mutual exclusion problem for unreliable processes: preliminary report*, Proc. 17th Annual Symp. on Foundations of Computer Science, 1976.

[26] C. A. SUNSHINE, *Formal techniques for protocol specification and verification*, IEEE Computer (Sept. 1979), pp. 20–27.

[27] D. C. WALDEN, *A system for interprocess communication in a resource sharing computer network*, Comm. ACM, 15 (1972), pp. 221–230.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

EFFICIENT ON-LINE CONSTRUCTION AND CORRECTION OF POSITION TREES*

MILA E. MAJSTER† AND ANGELIKA REISER‡

Abstract. This paper presents an on-line algorithm for the construction of position trees, i.e., an algorithm which constructs the position tree for a given string while reading the string from left to right. In addition, an on-line correction algorithm is presented which—upon a change in the string—can be used to construct the new position tree. Moreover, special attention is paid to computers with small memory. Compactification of the trees and transport costs between main and secondary storage are discussed.

Key words. Pattern matching, position tree, on-line algorithm

Introduction. Text editing systems, symbol manipulation problems, as well as a number of other computer applications, often require a search function which locates instances of a given string within a larger main string (P1). In some applications all positions, in others the leftmost position, have to be found. Other pattern matching problems are to search for the occurrences of the elements of a set of pattern strings within a given main string s (P2), to find the longest repeated substring of the main string s (P3), the internal matching problem, i.e., to find for each position i in $s = s_1 \cdots s_n$ another position j in s such that the common prefix of $s_is_{i+1} \cdots s_n$ and $s_js_{j+1} \cdots s_n$ is not shorter than the longest common prefix of $s_i \cdots s_n$ and $s_k \cdots s_n$, $k \neq i$, $k \neq j$ (P4). For the external matching problem (P5) we consider two strings, $s = s_1 \cdots s_n$ and $s' = s'_1 \cdots s'_m$, and a position i in s, and search for a position j in s' such that the longest common prefix of $s_i \cdots s_n$ and $s'_j \cdots s'_m$ is not shorter than the longest common prefix of $s_i \cdots s_n$ and $s'_k \cdots s'_m$, $k \neq j$. Another problem is concerned with finding the longest common substring of two strings s and s' (P6).

A naive algorithm for the solution of problem P1, where all possible alignments are tried successively, takes $O(n \cdot m)$ steps, where n is the length of the main string and m the length of the pattern. In 1970 [3] showed how to solve P1 in time proportional to $(n+m)$. This algorithm basically involves a preprocessing of the pattern string in order to construct a table which stores information of how far the pattern can be shifted against the main string if a mismatch at position j in the pattern occurs. This table is then used as a data structure in the algorithm. If we consider P2 where the k patterns $p_1, \cdots, p_k$ are searched for in the same string s consecutively, the above algorithm would take $O(k \cdot |s| + |p_1| + \cdots + |p_k|)$. This is one of the reasons which led in [5] to the development of an auxiliary data structure, the prefix tree, storing information about the main string. The prefix tree is called a position tree in [1]. Once a compacted version of the tree is constructed in $O(n)$ steps, where n is the length of the main string, a single search for a pattern costs time linear in the length of the pattern.

In [4] a similar data structure, serving the same purposes as the position tree, i.e., to reduce the complexity of pattern matching algorithms, is introduced. In addition this data structure is space economical.

At this point it is now important to note that the position tree construction algorithm processes the main string s from right to left. This feature presupposes that the whole text must be known before we can start to build the position tree. Similarly, another solution for the pattern matching problems P1, P2, [2] presupposes that at least the length of the main string and the set of patterns are known in advance.

* Received by the editors August 31, 1978, and in revised form November 30, 1979.

† Technische Universität, Institut für Informatik, 8 Munich, West Germany.

‡ Technische Universität, Institut für Informatik, 8, Munich, West Germany. The work of this author was supported by the Sonderforschungsbereich 49 in Munich-Programmiertechnik of the Deutsche Forschungsgemeinschaft.

If we have to wait with the construction of the position tree until the whole main string is known, we must face some considerable drawbacks: (D1) It is not possible to answer pattern matching problems and perform corrections, if necessary, for that part of the main string which has been already read in. This is particularly annoying if we consider, for example, a text editing system where the pattern matching is used to find those positions in the text which have to be corrected. Here, one would like to process that part of the string which is already known. In a typical text editing system with a usually small computer dedicated to the text editing job, we must face further considerable drawbacks; namely, (D2) the processing unit keeps waiting until the input device has scanned the last symbol of the input, (D3) as the main store will usually be too small, a considerable part of the text has to be transported on to secondary storage until it is going to be processed. The position tree construction algorithm of [5] constructs the position tree for $s_i \cdots s_n$ from the position tree for $s_{i+1} \cdots s_n$. In general the position tree T_{i+1}, for $s_{i+1} \cdots s_n$, will be too large to be kept as a whole in the main store. So we have to decompose the tree and shift parts of it to secondary store. Unfortunately, as the letter s_i will not be known in advance, one cannot predict which parts of the tree T_{i+1} will be needed for the construction of the tree T_i, for $s_i \cdots s_n$. Therefore, we have the problem (D4) of transferring considerable amounts of data between main and secondary storage. Hence, we are looking for a possibility to construct the position tree in an on-line way. Moreover, we are interested in answering pattern matching questions and in the possibility of "updating" or "correcting" that part of the string which has been already read. And last, we want to get rid of problem D4.

To solve D1 and D2 there is an immediate solution. Instead of treating the string $s = s_1 \cdots s_n$, we consider the reversed string $s_n \cdots s_1$ for which the position tree can be constructed from right to left as usual. This solution is hardly acceptable for two reasons. First, it does not solve D3 and D4. Second, it can be only used under the assumption that—for the problems P1, P2—the pattern can be reversed before the request. It is then not possible to start a search before a pattern is completely given. In particular, requests which are looking for a prefix p_1 of the pattern $p = p_1p_2$, and if this can be located, ask for the rest of the pattern, cannot be treated in this way. Moreover, the pattern matching problems P4, P5 cannot be solved with the position tree for the reversed string.

1. Preliminaries. In this paper we will use the following notation. An *alphabet* Σ is a finite set of symbols. A *string* over an alphabet Σ is a finite-length sequence of symbols from Σ. The empty string denoted by ε is the string with no symbols. If x and y are strings, then the *concatenation* of x and y is the string xy. If xyz is a string, $x, y, z \in \Sigma^*$, then x is a *prefix*, y a *substring* and z is a *suffix* of xyz. The *length* of a string x, denoted by $|x|$, is the number of symbols in x.

A *position* in a string of length n, $n \geqq 1$, is an integer between 1 and n. The symbol $a \in \Sigma$ *occurs* in position i of string x if $x = yaz$, with $|y| = i - 1$. Let $\$ \notin \Sigma$.[1]

A *position identifier* for position i in $x\$$[2] is the shortest substring u of $x\$$ such that

(i) $x\$ = yuz$, $|y| = i - 1$;

(ii) if $x\$ = y'uz'$, then $y = y'$, $z = z'$.

A Σ-*tree* is a labeled tree T such that for each node N in T the edges leaving N have distinct labels in Σ. If the edge (N, M) in T is labeled by a, we call M the a-son of N.

A *position tree* for a string $x\$ = x_1 \cdots x_{n+1}$, where $x_i \in \Sigma$, $1 \leqq i \leqq n$, is a $(\Sigma \cup \{\$\})$-tree T such that

[1] We use \$ as an endmarker for strings over Σ.

[2] The endmarker is needed to guarantee the existence of a position identifier for each position.

(i) T has $n+1$ leaves labeled $1, \cdots, n+1$. The leaves of T are in one-to-one correspondence with the positions in $x\$$.

(ii) The sequence of labels of edges on the path from the root to the leaf labeled i is the position identifier for position i.

Note that there is exactly one position tree for each string.

Example. The position tree for the string $abba\$$ is given in Fig. 1.

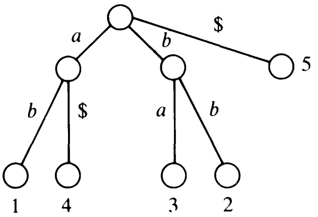


FIG. 1

2. On-line construction of position trees. The problem which we are going to solve in the following is: let a string $x = x_1x_2 \cdots$ be read from left to right without knowing the whole string in advance and construct after the reading of each letter the position tree for the actual prefix. The problem with the construction of the position tree when reading the text from left to right is based on the fact that we need an endmarker \$ for each string in order to guarantee that there exists a position identifier for each position in the string. This has the consequence that reading from left to right means the transition from

$$x_1x_2 \cdots x_i\$ \quad \text{to} \quad x_1 \cdots x_ix_{i+1}\$.$$

The efficiency of Weiner's algorithm stems from the fact that the changes which are caused by updating the tree reading the text from right to left are "local." If we work from left to right, changes are no longer "local." In particular, we have to solve the following two problems:

(i) A position identifier may become invalid by reading a new symbol, as, e.g., the position identifier for position 1 in

$$abcb\$ \rightarrow abcba\$.$$

(ii) All position identifiers which contain the endmarker have to be changed whenever a new symbol is read in, e.g., the position identifier for position 4 in

$$abcb\$ \rightarrow abcba\$.$$

In the following we describe how to construct the position tree for a string $xa\$$ from the position tree for $x\$$, where $x \in \Sigma^*$, $a \in \Sigma$.

Algorithm: Position tree on-line.

(1) For *each node* N that has a \$-son N' the following steps have to be performed. The order in which the nodes with \$-sons have to be processed is given in Lemma 1 below.

(a) If N does not have an a-son, then replace the \$-symbol by a.

(b) If N has an a-son N'' that is not a leaf then remove the edge between N and N' and make N' the \$-son of N'' (together with the position number of N').

(c) If N has an a-son N'' that is a leaf, then remove the edge between N and N' and make N' the \$-son of N'' together with the position number associated with N'. Moreover, attach a new son to N'', transfer the position number j of N'' to the new son; label the edge between N'' and its new son by the $(j+l)$th letter in $xa\$$, where l is the length of the position identifier for position j in $x\$$.

(2) Attach a \$-son at the root and give it the next position number.

Example. Consider the string $x\$ = abbab\$$. We construct the position tree for $xa\$$ from that for $x\$$. The position tree for $x\$$ is shown in Fig. 2.

The father of leaf 4 falls into case (a). The father of leaf 5 falls into case (c), the father of leaf 6 falls into case (b). Performing the algorithm for the father of leaves 4, 5, 6 (in that order) yields Fig. 3.

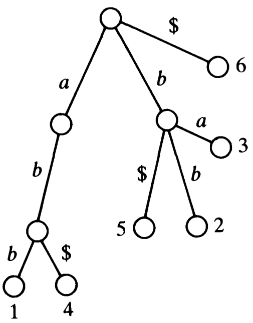


FIG. 2

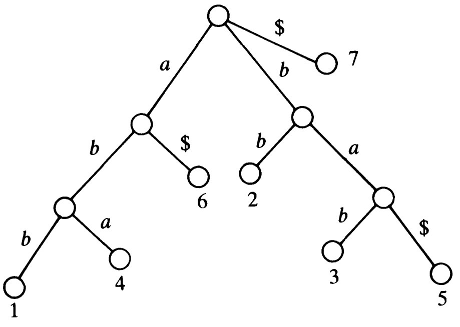


FIG. 3

We now want to make sure that the algorithm works correctly.

LEMMA 1. *Assume that step* 1 *of the above algorithm is performed successively for all nodes N with a* \$*-son in such a way that if a node is processed then all its descendants have been processed previously. Then the algorithm constructs the position tree for* $xa\$$ *from the position tree for* $x\$$.

Proof.

1) Let us first make sure that for each node M in the tree constructed by the algorithm and for each $\sigma \in \Sigma$ there is at most one edge with label σ starting in M. This is true by the observation that steps 1a, 1c evidently result in a tree with the requested property. For step 1b we must show that N'' does not have a \$-son before this step is performed. This is guaranteed by the fact that N'' is a descendant of N and is—by assumption—processed before N. Step 1 removes in each case the \$-edge from the considered node. Hence, step 2 also results in a tree with the requested property.

2) The next fact to be verified is that for each position identifier which is affected by the new letter a there is a change in the tree reflecting this change. There are three possibilities for a position identifier to be affected.

(a) The endmarker \$ is part of the position identifier. Let $x_i \cdots x_m\$$ be the position identifier for position i in $x_1 \cdots x_m\$$; hence there must be positions j_h, $1 \leqq h \leqq k$, with position identifier $x_ix_{i+1} \cdots x_m r_{j_h}$, $1 \leqq h \leqq k$, $r_{j_h} \in \Sigma^*/\{\varepsilon\}$. If $r_{j_h} \notin \{a\}\Sigma^*$, for all h, $1 \leqq h \leqq k$, the new position identifier for position i will be $x_i \cdots x_m a$, which is achieved by step 1a of the algorithm. If there is h_o, $1 \leqq h_o \leqq k$, where $r_{j_{h_o}} = as_{j_{h_o}}$, the position identifier for i becomes $x_i \cdots x_m a\$$ by step 1b or 1c.

(b) For a position j with identifier $x_j \cdots x_{j+r-1}a$, there is a position i with identifier $x_i \cdots x_m\$$ such that $x_j \cdots x_{j+r-1} = x_i \cdots x_m$. The position identifier for position j must be prolonged to $x_j \cdots x_{j+r+1}$ which is achieved by step 1c.

(c) The position identifier for a position i in $xa\$$ starts with the letter a. Then either a occurs only once in $x\$$ then the position identifier for i is obtained by step 1c, or a occurs more than once in $x\$$ then the position identifier for position i is obtained by 1b, or a does not occur in $x\$$ then the position identifier for the new letter a is obtained by step 1a.

LEMMA 2. *Let $L = N_1, \cdots, N_k$ be a list of all nodes that have a \$-son in the position tree for $x\$$, but ordered such that $(*) N_i$ is a descendant of $N_j \Rightarrow i < j$. If we perform the algorithm processing the nodes that have a \$-son in the order given by L, then we can sequentially update L to get a new list L' that contains all nodes that have a \$-son in the position tree for $xa\$$ and that fulfills $(*)$. The cost for constructing L' from L is $O(k)$.*

Proof. Let $L = N_1, \cdots, N_k$ be a list for the position tree for $x\$$ fulfilling $(*)$. Let us perform step 1 of the algorithm according to this list; i.e., we start with N_1, continue with N_2 and so on. To manipulate the list we perform the following steps.

(a) If step 1a is performed with N_i just remove N_i from the list.

(b) If step 1b is applied to N_i then replace N_i by its a-son.

(c) If step 1c is applied to N_i then replace N_i by its a-son.

(d) If step 2 is performed attach the root at the end of the list.

Let L' be the list obtained by the above steps. Let N, M be two nodes with a \$-son in the tree for $xa\$$. Let $M \neq$ root and N be a descendant of M. Then the father of N is a descendant of the father of M. The only possibility for a node (except the root) to be inserted into the list is to be a substitute for its father. Hence the list L' fulfills $(*)$. The cost for updating L is obviously $O(k)$ as each node has to be processed and the cost for each node is constant.

3. Costs of the algorithm. In order to be able to analyze the costs of the algorithm we assume that

(1) we hold the text that has been already read in an array,

(2) we represent the tree in the following form.

Each node is represented by a natural number; in particular the root is represented by 0. We associate with each node three fields. The first contains the position number m, if the node is a leaf corresponding to position m. If the node is not a leaf then this field contains a list of the sons[3], each given by its number and the label of the edge leading to it. The second field contains information about the depth of the node. The depth of the root is 0. The third field serves for linking those nodes that have a \$-son into a list. For example, the tree for $abbabb\$$ is shown in Fig. 4, which is represented by Table 1.

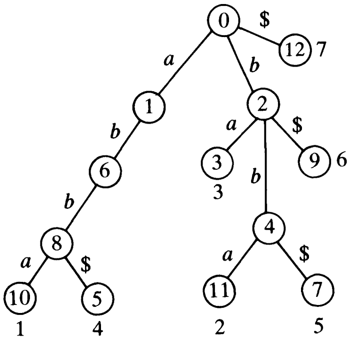


FIG. 4

[3] We will assume that the \$-son—if any—is always the first in the list of sons.

TABLE 1

	NR	son or p.n.	depth	successor in list
	0	(12, \$) (1, a), (2, b)	0	—
	1	(6, b)	1	—
	2	(9, \$), (3, a), (4, b)	1	0
	3	3	2	—
	4	(7, \$), (11, a)	2	2
	5	4	4	—
	6	(8, b)	2	—
	7	5	3	—
Head of list→	8	(5, \$), (10, a)	3	4
	9	6	2	—
	10	1	4	—
	11	2	3	—
	12	7	1	—

Based on this representation the costs for step 1a, 1b, 1c for each node in the list is O(number of sons), as we first check if the node has an a-son. The test whether or not a node is a leaf and the update can be done in constant time; in particular the search for the letter following the position identifier in the text for step 1c can be performed by selecting the $(m+h)$th component of the array where m is the position number of the leaf and h its depth. Hence step 1 takes for each node $O(|S|)$, where S is the set of its sons. Step 2 costs constant time. Hence, the cost for constructing the position tree for $xa\$$ from the position tree for $x\$$ can be bounded above by $O(\sum_{i=1}^{k}|S_i|+1)$, where S_i is the set of sons of the ith node N_i in the list $L = N_1, \cdots, N_k$. This cost can be bounded above by $O(|L|\cdot|\Sigma|+1)$. The cost of the on-line construction of the position tree for $x\$$, $x \in \Sigma^{n-1}$, can be given by $O((\sum_{i=1}^{n}|p_i(x)|)\cdot|\Sigma|)$, where $p_i(x)$ is the position identifier for the position i in $x\$$. This is based on the fact that for each position i the identifier $p_i(x)$ has to be updated as often as $|p_i(x)|$. Hence, the father of the leaf with position number i can occur at most as often as $|p_i(x)|$ in the disjoint union of all lists $\bigcup L_j$, where L_j is the list of nodes with \$-sons in the jth application of the algorithm.

In terms of the average length l_x for a position identifier in the string $x\$$, $x \in \Sigma^{n-1}$, i.e.,

$$l_x = \frac{\sum_{i=1}^{n}|p_i(x)|}{n},$$

the cost of the algorithm for constructing the position tree for $x\$$ can be given by $O(l_x\cdot n\cdot|\Sigma|)$.

4. Compactification. Concerning the space used by the algorithm we note that the position tree for a text of length n may have $O(n^2)$ vertices, as can be seen from the example $a^ib^ia^ib^i\$$. However, one can show that there is a *compacted* form of the position tree which needs only $O(n)$ space [5]. Here, compacted means that successive edges corresponding to single sons are contracted into one edge named by a string, as shown in Fig. 5.

The question how such string-labeled trees can be efficiently represented is discussed in [4].

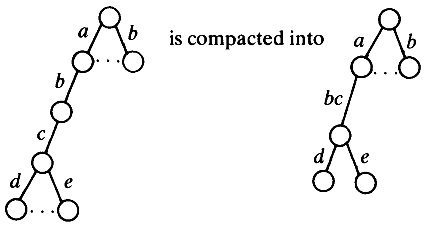


FIG. 5

The compacted position tree can be constructed in the same way as the noncompacted one. This can be seen as follows: if we start with an already compacted tree for $x\$$ and want to construct the compacted tree for $xa\$$, then at most those nodes which are in the list L are candidates for compactification.

We only sketch the algorithm for the on-line construction of compacted position trees and do not want to go into implementation and analysis details.

The following algorithm takes as input a compacted tree T for $x\$$. Each node in the tree T that has a single son is marked. In steps 1 and 2 the algorithm manipulates the tree basically in the same way as in the noncompacted case. In addition we have to take care of the marks: a mark has to be removed from a node if this node gets a second son and a mark has to be attached to a node if the node has lost all but one son. After the steps necessary to reflect the new identifiers in the tree there may exist two successive nodes with marks, which means that the tree is no longer compact and has to be compactified in step 3.

Algorithm: compacted position tree on-line.

(1) For each node N that has a \$-son N' do: (The order in which the nodes with \$-sons are processed is given in Lemma 1.)

  (a) *If N does not have an ar-son, $r \in \Sigma^*$,* then

    (a1) change the label of the edge (N, N') from \$ to a.

  (b) *If N has an ar-son N'', $r \in \Sigma^*$,* then

    (b1) remove the edge (N, N'),

    (b2) mark N by * if there is exactly one son left,

    (b3) *if $r \neq \varepsilon$* (this implies that N'' is not a leaf), then create a new node NN, make the sons of N'' the sons of NN, make NN the r-son of N''; if N'' was marked by * transfer the mark to NN, change the label of the edge (N, N'') from ar to a; *if $r = \varepsilon$ and N'' is not a leaf,* then remove mark at N'', if any; *if $r = \varepsilon$ and N'' is a leaf,* then attach a new son to N'', transfer the position number j of N'' to the new son, label the edge between N'' and its new son by the $(j+l)$th letter in $xa\$$(where l is the length of the position identifier for position j in $x\$$).

    (b4) attach N' as \$-son to N'' (together with the position number of N').

(2) Attach a \$-son to the root, give it the next position number, remove mark at root, if any.

(3) For all pairs of adjacent nodes marked by * compactify.

Let us briefly consider how we can find two adjacent nodes marked by *. New marks are only introduced in step (1.b2) at the node N if it has only one son, namely N''. If N'' was marked, the mark is either removed or transferred because N'' gets a new son, N'. Therefore, only if the father of N was marked, we get two adjacent marked nodes by

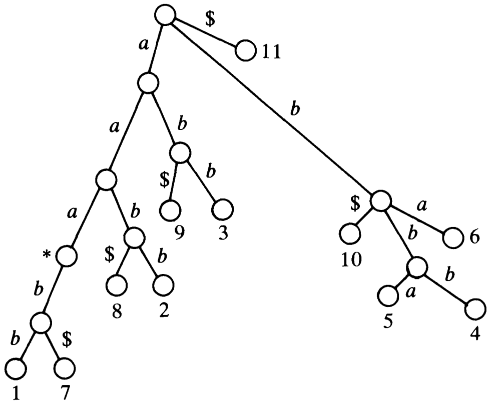


FIG. 6

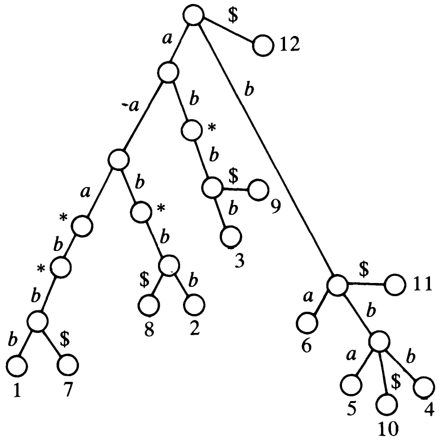


FIG. 7

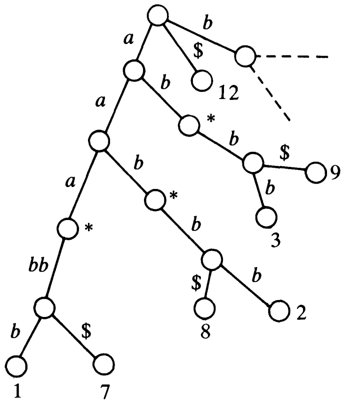


FIG. 8

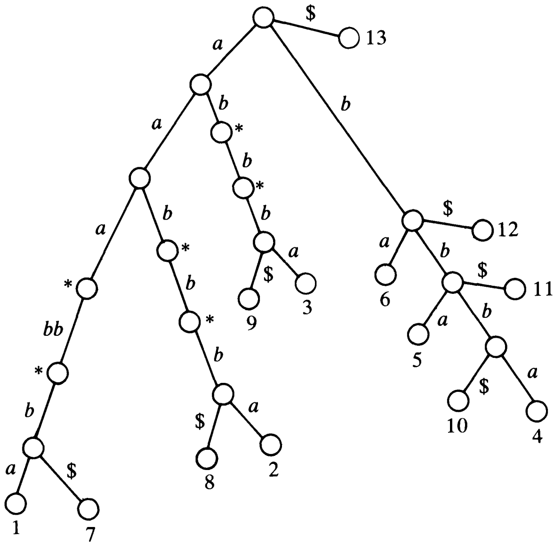


FIG. 9

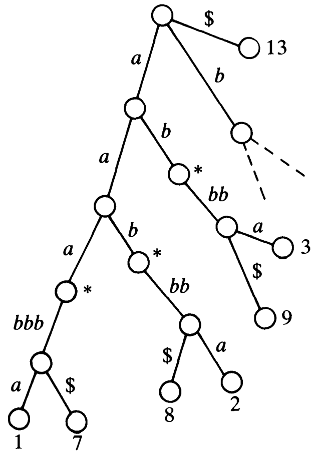


FIG. 10

the introduction of new marks. Existing marks are only transferred to new created sons which cannot result in two adjacent marked nodes. Thereby it is sufficient to maintain for each node which has a \$-son (members of the list L) the father if it is marked. This information can be easily obtained at the time when a node becomes a member of the list because it gets there as a substitute for its father.

Example. We start with the compacted tree for $a^3b^3a^3b\$=x\$$. We mark nodes with single sons by *, see Fig. 6, and read the letter b; i.e., we consider $xb\$=$ $aaabbbaaabb\$$. Steps 1 and 2 of the algorithm yield Fig. 7.
Then we compactify and get Fig. 8. We continue with $xbb\$=a^3b^3a^3b^3\$$. Steps 1 and 2 yield Fig. 9. Then we compactify to get Fig. 10.

5. Main store and transport cost. Let us now consider the problem D4; i.e., given a small computer, we consider the question how we can keep the cost of main store low and how we can reduce cost for transport between main and secondary storage. We shall assume a paging system in the following.

Let us first see which parts of the tree are actually involved if we construct the position tree for $xa\$$ from that for $x\$$. First we need all nodes N with \$-sons together with the information which sons N has. Second, if N has an a-son, this son is needed, and third, the \$-son is needed.

A first but, as we will see, not efficient approach to our problem could be to store the tree structure without the links for the list L in secondary store and to hold the information about the nodes with a \$-son in main store. This is done by maintaining a list of references to the nodes with \$-son. The ordering given by this list should correspond to the one given by L.

If we look closer at this solution we find that the following steps have to be performed:

(1a) If a \$-son of N is transformed into an a-son
  (i) the node N must be removed from the list. Its predecessor in the list must be linked with its successor. This step does not cause any page transfer because L is held in main store.
  (ii) in the list of references to sons of N the \$ must be replaced by an a. Hence the page containing the list of references to sons of N has to be rewritten.

(1b) If a \$-son has to be transferred from N to its a-son which is not a leaf, we have
  (i) to change the list of references to sons of N;
  (ii) to change the list of references to sons of the a-son;
  (iii) to change the height of the \$-son of N;
  (iv) to change the reference from the predecessor of N to N (in the list of nodes with \$-sons) into a reference from the predecessor of N to the a-son of N;
  (v) to erase the reference of N to its successor;
  (vi) to write a reference from the a-son of N to the successor.

Hence, the page containing the list of references to sons of N, the page containing the list of references to the sons of the a-son of N and the page containing the \$-son of N have to be rewritten. In addition the list in main store is manipulated.

(1c) If a \$-son has to be transferred from N to its a-son which is a leaf we have
  (i)–(vi) to change as above;
  (vii) to read one of the pages on which the text is situated;
  (viii) to create a new son for the a-son of N.

Here, in addition to the changes of (1b), we have to read one text page, to write the new node on some free space and to insert a reference to this new node into the list of references to the sons of the a-son of N.

If we assume that the references to the sons of a node are all stored on one page, (1a) causes 1 page transfer, (1b) may cause 3 page transfers, and (1c) may cause 4 page transfers, disregarding the cost of writing the new node. In order to reduce the number of page transfers we propose to modify slightly the representation of the position tree in the following way:

A We do not store the \$-sons in the tree representation but keep them separately.

B For each \$-son we store a reference to its father.

C Instead of maintaining the list of references to nodes with \$-sons, we maintain a list $L_\$$ of \$-sons which is kept in main store.

With this choice we get for our example *abbabb*\$, Fig. 11 and Table 2, plus the list in Table 3.

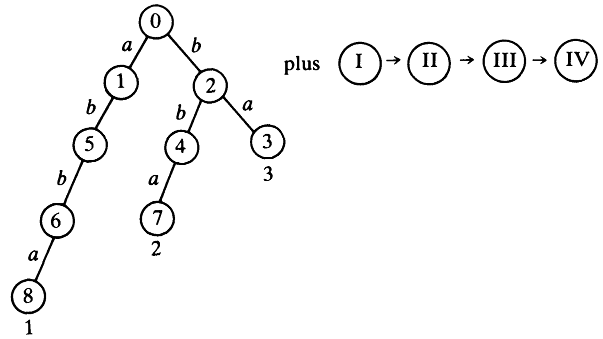


FIG. 11

TABLE 2

Nr.	references to sons or p. n.	depth
0	$(1, a)(2, b)$	0
1	$(5, b)$	1
2	$(4, b)(3, a)$	1
3	3	2
4	$(7, a)$	2
5	$(6, b)$	2
6	$(8, a)$	3
7	2	3
8	1	4

TABLE 3

Nr.	father	next	pos. nr.
I	6	II	4
II	4	III	5
III	2	IV	6
IV	0	—	7

Assuming this representation we review the page transports for our algorithm.

As we keep the list of \$-sons in main store, we must for each element in the list $L_\$$ check if its father has an a-son. This means that we must bring the page p containing this information. Then

(1a) If the \$-son s of N is changed into an a-son, s is removed from the list $L_\$$ in the main store and attached as an a-son to N. As we already brought in the page containing the list of references to sons of N, we try to store s on this page, if possible. If this is not possible we must bring a page with free space and write s on it.

(1b) If the \$-son s of N is transferred to the a-son of N which is not a leaf, we just look on the page p for the reference to this a-son and modify the reference to father of s. This step does not cause any additional page transport.

(1c) If the \$-son s of N is transferred to the a-son of N which is a leaf, we proceed as in (1b). In addition we must read one of the pages on which the text array is situated. As one may easily see, (1a) causes 1 page transfer, (1b) causes 1 page transfer, and (1c) causes 2 page transfers, again disregarding the cost of writing a new node and assuming that all son references of a node can be stored on one page. Only in (1a) do we have to write on a page. The pages which are transferred for (1b) and (1c) are read only. Compared with the previous solution we have gained by reducing the number of page transfers and by reducing the number of write accesses.

6. Correction algorithm. The substitution of a substring within a main string is a common operation in string manipulation systems. Hence, we are interested in a simple algorithm that performs this substitution and manipulates the position tree in the corresponding way. A correction algorithm for suffix trees has been given in [4]. This algorithm is rather complicated and presupposes a link structure which is constructed while the tree is built up by the tree construction algorithm [4]. In the following we are going to describe an algorithm which constructs the position tree for $x\gamma r\$$ from that for $x\beta r\$$, x, γ, β, $r \in \Sigma^*$, $\beta \neq \varepsilon$. Before we present the algorithm we have to say some words about position numbers. If the lengths of γ and β are different, then not only are position identifiers invalidated but also the position numbers for the text that follows β are affected. To circumvent the need to renumber the text that follows β we will assume in the following that our string positions are numbered in a Dewey-Decimal scheme as follows: let a string

$$\begin{array}{ccccc} a & b & b & c & d \\ 1 & 2 & 3 & 4 & 5 \end{array}$$

with annotated position numbers be given. Let us substitute bb by efg. The result is

$$\begin{array}{cccccc} a & e & f & g & c & d \\ 1 & 2 & 3.1 & 3.2 & 4 & 5 \end{array}\;.$$

A substitution of f by hl yields

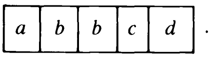
$$\begin{array}{ccccccc} a & e & h & l & g & c & d \\ 1 & 2 & 3.1.1 & 3.1.2 & 3.2 & 4 & 5 \end{array}\;.$$

Substituting c by the empty word yields

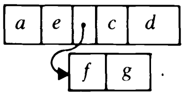
$$\begin{array}{cccccc} a & e & h & l & g & d \\ 1 & 2 & 3.1.1 & 3.1.2 & 3.2 & 5 \end{array}\;.$$

The position number scheme corresponds to the following string scheme for the text array.

a	b	b	c	d	is stored as
1	2	3	4	5	

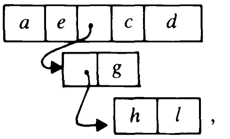


a	e	f	g	c	d	is stored as
1	2	3.1	3.2	4	5	



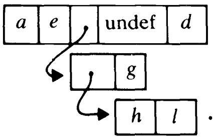
Next,

a	e	h	l	g	c	d	is given in
1	2	3.1.1	3.1.2	3.2	4	5	



and finally

a	e	h	l	g	d	is given in
1	2	3.1.1	3.1.2	3.2	5	



In the following we describe an algorithm which takes the string $x\beta r\$$, $x, \beta, r \in \Sigma^*$, $\beta \neq \varepsilon$,[4] the position tree for $x\beta r\$$ and the string $\gamma \in \Sigma^*$ as input and constructs the position tree for $x\gamma r\$$. For simplicity we will first assume that the position numbers for the input string are the natural numbers $1, 2, \cdots, |x\beta r\$|$. The position numbers of the corrected string will be Dewey-Decimal numbers. We denote the sequence of Dewey-Decimal numbers for $x\gamma r\$$ by $d_1, d_2, \cdots, d_{|x\gamma r\$|}$.

The algorithm makes use of a marking operation. This operation will be restricted to nodes that are different from the root. In addition we will use the following notation:

$$N_i = \text{the node that carries position number } i,$$

$$h_i = \text{height of } N_i, \qquad h(N) = \text{height of } N,$$

$$|\beta| = m, \qquad |\gamma| = l, \qquad |x| = n.$$

Let us briefly explain how the algorithm works. The algorithm performs seven steps. In

[4] The restriction is only introduced to avoid a large number of subcases in the algorithm. The restriction is not harmful as the transition from $x_1 \cdots x_n r\$$ to $x_1 \cdots x_n \gamma r\$$ can be always treated by considering the substitution of x_n by $x_n\gamma$ or by taking a right context if $n = 0$.

step 0 the leftmost position i, if any, is determined for which $1 \leqq i \leqq n$ and for which the position identifier "ends" in β. For example, in the string $dabcab\$$ with $x = dab$, $\beta = c$, $r = ab$, the position identifier for position 2 is abc, i.e., it "ends" in β. In the string $dabc\$$ with $x = dab$, $\beta = c$, $r = \varepsilon$, there is no position i, $1 \leqq i \leqq n$, for which the position identifier "ends" in β. We let i_0 be the leftmost position in x for which the position identifier "ends" in β, if any, else i_0 is set to $n+1$. i_0 will be used in step III. In step Ia all leaves carrying a position number j, $n+1 \leqq j \leqq n+m$, are removed. These position numbers belong to β. After the removal of these leaves it may be that a node N lost all sons. This situation occurs if two or more positions in β have a common prefix that does not occur outside β. The path from such a node N to its youngest ancestor that has more than one son has to be removed. This is performed in Ib.

As a result of Ia and Ib it may be the case that a father is left with one son and this son is a leaf with position number l. This situation occurs if the position l and some positions in β had a common prefix but the prefix did not occur at another position outside β. Then the path to the leaf l has to be shortened. This is done in II. In step III we treat all positions i, $i_0 \leqq i \leqq n$. The path to the node with position number i is shortened by pushing the position number i stepwise upwards. Whenever the node, from which the position number i is being passed to its father, is a leaf this leaf is removed. We stop moving the position number i upwards if the current node has height $n-i+1$. This corresponds to cutting off that suffix of the identifier for i that is a prefix of β. At the same time, however, we take care of all positions k, with $k < i_0$ or $k > n+m$. This is achieved by marking a node if it has exactly one son left and this is a leaf with position number l. If $i_0 \leqq l \leqq n$, the position l will be processed anyway by III. Otherwise, in step IV the path leading to the leaf l is shortened until the point is reached where a prefix common to l and another position is encountered. In step V the position numbers for γ are attached to the root. In step VI the position identifiers for γ are built up; simultaneously, a path to a node with position number l, $1 \leqq l \leqq n$ or $l > n+m$, is prolonged, if necessary; i.e., the position number is pushed downwards.

Correction algorithm.

```
(0)   $i := n$; while $(i + h_i > n+1) \wedge i \geqq 1$ do ⌜$i := i-1$⌟; $i_0 := i+1$
(Ia)  For $i = 1, \cdots m$ do
      ⌜remove the leaf with position number $n+i$; mark the father if there is
        exactly one son left and this is a leaf⌟
(Ib)  Choose a marked node without sons;
      remove node;
      if father has no more sons left, mark father;
      if father has only one son left and this is a leaf, mark father;
      if there are marked nodes without sons goto Ib;
(II)  Choose a marked node $R$; co $R$ has exactly one son and this is a leaf co
      remove mark;
      give position number of the son of $R$ to $R$;
      remove son;
      if $R$ is the only son, mark its father;
      if there are marked nodes left goto II.
(III) For $i = n, n-1, \cdots, i_0$ do
      ⌜while $i + h_i > n+1$ do
        ⌜give position number $i$ to father of $N = N_i$ and remove it at $N$;
          co now father $= N_i$ co
          if $N$ is a leaf then
          ⌜remove $N$;
```

```
            if $N_i$ has exactly one son left and this is a leaf mark $N_i$;
            if $N_i$ is a leaf and the only son mark father of $N_i$;┘┘
        if $N_i$ is a leaf remove mark—if any—else underline $N_i$.┘
(IV) Choose a marked and not underlined node $R$;
     remove mark;
     give $R$ the position number of its son;
     remove son;
     if $R$ is the only son, mark its father;
     if there is a marked and not underlined node left goto IV.
(V)  If $l \leqq m$ give position number $n+1, \cdots, n+1$ to the root else give position
     number $n+1, \cdots, n+m-1, (n+m).1, (n+m).2, \cdots, (n+m).l-m+1$
     to the root; underline root co Dewey-Decimal numbers used co
(VI) For each underlined node $N$ do
     ┌For each position number $i$ at $N$ do
       ┌remove position number $i$ from $N$;
        follow path beginning at node $N$ as long as the word of labels from the
        root to the current node coincides with a substring beginning at the
        position numbered $i$ in $x\gamma r\$$.
        Let $N_{last}$ be the last visited node.
        If $N_{last}$ is not a leaf then
        ┌attach a new son to $N_{last}$; give the new son the position number $i$; label
           the edge by the $h(N_{last})$th letter following the position $i$ in $x\gamma r\$$┘
        If $N_{last}$ is a leaf with position number $j$ then
        ┌remove position number $j$ from $N_{last}$; let $a$ be the $h(N_{last})$th letter
           following position $j$ in $x\gamma r\$$; let $a'$ be the $h(N_{last})$th letter following
           position $i$ in $x\gamma r\$$; while $a = a'$ do
           ┌attach a new son $NS$ to $N_{last}$; label the edge by $a(=a')$ and set
           $N_{last} := NS$┘
           attach two new sons $NS_1$ and $NS_2$ to $N_{last}$; label the edge to $NS_1$ by $a$
           and give $NS_1$ the position number $j$; label the edge to $NS_2$ by $a'$ and give
           $NS_2$ the position number $i$┘┘
      remove underlining and marks, if any, at $N$┘
```

Let us illustrate the algorithm by the following example.

Example. Consider the string $abcabc\$$ with position numbers $1, 2, \cdots, 7$. The position tree is shown in Fig. 12.

Let $x = ab$, $\beta = c$, $r = abc$, $\gamma = \varepsilon$. Step 0 yields $i_0 = 1$. Let us now perform step I of the algorithm. Hence we remove the leaf with position number 3 and mark its father as there is only one son left. There are no marked nodes without sons, hence we can skip Ib and get Fig. 13 as result of step I.

Performing step II yields Fig. 14. Starting step III with $n = 2$ ($n = |x|$) and step IV yield the tree Fig. 15. Step V is not performed as $l = 0$. Step VI yields Fig. 16, for the string $x\gamma r\$ = \underset{1}{a}\,\underset{2}{b}\,\underset{4}{a}\,\underset{5}{b}\,\underset{6}{c}\,\$$ with annotated position numbers which are interpreted as Dewey-Decimal numbers.

In order to prove the correction algorithm correct we need some auxiliary definitions.

A *reduced* identifier for position i in $x\beta r\$$, $i \notin \{n+1, \cdots, n+m\}$ is a prefix p_i of the position identifier such that p_i does not occur at another position j, $j \neq i$, $j \notin \{n+1, \cdots, n+m\}$ in $x\beta r\$$.

Clearly, the position identifier for position i, $i \notin \{n+1, \cdots, n+m\}$ is a reduced identifier.

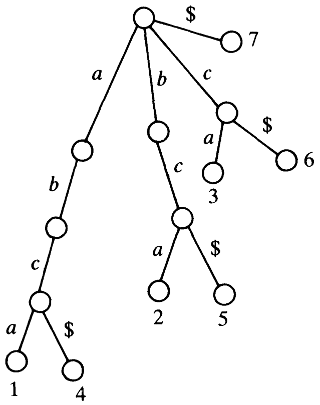


FIG. 12

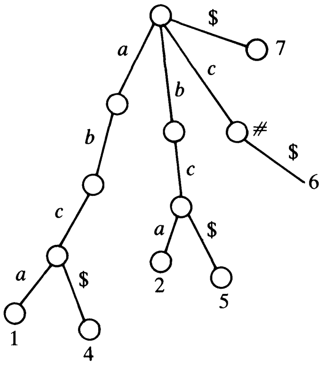


FIG. 13

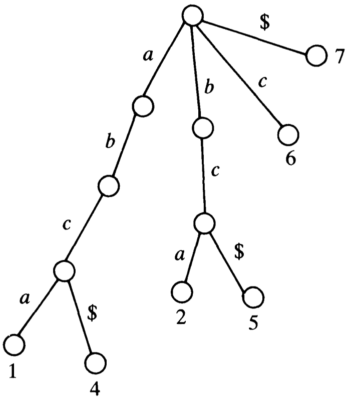


FIG. 14

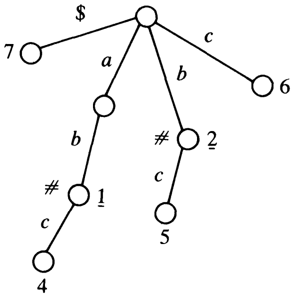


FIG. 15

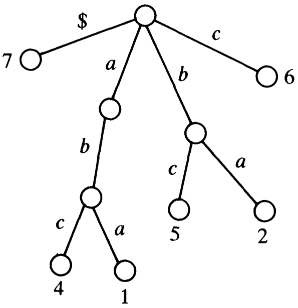


FIG. 16

A *partial* identifier for i, $1 \leqq i \leqq n$, in $x\beta r\$$ is a word p_i such that

(I) p_i is a prefix of $x_i \cdots x_n$;

(II) p_i is a prefix of the position identifier for i in $x\beta r\$$;

(III) p_i does not occur at j, $j \neq i$, $j \notin \{n+1, \cdots n+m\}$ in $x\beta r\$$.

If there is such a p_i, then the shortest word fulfilling I, II, III is called the shortest partial identifier, else $x_i \cdots x_n$ is called the shortest partial identifier.

A *partial* identifier for j, $n+m+1 \leqq j \leqq n+m+|r|$, in $x\beta r\$$, is a prefix p_j of the position identifier such that

(I) p_j does not occur at position k, $k \geqq n+m+1$, $k \neq j$;

(II) if p_j occurs at $k \leqq n$ then

$$|p_j| > |\text{shortest partial identifier for } k|.$$

The above declared notions correspond to the different steps of the algorithm, which is shown in the following.

LEMMA 3. *After step* Ia *of the algorithm the following holds:*

(i) *There may be leaves without position number.*

(ii) *A leaf is without position number if and only if it is marked.*

(iii) *If a node is marked then either it is a leaf and has no position number or it has exactly one son. This son is a leaf. If a node N has a single son and this is a non-marked leaf, then N is marked.*

Proof.

(i) If a node has only sons which are leaves with a position number $i \in \{n+1, \cdots, n+m\}$, then by step Ia this node becomes a leaf without position number.

(ii) If a leaf L has no position number, then it was the father of leaves with a position number i, $n+1 \leqq i \leqq n+m$. The node L had at least 2 sons before step Ia, as the input of the algorithm is a position tree. After the removal of the next to last son of L, L has been marked by the algorithm. If a leaf is marked after Ia then it must have been an inner node before Ia and did not have any position number.

(iii) Let the node N be marked as a result of Ia; then it was at some step the father of a leaf L which was the only son. In the sequel either this leaf is removed and N becomes a leaf without position number and is marked or the leaf remains there. If N has a single son which is a leaf and unmarked then N must have had another son, as the input of the algorithm is a position tree. After the removal of this son N has been marked.

LEMMA 4. *After step* I *of the algorithm the following holds:*

(i) *Every leaf has a position number.*

(ii) *A node is still marked iff it has exactly one son left which is a leaf.*

(iii) *Each path from the root to a leaf corresponds to a position identifier. For each position* $i \notin \{n+1, \cdots, n+m\}$ *the position identifier is still in the tree.*

Proof.

(i) This is obvious as every leaf without position number has been marked (Lemma 3). The marked leaves are removed by Ib.

(ii) Let a node be marked as a result of I; then it must have a son, otherwise it would have been removed by Ib. Moreover only nodes are marked which have a single son that is a leaf. Let a node be given that has a single son that is a leaf. Then by (i) this leaf has a position number. Hence N must have had at least another son before I has been performed. At the removal of the next to last son, N has been marked.

(iii) Obvious, as only leaves with position number $i \in \{n+1, \cdots, n+m\}$ are affected by step I.

Remark. In step I of the algorithm all and only those nodes $\neq$ root are removed which (1) lie on a path from the root to a leaf with position number i, $i \in \{n+1, \cdots, n+m\}$, and (2) do not lie on a path from the root to a leaf with position number $i \notin \{n+1, \cdots, n+m\}$.

LEMMA 5. *After step* II *of the algorithm*

(i) *Each leaf has a position number.*

(ii) *There are no marks left.*

(iii) *Each path from the root to a leaf corresponds to the shortest reduced identifier.*

Proof.

(i) Obvious by Lemma 4.

(ii) Obvious.

(iii) By Lemma 4, the position identifier is still contained in the tree before II. We may distinguish two cases. Either the leaf is a single son in which case its father is marked and will be processed by step II. The path is shortened as long as the respective father becomes a single son. After this the path reflects the shortest reduced identifier, as any shorter prefix would occur at least at one other position. In the other case the position identifier cannot be reduced any further.

LEMMA 6.

(i) *If after step* III *a node is underlined then it has at least one son.*

(ii) *After step* III

ii1) *Each path from the root to a non-leaf node with position number corresponds to a shortest partial identifier*;

ii2) *Each path from the root to a leaf corresponds to a partial identifier.*

Proof.

(i) Obvious, since nodes which are leaves are not underlined.

(ii1) Let p_i be the word of labels from the root to the inner node with position number i. Then $i \leqq n$ and p_i occurs at least at another position; hence p_i is not a partial identifier. But $p_i = x_i \cdots x_n$ as step II is performed until $i + h_i = n + 1$; hence p_i is a shortest partial identifier by definition.

ii2) Let the word of labels from the root to a leaf n_i with position number i be to p_i. If $i \leqq n$, then $h_i \leqq n + 1 - i$ and p_i is a prefix of the position identifier for position i in $x\beta r\$$ and p_i is a prefix of $x_i \cdots x_n$. Moreover, n_i is a leaf and hence p_i does not occur at another position j, $j \notin \{n+1, \cdots, n+m\}$. Hence, p_i is a partial identifier. If $i > n + m$, then the path from the root to n_i yields the shortest reduced identifier which is a partial identifier by definition.

Remark. After step IV for each $i \notin \{n+1, \cdots, n+m\}$ the path from the root to the node with position number i corresponds to the shortest partial identifier. This is clear, as an identifier is shortened in this step until a node is reached which has more than one son.

Finally, in step V the new position numbers are attached at the root and in step VI the position identifiers are built up.

In order to be able to perform the above algorithm we need the following information for each node:

(1) a list of sons,

(2) the position identifiers associated with the node,

(3) the height of the node,

(4) the father of the node.

In addition we have to maintain a list of underlined nodes and a list of marked nodes. Moreover, in order to guarantee random access to the leaves of the tree for step Ia, we store—in the array containing the text—for each position number a reference to the leaf with position number i.

Based on this representation the cost of step (0) is $O(n - i_0 + 2)$, where

$$i_0 = \begin{cases} \min\{i\colon 1 \leqq i \leqq n \text{ and pos. identifier for } i \text{ is longer than } n - i + 1\}, & \text{if this set is not empty;} \\ n + 1 \text{ else.} \end{cases}$$

Step Ia costs $O(m)$ steps, where m is the length of β.

Steps Ib and II can only be performed at nodes that lie on the path from a leaf with position number $n+i$, $1 \leqq i \leqq m$, to the root (in the tree for $x\beta r\$$). The number of nodes on the path from the leaf with position number j to the root (excluding the root) is given by $|p_j(x\beta r)|$, where $p_j(x\beta r)$ is the position identifier for position j in $x\beta r\$$. Hence the total number of nodes which will be affected by steps Ib, II can be bounded above by

$$\sum_{i=1}^{m} |p_{n+i}(x\beta r)|.$$

Each node is processed at most once and the operations for each node cost constant time; hence steps Ib, II cost

$$O\left(\sum_{i=1}^{m} |p_{n+i}(x\beta r)|\right).$$

Let T^{II} be the tree resulting from step II. Step III can affect only such nodes in the tree T^{II} that lie on a path from a leaf that has a position number i, $i_0 \leqq i \leqq n$, and fulfills $h_i + i > n+1$ to the ancestor A_i of that leaf for which

$$h(A_i) + i = n+1$$

holds. The path in T^{II} from the root to the leaf i contains (without root) at most $|p_i(x\beta r)|$ nodes; hence the path from the ancestor A_i to the leaf i contains at most

$$|p_i(x\beta r)| - h(A_i) = |p_i(x\beta r)| - (n+1-i)$$

nodes. As the loop in step III manipulates the nodes with position numbers i_0, $i_0+1, \cdots, n$, the total number of nodes affected by step III can be bounded above by

$$\sum_{i=i_0}^{n} (|p_i(x\beta r)| - (n+1-i)).$$

A lesser upper bound for the number of nodes affected by step III can be given in terms of the shortest reduced identifier. Let shr_i be the shortest reduced identifier for position i in $x\beta r\$$. As a result of step III every path from the root to a leaf i corresponds to the shr_i. Hence, the total number of nodes affected by step III is

$$\sum_{i=i_0}^{n} \max(0, |shr_i| - (n+1-i)).$$

Here, we have to use $\max(0, |shr_i| - (n+1-i))$ because $|shr_i|$ may be less than $n+1-i$. If this is the case for leaf i then step III will not be performed for the position number i.

The cost for manipulating a node is constant, hence the total cost of III can be given by

$$O\left(\sum_{i=i_0}^{n} \max(0, |shr_i| - (n+1-i))\right)$$
$$\leqq O\left(\sum_{i=i_0}^{n} (|p_i(x\beta r)| - (n+1-i))\right).$$

As a result of III for each position number i, $i_0 \leqq i \leqq n$, there is at most one marked node left the marking of which was caused by i; if in T^{III} the node N_i carrying the position number i is a leaf (T^{III} is the result of III), then the father of N_i may be marked. If N_i is not a leaf and hence underlined, then it may be marked. If N_i is not a leaf and hence underlined then it may be that there is a successor of N_i which was marked as the leaf with position number i was removed. If the position number i was not processed by III because $i+h_i \leqq n+1$, then i did not cause any marking. If N_i is a leaf as a result of III, at most the nodes on the path between the root and N_i can be affected by step IV. If N_i is not a leaf, then at most the nodes between N_i and the possibly existing marked successor can be affected by step IV. Hence we can bound the total number of nodes affected by step IV,

$$\sum_{i\in I_1} (n-i+1) + \sum_{i\in I_2} (|shp_i| - (n+1-i)),$$

where shp_i is the shortest partial identifier for i in $x\beta r\$$ and

$$I_1 = \{i\colon i_0 \leqq i \leqq n \text{ and } i+h_i > n+1 \text{ in } T^{\mathrm{II}} \text{ and } N_i \text{ is leaf in } T^{\mathrm{III}}\},$$

$$I_2 = \{i\colon i_0 \leqq i \leqq n \text{ and } i+h_i > n+1 \text{ in } T^{\mathrm{II}} \text{ and } N_i \text{ is not a leaf in } T^{\mathrm{III}}\},$$

where T^{II} is the tree resulting from step II and T^{III} is the result of step III.

Step (V) costs $O(l)$ time.

For step VI we note that an underlined node is either the root and carries l position numbers or it is an inner node and carries exactly one position number. The total cost of step VI can be bounded above by

$$O\Bigl(\sum_{d\in\{d_{n+1},\cdots,d_{n+l}\}} |p_d(x\gamma r)| + \sum_{i=i_0}^{n} (|p_i(x\gamma r)| - |shp_i|)\Bigr),$$

where shp_i is the shortest partial identifier for i in $x\beta r\$$, $p_i(x\gamma r)$ is the position identifier for position i in $x\gamma r\$$, and $d_{n+1}, \cdots, d_{n+l}$ are the (Dewey-Decimal) position numbers for the $(n+1), \cdots, (n+l)$th letters in $x\gamma r\$$. This can be seen by the fact that the inner loop of VI executes for $d \in \{d_{n+1}, \cdots, d_{n+1}\}$, as many steps as the length of $p_d(x\gamma r)$. In the case of $i \in \{i_0, \cdots n\}$, the shortest partial identifier for position i is prolonged to become $p_i(x\gamma r)$.

The above algorithm and its cost considerations work under the condition that the position numbers of the input string $x\beta r\$$ are the natural numbers $1, 2, \cdots, |x\beta r\$|$. However, as we may want to apply the correction algorithm repeatedly, we have to give some consideration to the general case where the input may be numbered by Dewey-Decimal numbers. There is one obvious approach: after an application of the correction algorithm all positions that are situated to the right of the correction position n are renumbered resulting in position numbers $1, 2, \cdots, |x\gamma r\$|$. This solution corresponds to shifting text in the text array.

In general if there is no restriction as to when corrections can be made, this solution will not be feasible. Hence, we have to find out which changes have to be made to our correction algorithm in order to allow for input with Dewey-Decimal position numbers. Let

$$d_1, d_2, \cdots, d_{|x\beta r\$|}$$

be the Dewey-Decimal position numbers for $x\beta r\$$. Let $|x| = n$, $|\beta| = m > 0$, as before. We call the text array with pointers to substrings (as explained in the beginning of § 6) an extended array. Let the position number of the first letter of β be d^{β}, i.e., $d^{\beta} = d_{n+1}$.

In general we will have to deal with the case that the input of the algorithm consists of the position number d^{β}, the strings β, γ, the position tree for $x\beta r\$$, and the extended text array for $x\beta r\$$. The size of n, i.e., the length of the text to the left of β, will not be given as input in general (even though it could be determined by traversing the extended text array).

We will use the following notations and operations. Let

$$d_1, \cdots, d_{|w|}$$

be the Dewey-Decimal position numbers for the text w. We define

$$\text{pred}(d_i) = d_{i-1}, \quad \text{for } i > 1,$$
$$\text{succ}(d_i) = d_{i+1}, \quad \text{for } i < |w|;$$

d_i is the position number of the ith letter in w, pred(d) determines the position number of the position left of d, succ(d) determines the position number of the position right of d.

Moreover we define an addition between Dewey-Decimal numbers and natural numbers by

$$d_i + h := d_{i+h}, \quad \text{if } i + h \leqq |w|.$$

The order between Dewey-Decimal numbers is the lexicographical order.

Let us now consider the position tree for $x\beta r\$$ with position numbers $d_1, \cdots, d_{|x\beta r\$|}$. As before, let N_d be the node carrying the position number d and let $h(d)$ be the height of N_d. We consider the modifications which have to be made to our original correction algorithm in order to allow for Dewey-Decimal position numbers for the input.

One can immediately see that steps Ib, II, IV, VI of the correction algorithm can remain unchanged if we want to use Dewey-Decimal numbers for the input. Steps 0, Ia, III and V have to be substituted by

```
(0′)  If $d^\beta > d_1$ then
      ⌜$d := \text{pred}(d^\beta)$;
        while $d + h(d) > d^\beta \wedge d > d_1$ do
        ⌜$d := \text{pred}(d)$⌟;
        If $d = d_1 \wedge d + h(d) > d^\beta$ then $d^0 := d$
        else $d^0 := \text{succ}(d)$⌟
(Ia′) $d := d^\beta$;
      For $i = 1, \cdots, m$ do
      ⌜remove leaf with position number $d$; mark father if there is exactly one son
        left and this is a leaf; $d := \text{succ}(d)$⌟
(III′) If $d^\beta > d_1$ then
      ⌜$d := \text{pred}(d^\beta)$;
        while $d \geqq d^0$ do
        ⌜while $d + h(d) > d^\beta$ do
          ⌜give position number $d$ to father of $N = N_d$ and remove it at $N$;
            co now father $= N_d$co
            if $N$ is a leaf then
            ⌜remove $N$;
              if $N_d$ has exactly one son left and this is a leaf then mark $N_d$;
              if $N_d$ is a leaf and the only son then mark father of $N_d$⌟⌟;
              if $N_d$ is a leaf then remove mark—if any—else underline $N_d$;
              if $d = d_1$ then exist else $d := \text{pred}(d)$⌟⌟
(V′)  If $l \leqq m$ give the position numbers $d_{n+1}, \cdots, d_{n+l}$ to the root else give the
      position numbers $d_{n+1}, \cdots, d_{n+m-1}, d_{n+m} . 1, d_{n+m} . 2, \cdots, d_{n+m} . (l - m +$
      $1)$ to the root.
```

As one can see, the correction algorithm using Dewey-Decimal numbers for the input can be obtained by minor modifications of the original algorithm. It should be noted, however, that this new algorithm is more expensive. This can be seen from the following observations. (i) Given a position number d, there is no immediate access to the next smaller or next greater position number. We have to find these position numbers in the extended array. This costs at most as many steps as there are levels of references to substrings in the extended array. (ii) In order to calculate $d + h(d)$ for steps (0′) and (III′) we have to find the $h(d)$th position number following d. (III) To determine the $h(N_{\text{last}})$th letter following position d in step VI we have to traverse parts of the extended array.

The costs for (ii) and (iii) can be reduced if we maintain information about the length of the substrings that are pointed at from the main array. Work is in progress that investigates some time-space tradeoffs of the above sketched solution and investigates garbage collection problems caused by replacing β by γ with $|\gamma| < |\beta|$.

Let us finally consider the correction problem in compacted trees. For compacted trees there is a simpler solution to the correction task which will be explained in the following. We assume for simplicity that the position numbers of the input $x\beta r\$$ are the natural numbers $1, 2, \cdots, |x\beta r\$|$. The algorithm takes as input the compacted position tree for $x\beta r\$$. If a node N in this tree has a single son, then N is marked.

For each position number i, let dh_i be the length of the word of labels on the path from the root to the node carrying the position number i. As before, $|x| = n$, $|\beta| = m > 0$, and $|\gamma| = l$. In the algorithm we make use of an auxiliary symbol $\perp \notin \Sigma \cup \{\$\}$. The position numbers for $x\gamma r\$$ are $d_1, \cdots, d_{|x\gamma r\$|}$. We write $x\gamma r\$ = a_1 a_2 \cdots a_n a_{n+1} \cdots a_{n+l} \cdots a_{|x\gamma r|}\$$.

Correction in compacted position trees.

```
(0)   $i := n$;
      while $i + dh_i > n + 1 \wedge i \geqq 1$ do $\ulcorner i := i - 1 \lrcorner$;
      $i_0 := i + 1$
(I)   $i := i_0$;
      while $i \leqq n + m$ do
      $\ulcorner$remove the leaf with position number $i$;
        if the father $F$ has exactly one son $L$ left and this is not a leaf then
        $\ulcorner$mark $F$;
        if there are two adjacent marked nodes, then compactify
        if the father $F$ has exactly one son $L$ left and this is a leaf then
        $\ulcorner$give $F$ the position number $j$ of $L$;
          remove $L$;
          if $F$ is the single son of $E$ then
          $\ulcorner$give $E$ the position number $j$;
            remove $F$; remove mark at $E \lrcorner$
          else replace the word of labels $c_1 \cdots c_n$ on the edge from $E$ to $F$ by
          $c_1 \lrcorner$ ;
      $i := i + 1 \lrcorner$
(II)  attach a $\perp$-son to the root with position number $d_{i_0}$; remove mark at root, if
      any
(IIIa) For $i = i_0, i_0 + 1, \cdots, n + l$ do
      $\ulcorner$let $a$ be the $i$th letter in $x\gamma r\$$;
        for each node $N$ that has a $\perp$-son $N'$ do;
        (The order in which the nodes are processed is that given in Lemma 1)
        $\ulcorner$if $N$ does not have an $ar$-son, $r \in \Sigma^*$, then change the label of the edge
          $(N, N')$ from $\perp$ to $a$;
        if $N$ has an $ar$-son $N''$, $r \in \Sigma^*$, then
        $\ulcorner$remove the edge $(N, N')$;
```

```
              mark $N$, if there is exactly one son left;
              if $r \neq \varepsilon$ then
              ⌜create a new $NN$;
                make the sons of $N''$ the sons of $NN$;
                make $NN$ the $r$-son of $N''$;
                if $N''$ was marked transfer the mark to $NN$;
                change the label of $(N, N'')$ from $ar$ to $a$⌟;
              if $r = \varepsilon$ and $N''$ is not a leaf then remove mark at $N''$, if any;
              if $r = \varepsilon$ and $N''$ is a leaf then
              ⌜attach a new son to $N''$; transfer the position number $j$ of $N''$ to the
                new son; label the edge between $N''$ and the new son by the $l$th letter
                following position $j$ in $a_1, \cdots, a_i \perp r\$$ (where $l$ is the length of the
                position identifier for position $j$ in $a_1 \cdots a_{i-1} \perp r\$$)⌟;
             attach $N'$ as a $\perp$-son to $N''$ together with the position number⌟⌟;
          attach a $\perp$-son at the root; give it position number $d_{i+1}$; remove mark at
          root, if any;
          for all pairs of adjacent marked nodes: compactify⌟
    (IIIb) remove $\perp$-son of root
     (IV) $i := n + l + 1$:
          while there are nodes with a $\perp$-son do
          ⌜let $a$ be the $i$th letter in $x\gamma r\$$;
            for each node that has a $\perp$-son do: (order given in Lemma 1)
            ⌜ as in step IIIa⌟;
            for all pairs of adjacent marked nodes;
            compactify;
            $i := i + 1$⌟
```

The algorithm basically words as follows. In step 0 the leftmost position i_0 is determined by the position identifier of which "ends" in β. In step I the position identifiers for all positions i, $i_0 \leqq i \leqq n + m$ are removed from the tree. The tree is now a compacted position tree for $x_1 \cdots x_{i_0 - 1} r\$$. In step II we attach an auxiliary $\perp$-son with position number d_{i_0} to the root; this corresponds to work with the string $x_1 \cdots x_{i_0-1} \perp r\$$. In step III we use the on-line construction algorithm for compacted position trees for $a_1 \cdots a_{i_0} \perp r\$, \cdots, a_1 \cdots a_{n+l} \perp r\$$. In step IV we continue to use the on-line algorithm until the $\perp$ does not occur any more as part of any position identifier. In contrast to step III we do not introduce any new position numbers in step IV, as the position numbers for the positions right of γ are already in the tree.

The costs of the algorithm are as follows: let $x^* = x_{i_0} \cdots x_n$ and let $r^* = r_1 \cdots r_{j_0}$, where $\gamma_l r_1 \cdots r_{j_0}$ is the longest prefix of $\gamma_l r\$$ that occurs twice in $x\gamma r\$$. Step 0 costs $O(|x^*|)$. Step I costs $O(|x^*| + |\beta|)$, because the number of iterations is $|x^*| + |\beta|$ and each iteration costs constant time. Step IIIa basically applies the on-line construction algorithm for compacted trees $(|x^*| + |\beta|)$-times, step IV applies the on-line construction algorithm[5] $|r^*|$-times.

Acknowledgment. The authors would like to thank Prof. R. Bayer for pointing out the problem to them and for many useful discussions.

[5] One application of the on-line construction algorithm costs $O(\sum_{i \in I} |S_i| + 1)$, where the list of nodes with a $\perp$-son is $\{N_i\}_{i \in I}$ and S_i is the set of sons of N_i.

Note added in proof. In the above we use the notions "height of a node" and "depth of a node" synonymously.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMANN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1975.

[2] R. COHEN AND M. CIMET, *A scheme for constructing on-line linear time recognition algorithms*, Conference on Theoretical Computer Science, Waterloo, Ontario, Canada, 1977, pp. 70–80.

[3] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1976), pp. 262–272.

[4] E. M. MCCREIGHT, *A space-economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23 (1976), pp. 262–272.

[5] P. WEINER, *Linear pattern matching algorithms*, IEEE 14th Annual Symposium on Switching and Automata Theory 1973, pp. 1–11.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0013 $01.00/0

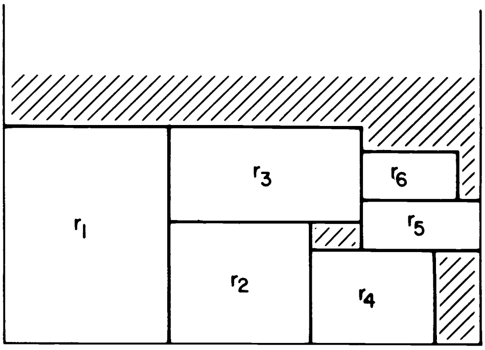
PERFORMANCE BOUNDS FOR LEVEL-ORIENTED TWO-DIMENSIONAL PACKING ALGORITHMS*

E. G. COFFMAN, JR.†, M. R. GAREY‡, D. S. JOHNSON‡, AND R. E. TARJAN§

Abstract. We analyze several "level-oriented" algorithms for packing rectangles into a unit-width, infinite-height bin so as to minimize the total height of the packing. For the three algorithms we discuss, we show that the ratio of the height obtained by the algorithm to the optimal height is asymptotically bounded, respectively, by 2, 1.7, and 1.5. The latter two improve substantially over the performance bounds for previously proposed algorithms. In addition, we give more refined bounds for special cases in which the widths of the given rectangles are restricted and in which only squares are to be packed.

Key words. level-oriented packing algorithm, bin-packing, two-dimensional packing

1. Introduction. We consider the following two-dimensional packing problem, first proposed in [1]: Given a collection of rectangles, and a bin with fixed width and unbounded height, pack the rectangles into the bin so that no two rectangles overlap and so that the height to which the bin is filled is as small as possible. We shall assume that the given rectangles are oriented, each having a specified side that must be parallel to the bottom of the bin. We also assume, with no loss of generality, that the bin width has been normalized to 1. Fig. 1 provides an illustration, where the first dimension specified for each rectangle corresponds to the side that must be parallel to the bottom of the bin (we use this convention throughout the paper).



Dimensions for r_i, $1 \leqq i \leqq 6$: $\frac{7}{20}\times\frac{9}{20}$, $\frac{3}{10}\times\frac{1}{4}$, $\frac{2}{5}\times\frac{1}{5}$, $\frac{1}{4}\times\frac{1}{5}$, $\frac{1}{4}\times\frac{1}{10}$, $\frac{1}{5}\times\frac{1}{10}$

FIG. 1. *A packing of oriented rectangles in a unit-width bin.*

This problem is a natural generalization of the one-dimensional bin-packing problem studied in [9]. Indeed, if all rectangles are required to have the same height, then the two problems coincide. On the other hand, the case in which all rectangles have the same *width* corresponds to the well-known makespan minimization problem of combinatorial scheduling theory [3]. Both these restricted problems are known to be NP-complete [3], [7], from which it follows trivially that the two-dimensional packing

* Received by the editors February 9, 1979, and in final revised form February 4, 1980.

† Bell Laboratories, Murray Hill, New Jersey 07974. The work of this author was supported in part by the National Science Foundation and Bell Laboratories.

‡ Bell Laboratories, Murray Hill, New Jersey 07974.

§ Stanford University, Stanford, California. The work of this author was supported in part by the National Science Foundation under Grant MCS 75-22870-A02, the U.S. Department of Naval Research under Contract N 00014-76-C-0688, a Guggenheim Fellowship, and Bell Laboratories.

problem is also NP-complete. For this reason we shall focus on fast heuristic algorithms for solving this problem, seeking to prove close bounds on the extent to which they can deviate from optimality. Those readers unfamiliar with this "approximation algorithms" approach may wish to consult one or more of [4], [6], [7], for general background and for examples of other problems to which it has been applied.

For L an arbitrary list of rectangles, all assumed to have width no more than 1, let $\mathrm{OPT}(L)$ denote the minimum possible bin height within which the rectangles in L can be packed, and let $A(L)$ denote the height actually used by a particular algorithm when applied to L. The results in [1] are concerned primarily with demonstrating *absolute performance bounds* for various algorithms A, i.e., bounds of the form

$$A(L) \leqq \beta \cdot \mathrm{OPT}(L)$$

for all lists L. In contrast, we will be interested in proving *asymptotic performance bounds* of the form

$$A(L) \leqq \beta \cdot \mathrm{OPT}(L) + \gamma$$

for all lists L. This is of interest because in many cases the worst absolute performance can be achieved only by highly specialized, "small" examples. The constant β in such an asymptotic bound is intended to characterize the behavior of the algorithm as the ratio between $\mathrm{OPT}(L)$ and the maximum height rectangle in L goes to infinity. For this purpose, we may, without loss of generality, normalize the height of the tallest rectangle in L to 1. (Of course, any height other than 1 would serve just as well for this normalization; a different choice would affect only the additive constant γ in our bounds, leaving the multiplicative constant β unchanged.)

The difference between these two types of bounds is illustrated by the Next-Fit Decreasing Height (NFDH) algorithm, to be defined in § 2, where it is known [2], [8] that

$$\mathrm{NFDH}(L) \leqq 3 \cdot \mathrm{OPT}(L)$$

for all lists L, and that there exist lists L for which $\mathrm{NFDH}(L)$ is arbitrarily close to $3 \cdot \mathrm{OPT}(L)$. We shall show, however, that if the height as well as the width of each rectangle is no more than 1, then

$$\mathrm{NFDH}(L) \leqq 2 \cdot \mathrm{OPT}(L) + 1,$$

for all L, and the multiplicative constant 2 cannot be improved upon. In the case of NFDH, as with the other algorithms we shall consider, asymptotic performance bounds seem to provide more accurate and useful information, properly relegating "transient" effects to the additive constant. (However, as we note in the conclusion, many of our asymptotic results also provide good absolute bounds.)

In addition to proving asymptotic bounds that hold for all lists L, we will also be interested in proving such bounds for special cases in which the rectangles in L satisfy additional width restrictions or all are required to be squares. Section 2 examines several approximation algorithms that are natural analogues of the one-dimensional packing algorithms considered in [9], and provides best possible performance bounds for them. Perhaps surprisingly, these bounds turn out to be essentially the same as those for the one-dimensional case. In § 3 we propose a new algorithm for the two-dimensional packing problem, and prove tight bounds on its performance. These bounds demonstrate that the new algorithm is a substantial improvement over the algorithms in § 2.

Finally, in § 4, we compare the performance bounds for our algorithms with those for the non-level-oriented algorithms of [1] and discuss several variants of our algorithms that might perform somewhat better in practice.

2. The NFDH and FFDH algorithms. The packing algorithms that we analyze in this section both assume that the rectangles in the list L are ordered by decreasing (actually, nonincreasing) height, and they pack the rectangles in the order given by L so as to form a sequence of *levels*. All rectangles will be placed with their bottoms resting on one of these levels. The first level is simply the bottom of the bin. Each subsequent level is defined by a horizontal line drawn through the top of the first (and hence maximum height) rectangle placed on the previous level. Notice how this corresponds with one-dimensional bin-packing; the horizontal slice determined by two adjacent levels can be regarded as a bin (lying on its side) whose width is determined by the maximum height rectangle placed in that bin. The following two level algorithms are suggested by analogous algorithms studied for one-dimensional bin-packing:

(1) *Next-Fit Decreasing-Height* (NFDH). With this algorithm, rectangles are packed left-justified on a level until there is insufficient space at the right to accommodate the next rectangle. At that point, the next level is defined, packing on the current level is discontinued, and packing proceeds on the new level.

(2) *First-Fit Decreasing-Height* (FFDH). At any point in the packing sequence, the next rectangle to be packed is placed left-justified on the first (i.e., lowest) level on which it will fit. If none of the current levels will accommodate this rectangle, a new level is started as in the NFDH algorithm.

Fig. 2 shows the results of applying the two packing rules to the same list. The essential difference between them is that whereas FFDH can always return to a previous level for packing a new rectangle, NFDH always places subsequent rectangles at or above the currrent level.

Some notation will be useful for conducting our analysis of these two algorithms. Let the list L be given as $r_1, r_2, \cdots, r_n$, and let $w(r)$ and $h(r)$ denote the width and height of rectangle r. By our previous assumptions we have $0 \leqq w(r) \leqq 1$ and $0 \leqq h(r) \leqq 1$, and we have $h(r_1) \geqq h(r_2) \geqq \cdots \geqq h(r_n)$. The space between two consecutive levels will be called a *block*. Packings will be regarded as a sequence of blocks $B_1, B_2, \cdots, B_k$, where the index increases from the bottom to the top of the packing. Let A_i denote the total area of the rectangles in block B_i, and let H_i denote the height of block B_i. Note that, by the manner in which these algorithms define levels, we have $H_1 \geqq H_2 \geqq \cdots \geqq H_k$.

Consider a particular rectangle r packed in block B_i. If block B_{i+1} was nonempty at the time r was packed, then we say that r is a *fallback item*. If, on the other hand, block B_i was the highest nonempty block at the time r was packed, then r is called a *regular item*. Note that all rectangles in an NFDH packing are regular items, and in any block in an FFDH packing all regular items are taller than and appear to the left of any fallback items in that block.

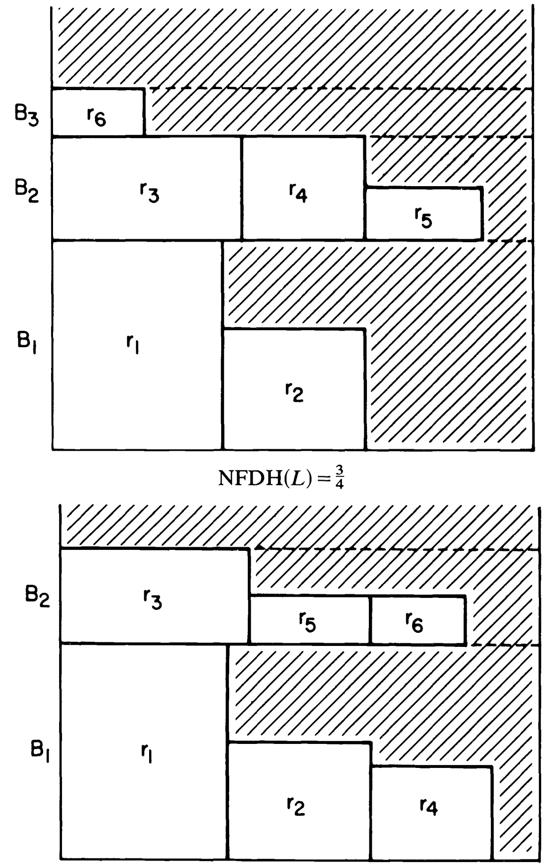
Our first result provides a tight bound on the performance of the NFDH algorithm.

THEOREM 1. *For any list L ordered by nonincreasing height,*

$$\text{NFDH}(L) \leqq 2 \cdot \text{OPT}(L) + 1.$$

Moreover, the multiplicative constant 2 *is the smallest possible.*

Proof. Consider the NFDH packing of such a list L, with blocks $B_1, B_2, \cdots, B_t$. For each i, let x_i be the width of the first rectangle in B_i, and y_i be the total width of rectangles in B_i. For each $i < t$, the first rectangle in B_{i+1} does not fit in B_i. Therefore $y_i + x_{i+1} > 1$, $1 \leq i < t$. Since each rectangle in B_i has height at least H_{i+1}, and the first



NFDH$(L)=\frac{3}{4}$

FFDH$(L)=\frac{13}{20}$

FIG. 2. NFDH *and* FFDH *algorithms applied to the list* $L=(r_1, r_2, r_3, r_4, r_5, r_6)$, *with the dimensions of* r_i *as given in Fig.* 1.

rectangle in B_{i+1} has height H_{i+1}, $A_i+A_{i+1}\geqq H_{i+1}(y_i+x_{i+1})>H_{i+1}$. Therefore, if A denotes the total area of all the rectangles,

$$\begin{aligned}\text{NFDH}(L)=\sum_{i=1}^{t}H_i&\leqq H_1+\sum_{i=1}^{t-1}A_i+\sum_{i=2}^{t}A_i\\&\leqq H_1+2A\\&\leqq 1+2\,\text{OPT}(L),\end{aligned}$$

which is the desired bound.

Examples showing that the coefficient of 2 is smallest possible are derived trivially from the corresponding examples for the Next-Fit algorithm of one-dimensional bin-packing. The list L has n rectangles, where n is a multiple of 4. All the rectangles have height 1, the odd numbered ones have width $\frac{1}{2}$, and the even numbered ones have width ε, for a suitably small $\varepsilon>0$. The optimum and NFDH packings of L are shown in Fig. 3. In this case we have NFDH$(L)=n/2$ and OPT$(L)=n/4+1$, so the ratio NFDH$(L)/$OPT(L) can be made arbitrarily close to 2 by choosing n suitably large and ε suitably small. □

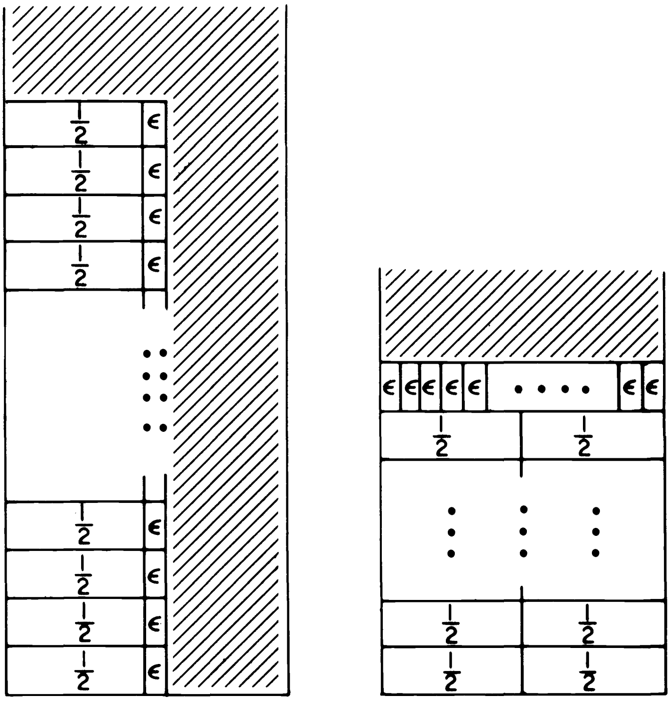


FIG. 3. *Worst-case examples for Theorem* 1.

It is perhaps surprising that the multiplicative constant of 2 in Theorem 1 is the same as the multiplicative constant for Next-Fit in the one-dimensional case. One might expect the two-dimensional algorithm to perform more poorly, since only the first item in a block need have height equal to the block's height, and there may be wasted space above the subsequent items that does not correspond to any waste in the one-dimensional case.

This wasted space, however, can only affect the additive constant in our result. Every item in B_i, $1 \leqq i \leqq t$, must have height at least H_{i+1} (set $H_{t+1} = 0$ by convention). Thus the total wasted space in B_i *above* items in B_i is at most $H_i - H_{i+1}$. The cumulative waste is consequently bounded by

$$\sum_{i=1}^{t} (H_i - H_{i+1}) = H_1 - H_{t+1} = H_1 = 1$$

(by our normalization assumption on heights). This same "collapsing sum" principle is at work in all results of this paper, and helps explain why in general the multiplicative constants do not change as we go from the one- to the two-dimensional case (although the proof techniques certainly do).

We turn now to the FFDH algorithm. It is routine to prove that FFDH(L)$\leqq$ NFDH(L) for all lists L. In the following we show that the worst-case bounds are significantly lower for the FFDH algorithm. We first prove the bound for the general case, and then we consider the special cases in which all rectangles have width no more than some fixed $\alpha < 1$ and in which all the rectangles are squares. The multiplicative constants we obtain in all three situations are best possible, and equal (when relevant) the corresponding constants in the one-dimensional case [9].

THEOREM 2. *For any list L ordered by nonincreasing height,*

$$\mathrm{FFDH}(L) \leqq 1.7 \cdot \mathrm{OPT}(L) + 1.$$

Furthermore, the multiplicative constant 1.7 *is the smallest possible.*

Proof. The proof is based on the analogous proof for the First-Fit algorithm of one-dimensional bin-packing [5], [9]. We begin by defining the following weighting function.

$$W(x) = \begin{cases} \frac{6}{5}x & \text{if } 0 \leqq x \leqq \frac{1}{6}, \\ \frac{9}{5}x - \frac{1}{10} & \text{if } \frac{1}{6} < x \leqq \frac{1}{3}, \\ \frac{6}{5}x + \frac{1}{10} & \text{if } \frac{1}{3} < x \leqq \frac{1}{2}, \\ \frac{6}{5}x + \frac{4}{10} & \text{if } \frac{1}{2} < x \leqq 1. \end{cases}$$

We extend this function to rectangles r by writing $W(r) = W(w(r))$, and set

$$A = \sum_{r \in L} h(r) \cdot W(r).$$

It is proved in [5] that no collection of numbers x summing to 1 or less can have $W(x)$ summing to more than 1.7. We can apply this result to our case by cutting the optimal packing into horizontal slices, formed by drawing a line through the top and bottom of each rectangle. Summing over all the slices, we have $A \leqq 1.7\ \mathrm{OPT}(L)$.

Thus, all that remains is to show that $A \geqq \mathrm{FFDH}(L) - 1$. Let T_1 be the set of blocks in the FFDH packing whose first rectangle has width at most $\frac{1}{2}$, and T_2 the set of blocks whose first rectangle has width greater than $\frac{1}{2}$. For $i = 1, 2$, let $H(T_i)$ be the total height of blocks in T_i. Since the first item in block B_i has height H_i, and since $W(x) > 1$ for $x > \frac{1}{2}$, we have

$$\sum_{B \in T_2} \sum_{r \in B} h(r) \cdot W(r) \geqq H(T_2).$$

We will show that

$$\sum_{B \in T_1} \sum_{r \in B} h(r) \cdot W(r) \geqq H(T_1) - 1.$$

Consequently $A \geqq H(T_1) + H(T_2) - 1 = \mathrm{FFDH}(L) - 1$, as desired.

Let L_1 be the sublist of L consisting of the rectangles in blocks of T_1. Note that the FFDH packing of L_1 would yield a set of blocks identical to T_1. Let $B_1, B_2, \cdots, B_t$ denote these blocks, with index increasing from the bottom to the top of the packing, and let H_i be the height of B_i, $1 \leqq i \leqq t$, with $H_{t+1} = 0$ by convention. Classify items as regular items or fallback items according to their roles in the FFDH packing of L_1. Let f_i be the first regular item in B_i, and let R_i be the set of all regular items in B_i. For $1 < i \leqq t$ and $1 \leqq j < i$, define F_{ij} to be the set of fallback items packed in B_j after the last regular item was packed in B_{i-1} but before the first regular item was packed in B_i. For $1 < i \leqq t$ and $1 \leqq j < i$, define S_{ij} to be the set of all fallback items packed in B_j after the first regular item was packed in B_i but before the last regular item was packed in B_i. See Fig. 4. Note that the sets F_{ij} and S_{ij} are all disjoint, and that $F_{i,i-1} = \varnothing$, $1 < i \leqq t$. Moreover, we have

$$L_1 = \bigcup_{i=1}^{t} \left[R_i \cup \bigcup_{j<i} F_{ij} \cup \bigcup_{j<i} S_{ij} \right].$$

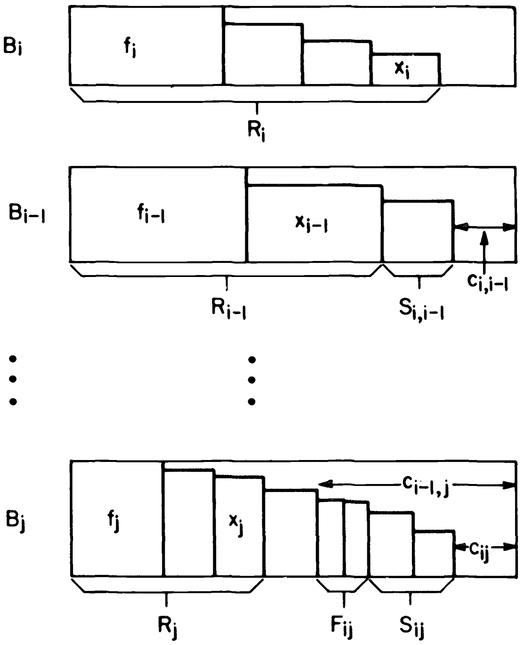


FIG. 4. *Illustration for proof of Theorem 2 (blocks have been separated for display purposes). Note that all items in F_{ij} come between x_{i-1} and f_i, while all items in S_{ij} and $S_{i,i-1}$ come between f_i and x_i in the list.*

For each i, $1<i\leqq t$, and each j, $1\leqq j<i$, define the *coarseness* c_{ij} to be the width of the empty space at the bottom right end of B_j when the last regular item is packed in B_i. Again see Fig. 4. Note that the following relations hold:

$$(2.1) \qquad w(f_i)\geqq c_{ij}+\sum_{r\in S_{ij}} w(r), \qquad 1<i\leqq t, \quad 1\leqq j<i,$$

$$(2.2) \qquad c_{ij}=c_{i-1,j}-\sum_{r\in S_{ij}\cup F_{ij}} w(r), \qquad 2<i\leqq t, \quad 1\leqq j<i.$$

Let us define $c_i=\max_{j<i} c_{ij}$ for $1<i\leqq t$, with $c_1=0$. We claim that for every i, $1<i<t$, there exists an $i'<i$ such that

$$(2.3) \qquad \sum_{r\in R_{i-1}} W(r)+\tfrac{6}{5}\sum_{r\in F_{ii'}\cup S_{ii'}} w(r)\geqq 1+\tfrac{6}{5}(c_{i-1}-c_{ii'}).$$

Using this claim (to be proved later), we can complete the proof as follows: We first observe that

$$\sum_{r\in L_1} h(r)\cdot W(r)\geqq \sum_{i=1}^{t} H_{i+1}\sum_{r\in R_i} W(r)+\sum_{i=1}^{t} W(f_i)(H_i-H_{i+1})$$
$$+\sum_{i=2}^{t} H_i\sum_{r\in F_{ii'}} W(r)+\sum_{i=2}^{t} H_{i+1}\sum_{r\in S_{ii'}} W(r).$$

This is because every regular item in B_i has height at least H_{i+1}, $1\leqq i\leqq t$ (recall that by convention $H_{t+1}=0$), while the first item in B_i has height H_i, and all elements of $F_{ii'}$ and $S_{ii'}$ have heights at least H_i and H_{i+1}, respectively, $1<i\leqq t$.

Now we use (2.1) and the fact that for all rectangles r, $W(r) \geqq \frac{6}{5}w(r)$, to conclude that the above sum is at least as large as

$$\sum_{i=2}^{t} \left[H_i \sum_{r \in R_{i-1}} W(r) + \tfrac{6}{5}(H_i - H_{i+1})\left(c_{ii'} + \sum_{r \in S_{ii'}} w(r) \right) + \tfrac{6}{5}H_i \sum_{r \in F_{ii'}} w(r) + \tfrac{6}{5}H_{i+1} \sum_{r \in S_{ii'}} w(r) \right]$$

$$\geqq \sum_{i=2}^{t} \left[H_i \sum_{r \in R_{i-1}} W(r) + \tfrac{6}{5}(H_i - H_{i+1})c_{ii'} + \tfrac{6}{5}H_i \sum_{r \in F_{ii'} \cup S_{ii'}} \right]$$

$$\geqq \sum_{i=2}^{t} [H_i + \tfrac{6}{5}H_i(c_{i-1} - c_{ii'}) + \tfrac{6}{5}(H_i - H_{i+1})c_{ii'}],$$

by (2.3). We thus conclude that

$$\sum_{r \in L_1} h(r) \cdot W(r) \geqq \sum_{i=2}^{t} [H_i - \tfrac{6}{5}H_{i+1}c_{ii'} + \tfrac{6}{5}H_i c_{i-1}]$$

$$\geqq H(T_1) - H_1 - \tfrac{6}{5} \sum_{i=2}^{t} H_{i+1}c_{ii'} + \tfrac{6}{5} \sum_{i=1}^{t-1} H_{i+1}c_i$$

$$\geqq H(T_1) - H_1 + \tfrac{6}{5} \sum_{i=2}^{t-1} H_{i+1}(c_i - c_{ii'}) \geqq H(T_1) - 1,$$

as desired, since $c_i \geqq c_{ii'}$ and $c_1 = H_{t+1} = 0$ by definition, and since $H_1 = 1$ by our normalization assumption (the tallest item has height exactly 1).

Thus it only remains to prove the claim, i.e., to show that for any i, $1 < i \leqq t$, there exists an $i' < i$ such that (2.3) holds. First, suppose $\sum_{r \in R_{i-1}} W(r) \geqq 1$. If $c_i \geqq c_{i-1}$, then for some $i' < i$, $c_i = c_{ii'} \geqq c_{i-1}$, yielding

$$\sum_{r \in R_{i-1}} W(r) + \tfrac{6}{5} \sum_{r \in F_{ii'} \cup S_{ii'}} w(r) \geqq 1 \geqq 1 + \tfrac{6}{5}(c_{i-1} - c_{ii'}),$$

as desired. If $c_i < c_{i-1}$, then $c_{i-1} > 0$ and hence $i - 1 \geqq 2$, so for some $i' < i - 1 < i$, $c_{i-1} = c_{i-1,i'}$. Then by (2.2) $F_{ii'} \cup S_{ii'}$ contains items of total width at least $c_{i-1,i'} - c_{ii'}$, and once again (2.3) follows.

Now suppose $\sum_{r \in R_{i-1}} W(r) < 1$. We apply Lemma 4 of [5], which says that if B is a set of one or more numbers x satisfying $c < x \leqq 1$ and $\sum_{x \in B} W(x) < 1$, then either $|B| = 1$ and the single element $x \in B$ satisfies $x \leqq \frac{1}{2}$, or else

$$\sum_{x \in B} x \leqq 1 - c - \tfrac{5}{6}\left(1 - \sum_{x \in B} W(x)\right).$$

Consider the set R_{i-1}. By the definition of the packing T_1, the first item in R_{i-1} must have width $\frac{1}{2}$ or less, which implies by (2.1) that $c_{i-1} \leqq \frac{1}{2}$. Since every item in R_{i-1} must have width exceeding c_{i-1}, the hypotheses of the lemma apply to the set of widths of items in R_{i-1}, with $c = c_{i-1}$, and so one of the two possibilities must apply. The first possibility cannot occur since it would imply that $w(f_i) > \frac{1}{2}$, contrary to the definition of T_1. Thus we conclude,

$$\sum_{r \in R_{i-1}} w(r) \leqq 1 - c_{i-1} - \tfrac{5}{6}\left(1 - \sum_{r \in R_{i-1}} W(r)\right).$$

Letting $i' = i-1$, we have

$$\begin{aligned} c_{ii'} &= 1 - \sum_{r \in R_{i-1}} w(r) - \sum_{r \in F_{ii'} \cup S_{ii'}} w(r) \\ &\geqq 1 - \left(1 - c_{i-1} - \tfrac{5}{6}\left(1 - \sum_{r \in R_{i-1}} W(r)\right)\right) - \sum_{r \in F_{ii'} \cup S_{ii'}} w(r) \\ &\geqq c_{i-1} + \tfrac{5}{6}\left(1 - \sum_{r \in R_{i-1}} W(r)\right) - \sum_{r \in F_{ii'} \cup S_{ii'}} w(r). \end{aligned}$$

which once again implies (2.3), as desired. This completes the proof of the claim and hence of the performance bound for FFDH.

Examples showing that the multiplicative constant 1.7 is the smallest possible follow immediately from the corresponding examples in [9], simply by taking the sizes given there to be widths and taking all heights to be 1. □

THEOREM 3. *Let L be any list of rectangles ordered by nonincreasing height such that no rectangle in L has width exceeding $1/m$ for some fixed $m \geqq 2$. Then*

$$\text{FFDH}(L) \leqq \left(1 + \frac{1}{m}\right)\text{OPT}(L) + 1.$$

Furthermore, the multiplicative constant $(1 + 1/m)$ is the smallest possible.

Proof. We would like to argue that for each block B_i of the FFDH packing, the total area A_i of rectangles in B_i is at least $(m/(m+1))H_{i+1}$. This would imply that

$$\text{OPT}(L) \geqq \sum_{i=1}^{t-1} A_i \geqq \frac{m}{m+1} \sum_{i=1}^{t-1} H_{i+1} = \frac{m}{m+1}(\text{FFDH}(L) - 1),$$

and the performance bound would follow. Unfortunately, it may be the case that for some $i < t$, $A_i < (m/(m+1))H_{i+1}$. It is the task of our proof to show that such shortfalls cannot hurt us.

Define A_{ij} and w_{ij}, $1 < i < t$ and $i \leqq j < i$, to be the total area and width, respectively, of items packed in B_j when the last regular item is packed in B_i (note that A_{ii} and w_{ii} are the total area and width, respectively, of regular items in B_i). Define $\Delta_i = \max(0, m/(m+1) - w_{ij})$. We shall prove that for all i, $1 \leqq i \leqq t$,

$$\sum_{j=1}^{i} A_{ij} \geqq \frac{m}{m+1} \sum_{j=1}^{i} H_{j+1} - \Delta_i H_{i+1}, \tag{3.1}$$

where, as usual, $H_{t+1} = 0$ by convention.

The proof is by induction. Inequality (3.1) clearly holds for $i = 1$, in which case it reduces to

$$A_{11} \geqq H_2\left(\frac{m}{m+1} - \max\left(0, \frac{m}{m+1} - w_{11}\right)\right),$$

which is true since $A_{11} \geqq H_2 \cdot w_{11}$. Consider any i satisfying $1 < i \leqq t$, and suppose (3.1) holds for $i-1$. If $\Delta_{i-1} = 0$, then we have

$$\sum_{j=1}^{i-1} A_{ij} \geqq \sum_{j=1}^{i-1} A_{i-1,j} \geqq \frac{m}{m+1} \sum_{j=1}^{i-1} H_{j+1},$$

and (3.1) will follow for arbitrary i in the same way it did for $i = 1$. So suppose $\Delta_{i-1} > 0$. There remain two cases to consider.

(i) B_i contains no regular item with width less than $1/(m+1)$. Then since the width of the first item in B_i is at least $1/(m+1)+\Delta_{i-1}$, and there must be at least m regular items, we have $w_{ii} \geqq m/(m+1)+\Delta_{i-1}$. We thus can conclude,

$$\begin{aligned} A_{ii} &\geqq \left(\frac{m}{m+1}+\Delta_{i-1}\right)H_{i+1}+\left(\frac{1}{m+1}+\Delta_{i-1}\right)(H_i-H_{i+1}) \\ &\geqq \frac{m}{m+1}H_{i+1}+\Delta_{i-1}H_i. \end{aligned}$$

Combining this with (3.1) for $i-1$, we obtain

$$\sum_{j=1}^{i} A_{ij} \geqq A_{ii}+\sum_{j=1}^{i-1} A_{i-1,j} \geqq \frac{m}{m+1}\sum_{j=1}^{i} H_{j+1}+\Delta_{i-1}H_i-\Delta_{i-1}H_i,$$

so (3.1) continues to hold for i in case (i).

(ii) B_i contains a regular item of width less than $1/(m+1)$. This can only happen if B_{i-1} received fallback items of total width exceeding Δ_{i-1} before the last regular item in B_i was packed. We thus have

$$A_{i,i-1}-A_{i-1,i-1} \geqq \Delta_{i-1}H_{i+1}.$$

Combining this inequality with (3.1) for $i-1$, and the fact that the first item in B_i has width at least $1/(m+1)+\Delta_{i-1}$, we obtain

$$\begin{aligned} \sum_{j=1}^{i} A_{ij} &\geqq A_{ii}+\sum_{j=1}^{i-1} A_{i-1,j}+(A_{i,i-1}-A_{i-1,i-1}) \\ &\geqq \left(\frac{m}{m+1}-\Delta_i\right)H_{i+1}+\left(\frac{1}{m+1}+\Delta_{i-1}\right)(H_i-H_{i+1})+\frac{m}{m+1}\sum_{j=1}^{i-1} H_{j+1}-\Delta_{i-1}H_i \\ &\qquad +\Delta_{i-1}H_{i+1} \\ &\geqq \frac{m}{m+1}\sum_{j=1}^{i} H_{j+1}-\Delta_i H_{i+1}. \end{aligned}$$

So (3.1) continues to hold for i in this case also.

By induction we can thus conclude that

$$\begin{aligned} A \geqq \sum_{j=1}^{t} A_{t,j} &\geqq \frac{m}{m+1}\sum_{j=1}^{t} H_{j+1}-\Delta_t H_{t+1} \\ &= \frac{m}{m+1}\sum_{j=1}^{t} H_{j+1} = \frac{m}{m+1}(\mathrm{FFDH}(L)-1). \end{aligned}$$

The desired performance bound follows.

Examples showing that the multiplicative constant of $(1+1/m)$ is the smallest possible once again follow in a straightforward way from the corresponding one-dimensional examples in [9]. □

Our next result concerns the interesting special case in which only *squares* are being packed. Note that this case does not have a nontrivial one-dimensional counterpart—if all items have the same height they must also all have the same width, and the problem becomes trivial. Note also that for this special case we can no longer make our normalizing assumption that the tallest item has height 1, and must settle for assuming that no square has size (size = width = height) exceeding 1. Our result for squares is as follows.

THEOREM 4. *For all lists of squares ordered by nonincreasing size,*

$$\mathrm{FFDH}(L) \leqq \tfrac{3}{2}\mathrm{OPT}(L) + 1.$$

Furthermore, the multiplicative constant $\frac{3}{2}$ *is the smallest possible.*

Proof. We divide the blocks of the FFDH packing into three groups. G_1 contains all blocks up to the highest block that contains a square of size exceeding $\frac{1}{2}$ but no square in the range $(\frac{1}{3}, \frac{1}{2}]$. (If no such block exists, G_1 is empty.) G_2 consists of all blocks above G_1 that contain at least one square in the range $(\frac{1}{3}, \frac{1}{2}]$. Note that each block of G_2 will thus contain exactly two items of size exceeding $\frac{1}{3}$, except possibly the last block, which may contain only one. G_3 contains all remaining blocks. Note that these blocks are above all of those of G_1 and G_2. For $i = 1, 2, 3$, let g_i denote the total height of blocks in G_1. We consider two cases, depending on the relative sizes of g_1 and g_3.

First, suppose $g_3 \leqq g_1/2$. Let us say that two items "overlap" in a packing if a horizontal line can be drawn which passes through the interiors of both items. Consider the first items in all the blocks of G_1. Note that none of them can overlap each other in an optimal packing, since all have size exceeding $\frac{1}{2}$. Furthermore, none of the items in all the blocks of G_2 with size exceeding $\frac{1}{3}$ can overlap any of these first squares from G_1 in an optimal packing. For if any such item from G_2 did, it would surely fit with the first square from the top block of G_1, and by definition of G_1 no such item did fit. Thus at least a total of g_1 of the height of the optimal packing is not overlapped by any item of size exceeding $\frac{1}{3}$ from G_2. If g' is the total height of such items, we therefore must have $\mathrm{OPT}(L) \geqq g_1 + g'/2$. Since every block of G_2 has its height determined by the size of its first square, which is less than $\frac{2}{3}$, and since every block except possibly the last has a second square of size exceeding $\frac{1}{3}$, we have $g' \geqq \frac{3}{2}g_2 - \frac{1}{3}$. We thus conclude that $\mathrm{OPT}(L) \geqq g_1 + \frac{3}{4}g_2 - \frac{1}{6}$.

If $g_3 \leqq g_1/2$, then $\mathrm{FFDH}(L) = g_1 + g_2 + g_3 \leqq \frac{3}{2}g_1 + g_2 \leqq \frac{3}{2}(g_1 + \frac{3}{4}g_2 - \frac{1}{6}) + \frac{1}{4} \leqq \frac{3}{2}\,\mathrm{OPT}(L) + \frac{1}{4}$, and we are done. So suppose $g_3 > g_1/2$.

For $i = 1, 2, 3$, let A_i denote the total area of squares in G_i, and let $A = A_1 + A_2 + A_3$. We will show that $\mathrm{FFDH}(L) \leqq \frac{3}{2}A + 1 \leqq \frac{3}{2}\,\mathrm{OPT}(L) + 1$, and thus complete the proof of the performance bound.

We first consider A_3. Let p be the size of the first (hence largest) square packed in G_3, and let m be such that $1/(m+1) < p \leqq 1/m$. Note that $m \geqq 3$. Since the widest square in G_3 has width (and height) no greater than $1/m$, the proof of Theorem 3 yields

$$A_3 \geqq \frac{m}{m+1}\left(g_3 - \frac{1}{m}\right) = \frac{m}{m+1}g_3 - \frac{1}{m+1}. \tag{4.2}$$

Now consider A_2. Any block of G_2 with two squares of width x and y, $x \geqq y \geqq \frac{1}{3}$, is at least $(x^2 + y^2)/x$ full. Since this expression is minimized at $x = y = \frac{1}{3}$, the block is at least $\frac{2}{3}$ full. Only the top block of G_2 may fail to meet these conditions. If so, that block has height at most $\frac{1}{2}$ (or else it would be in G_1). Thus

$$A_2 \geqq \tfrac{2}{3}g_2 - \tfrac{1}{3}(\tfrac{1}{2}) = \tfrac{2}{3}g_2 - \tfrac{1}{6}. \tag{4.3}$$

Finally, consider A_1. We claim that each block of G_1 is at least $(m+2)/(2m+2)$ full, and hence

$$A_1 \geqq \frac{m+2}{2m+2}g_1. \tag{4.4}$$

Let B be a block of G_1, and let $x > \frac{1}{2}$ be the size of the first square in B. If $m = 3$, either $x > \frac{2}{3}$ or B contains a second square of size at least $p > 1/(m+1)$ and hence is at least

$(x^2+(1/(m+1))^2)/x$ full. In both cases it is easy to verify that B is at least $(m+2)/(2m+2)=\frac{5}{8}$ full. For $m\geqq 4$, note that B is filled to width at least $1-p$ by squares of size at least p, and is therefore at least $(x^2+(1-x-1/m)/(m+1))/x$ full. Since this expression is minimized at $x=\frac{1}{2}$, we see that the block is at least

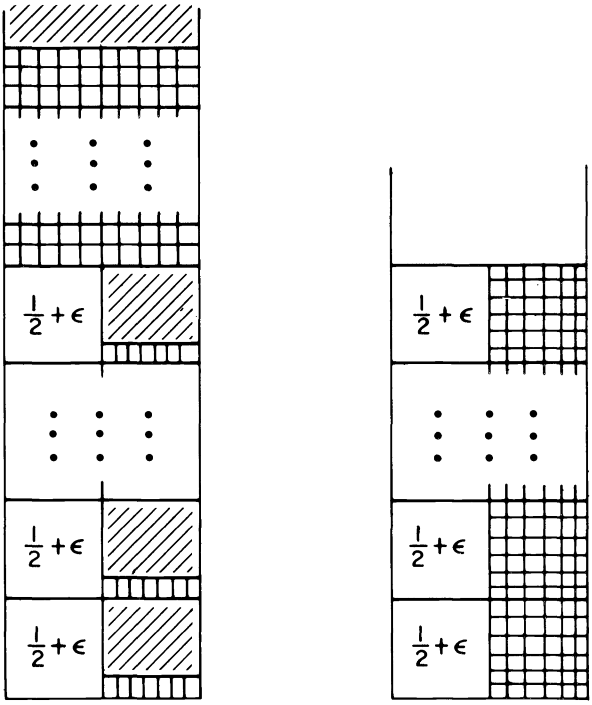
$$\frac{1}{2}+\frac{2(\frac{1}{2}-1/m)}{m+1}=\frac{m+1+2-4/m}{2m+2}\geqq\frac{m+2}{2m+2}$$

full. Thus (4.4) holds. Combining it with the formulas for A_3 and A_2 and using the fact that $g_3\geqq g_1/2$, we obtain

$$\begin{aligned}A_1+A_2+A_3&\geqq\frac{m+2}{2m+2}g_1+\tfrac{2}{3}g_2-\tfrac{1}{6}+\frac{m}{m+1}g_3-\tfrac{1}{4}\\&\geqq\frac{m+2}{2m+2}g_1+\tfrac{2}{3}g_2+\tfrac{2}{3}g_3+\left(\frac{m}{m+1}-\tfrac{2}{3}\right)g_3-\tfrac{2}{3}\\&\geqq\left(\left(\frac{m+2}{2m+2}+\tfrac{1}{2}\left(\frac{m}{m+1}-\tfrac{2}{3}\right)\right)\right)g_1+\tfrac{2}{3}(g_2+g_3-1)\\&\geqq\left(\frac{2m+2}{2m+2}-\tfrac{1}{3}\right)g_1+\tfrac{2}{3}(g_2+g_3-1)=\tfrac{2}{3}(g_1+g_2+g_3-1)\\&\geqq\tfrac{2}{3}(\mathrm{FFDH}(L)-1).\end{aligned}$$

The performance bound follows.

The construction of examples showing that the multiplicative constant $\frac{3}{2}$ is the best possible is quite simple (see Fig. 5). For suitably small $\varepsilon>0$, the list L consists of n



FFDH$(L)\geq(3n/4)-\varepsilon/2$ OPT$(L)\leq(n/2)+n\varepsilon$

FIG. 5. *Worst-case examples for Theorem* 4.

squares of size $\frac{1}{2}+\varepsilon$ followed by $\lfloor(\frac{1}{2}-\varepsilon)/\varepsilon\rfloor\cdot\lfloor n(\frac{1}{2}+\varepsilon)/\varepsilon\rfloor$ squares of size ε. Then $\mathrm{OPT}(L)\leqq(n/2)+n\varepsilon$ and

$$\mathrm{FFDH}(L)\geqq((n/2)+n\varepsilon)+\left[\frac{\frac{1}{2}-\varepsilon}{\varepsilon}\right]\left[\frac{n/2}{\varepsilon}\right]\varepsilon^2$$

$$\geqq(3n/4)-\varepsilon/2,$$

so $\mathrm{FFDH}(L)/\mathrm{OPT}(L)$ approaches $\frac{3}{2}$ as ε approaches 0. □

3. The Split-Fit algorithm. In this section we describe an algorithm which is slightly more complicated than those analyzed in the previous section but which, as we shall see, performs significantly better. We call this algorithm Split-Fit (abbreviated SF), and it operates as follows.

Let $m\geqq 1$ be the largest integer such that all the given rectangles have width $1/m$ or less. Divide the given list L of rectangles into two lists L_1 and L_2, both ordered by nonincreasing height, such that L_1 contains all the rectangles from L that have width greater than $1/(m+1)$ and L_2 contains all the rectangles from L that have width $1/(m+1)$ or less. Then pack the rectangles in L_1 using the FFDH algorithm (note that there will be *no* fallback items). Rearrange the blocks of this packing so that all blocks having total width more than $(m+1)/(m+2)$ are below all those blocks that have total width less than or equal to $(m+1)/(m+2)$. This leaves sufficient room that we can create a rectangle R of width $1/(m+2)$ to the right of the latter group of blocks, as shown in Fig. 6.

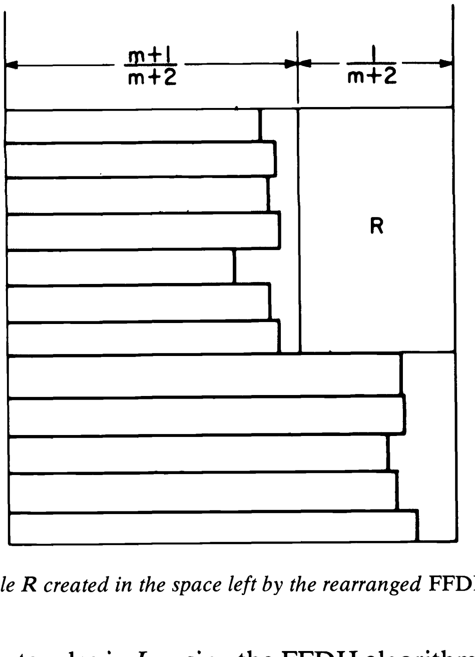


FIG. 6. *The rectangle R created in the space left by the rearranged* FFDH *packing of L_1.*

Finally, pack the rectangles in L_2 using the FFDH algorithm, allowing new blocks to be established within R (independently of the blocks in the packing of L_1) but not allowing any rectangle to overlap the top of R. For purposes of applying the FFDH algorithm, blocks within R are regarded as being below those established above the packing of L_1. Thus, in general, some of the rectangles from L_2 will be packed in R, and the remainder (those that fail to fit in R) will be packed above the packing for L_1.

The following theorem characterizes the worst-case performance of Split-Fit.

THEOREM 5. *For any list L of rectangles, if all rectangles in L have width less than or equal to $1/m$ (m an integer), then*

$$\mathrm{SF}(L) \leqq \frac{m+2}{m+1}\,\mathrm{OPT}(L)+2.$$

Furthermore, the multiplicative constant $(m+2)/(m+1)$ is the smallest possible.

Proof. Let T denote the region of the SF packing that contains rectangles from L_1 (not including R), and let S denote the region above the packing for L_1, thus dividing the SF packing into three disjoint regions, R, S and T. Let $H(S)$, $H(R)$ and $H(T)$ denote the heights of these regions. Index the blocks within each of these regions from bottom to top as $B_1(S), B_2(S), \cdots, B_{k(S)}(S)$; $B_1(T), B_2(T), \cdots, B_{k(T)}(T)$; and $B_1(R), B_2(R), \cdots, B_{k(R)}(R)$. For $X \in \{R, S, T\}$ and $1 \leqq i \leqq k(X)$, let $H_i(X)$ denote the height of block $B_i(X)$. We divide the proof into two cases, depending on whether S contains any rectangles of width $1/(m+2)$ or less.

Case 1. All rectangles in S have width larger than $1/(m+2)$.

In this case, since all rectangles in S also have width at most $1/(m+1)$, each block in S except possibly the last one contains exactly $m+1$ regular items (and no fallback items). The total height of all these rectangles is thus at least

$$\begin{aligned}(m+1)\sum_{i=1}^{k(S)-1} H_{i+1}(S) &\geqq (m+1)\sum_{i=2}^{k(S)} H_i(S)\\ &\geqq (m+1)(H(S)-1).\end{aligned}$$

Similarly, all rectangles in T have width greater than $1/(m+1)$ and no more than $1/m$, so each block in T except possibly the last contains exactly m regular items (and no fallback items). The total height of all these rectangles is at least

$$\begin{aligned}m\sum_{i=1}^{k(T)-1} H_{i+1}(T) &\geqq m\sum_{i=2}^{k(T)} H_i(T)\\ &\geqq m(H(T)-1).\end{aligned}$$

Now consider any optimal packing of L. We can divide this packing into "slices" by drawing a horizontal line from one side of the bin to the other through the top and bottom of each rectangle. For $0 \leqq i \leqq m$, let Z_i denote the cumulative height of all such slices that contain exactly i rectangles (subrectangles of the original rectangles) of width exceeding $1/(m+1)$. Notice that

$$\mathrm{OPT}(L) = \sum_{i=0}^{m} Z_i.$$

The total height of all rectangles from L having width greater than $1/(m+1)$ is then $\sum_{i=0}^{m} iZ_i$, so we must have

$$m(H(T)-1) \leqq \sum_{i=0}^{m} iZ_i.$$

Rewriting, we obtain

$$H(T) \leqq \frac{1}{m}\sum_{i=0}^{m} iZ_i + 1.$$

The total height of all rectangles from L having width greater than $1/(m+2)$ but no more than $1/(m+1)$ is at most $\sum_{i=0}^{m}(m+1-i)Z_i$, so we must have

$$(m+1)(H(S)-1)\leqq\sum_{i=0}^{m}(m+1-i)Z_i,$$

or rewriting,

$$H(S)\leqq\frac{1}{m+1}\sum_{i=0}^{m}(m+1-i)Z_i+1.$$

Combining these, we obtain

$$\begin{aligned}\mathrm{SF}(L)&=H(S)+H(T)\\&\leqq\frac{1}{m}\sum_{i=0}^{m}iZ_1+\frac{1}{m+1}\sum_{i=0}^{m}(m+1-i)Z_i+2\\&=\sum_{i=0}^{m}\left(\frac{i}{m}+1-\frac{i}{m+1}\right)Z_i+2\\&=\sum_{i=0}^{m}\left(1+\frac{i}{m(m+1)}\right)Z_i+2\\&\leqq\sum_{i=0}^{m}\left(1+\frac{1}{m+1}\right)Z_i+2=\frac{m+2}{m+1}\mathrm{OPT}(L)+2,\end{aligned}$$

and hence the claimed bound holds in Case 1.

Case 2. Some rectangle in S has width $1/(m+2)$ or less.

.We first claim the following:

$$A(T)\geqq\frac{m+1}{m+2}H(T)-\frac{1}{(m+1)(m+2)}H(R)-\frac{m}{m+2}. \tag{5.1}$$

Consider the FFDH packing of the rectangles of L_1 (which we later rearrange to form T in the process of performing Split-Fit). Index the bins from bottom to top as $B_1, B_2,\cdots, B_{k(T)}$, and denote the height of B_i by H_i, $1\leqq i\leqq k(T)$. Let T' be the subset consisting of those blocks which have width less than $(m+1)/(m+2)$. Since all items packed are regular items of width at least $1/(m+1)$, and all blocks except possibly the last have width at least $m/(m+1)$, we have the following (assuming by convention that $H_{k(T)+1}=0$):

$$\begin{aligned}A(T)&\geqq\frac{m+1}{m+2}\sum_{i=1}^{k(T)-1}H_{i+1}+\frac{1}{m+1}\sum_{i=1}^{k(T)}(H_i-H_{i+1})-\frac{1}{(m+1)(m+2)}\sum_{B_i\in T'}H_{i+1}\\&\geqq\frac{m+1}{m+2}(H(T)-H_1)+\frac{1}{m+1}H_1-\frac{1}{(m+1)(m+2)}\sum_{B_i\in T'}H_i\\&\geqq\frac{m+1}{m+2}H(T)-\frac{m}{m+2}-\frac{1}{(m+1)(m+2)}H(R),\end{aligned}$$

as desired, since $H_1\leqq 1$ and $H(R)=\sum_{B_i\in T'}H_i$.

Our second claim concerns region R:

$$A(R)\geqq\frac{1}{2(m+2)}(H(R)-2). \tag{5.2}$$

Since S by assumption contains items of width less than or equal to $1/(m+2)$ and rectangle R has width $1/(m+2)$, we must have $\sum_{i=1}^{k(R)} H_i(R) \geqq H(R)-1$. The packing of R must contain at least as much area as would its packing under NFDH, so by the proof of Theorem 1 (normalized to a bin width of $1/(m+2)$) we obtain

$$A(R) \geqq \tfrac{1}{2}\left(\sum_{i=1}^{k(R)} H_i(R) - H_1(R)\right) \cdot \frac{1}{m+2}$$

$$\geqq \tfrac{1}{2}(H(R)-1-H_1(R)) \cdot \frac{1}{m+2} \geqq \frac{1}{2(m+2)}(H(R)-2),$$

as desired.

Finally, let us turn to region S. This is packed by FFDH using items all of width $1/(m+1)$ or less. Thus by the proof of Theorem 3

$$A(S) \geqq \frac{m+1}{m+2}(H(S)-1).$$

Putting together (5.1), (5.2) and (5.3) we obtain

$$\mathrm{OPT}(L) \geqq A(T)+A(R)+A(S)$$

$$\geqq \frac{m+1}{m+2}H(T) - \frac{1}{(m+1)(m+2)}H(R) - \frac{m}{m+2} + \frac{1}{2(m+2)}H(R)$$

$$-\frac{1}{m+2} + \frac{m+1}{m+2}H(S) - \frac{m+1}{m+2}$$

$$\geqq \frac{m+1}{m+2}(H(T)+H(S)-2) = \frac{m+1}{m+2}(\mathrm{SF}(L)-2),$$

and the performance bound follows.

The fact that the multiplicative constant $(m+2)/(m+1)$ cannot be improved for Split-Fit follows immediately from the examples used to show that the bound of Theorem 3 is best possible. For a fixed value of m, we simply take the examples having rectangles of width no more than $1/(m+1)$ for which First-Fit achieves $((m+2)/(m+1)) \cdot \mathrm{OPT}(L)$ and add to the beginning of L, m rectangles of height 1 and width $1/m$. The performance of Split-Fit in region S is the dominant factor in such cases, so it performs essentially as First-Fit and achieves the same bound. □

The overall bound for Split-Fit in the unrestricted case is thus

$$\mathrm{SF}(L) \leqq \tfrac{3}{2}\,\mathrm{OPT}(L)+2,$$

which compares quite favorably with the corresponding bound for FFDH. Further improvements may well be obtainable if the idea of "splitting" implicit in Split-Fit is used in more elaborate ways [2], [8].

4. Discussion. The level-oriented packing methods analyzed here would seem to be wasteful of space, since they never consider packing rectangles in the space between the tops of the shorter items in a block and the top half of the block itself. Thus one might expect that the non-level-oriented Bottom-Left methods of [1], which pack by always placing the next rectangle from L as low as possible and then as far to the left as possible, would perform significantly better. However, straightforward extensions of the "checkerboard construction" from [1] can be used to show that these algorithms cannot satisfy an asymptotic performance bound better than $2 \cdot \mathrm{OPT}(L)$, regardless of whether L is ordered by decreasing height, increasing height, decreasing width, or increasing width. Thus, in contrast to expectations, the Bottom-Left methods turn out to be substantially *worse* than the level methods.

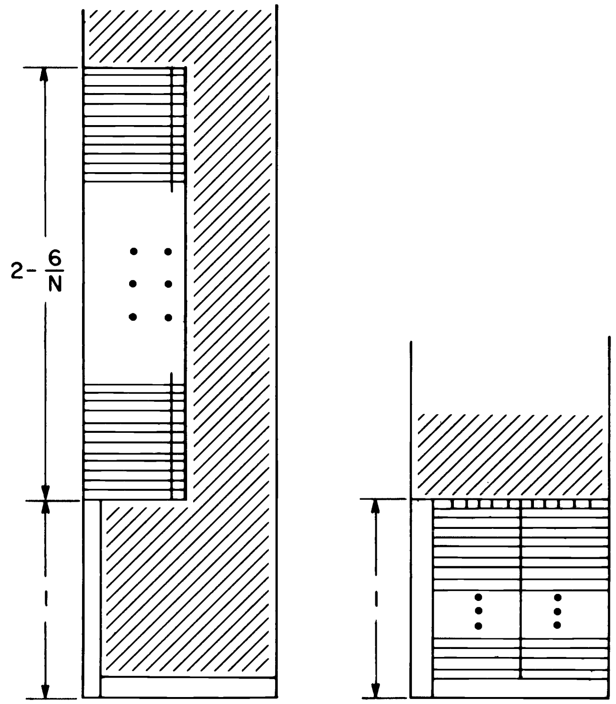
In a sense, the wasted space referred to above for level-oriented packing methods has an effect that is more apparent than real. In each of our proofs, we essentially showed that it was bounded by a constant, independent of OPT(L), and that is why the results for the one-dimensional case carried over so nicely. The wasted space does become significant, however, if we consider absolute bounds, bounds of the form $A(L) \leqq \alpha \cdot \mathrm{OPT}(L)$, where no additive constant is allowed. Such bounds may well be of interest in situations where optimal packings are not very tall, so we have proved our asymptotic results in such a way that absolute bounds can be derived as corollaries.

Note that in each of our proofs except the one for squares, we have assumed the height of the tallest rectangle to be exactly 1. This means that $\mathrm{OPT}(L) \geqq 1$, and hence a performance bound of the form

$$A(L) \leqq \beta \cdot OPT(L) + \gamma$$

implies the absolute bound

$$A(L) \leqq (\beta + \gamma) \cdot \mathrm{OPT}(L).$$



NFDH PACKING
$\mathrm{NFDH}(L) = 3 - \frac{6}{N}$

OPTIMAL PACKING
$\mathrm{OPT}(L) = 1$

FIG. 7. *Example of a list L with* $\mathrm{NFDH}(L) = (3 - 6\varepsilon)\,\mathrm{OPT}(L)$ *for any* $\varepsilon = 1/N$. *The list consists of one* $2\varepsilon \times 1$ *rectangle, one* $(1 - 2\varepsilon) \times 2\varepsilon$ *rectangle, and* $2N - 6$ *pairs of rectangles with dimensions* $(\frac{1}{2} - \varepsilon) \times \varepsilon$ *and* $3\varepsilon \times \varepsilon$.

Hence we obtain from Theorem 1 the previously known [2], [8] absolute bound $\mathrm{NFDH}(L) \leqq 3 \cdot \mathrm{OPT}(L)$ for all lists L. Fig. 7 provides an example to show that this is the best possible absolute bound (all the other absolute lower bounds we cite can be proved using the same basic ideas in conjunction with lower bound examples for the corresponding one-dimensional algorithms). Theorem 2 yields the absolute bound $\mathrm{FFDH}(L) \leqq 2.7 \cdot \mathrm{OPT}(L)$. This bound also can be shown to be tight. Theorem 3 yields the absolute bound $\mathrm{FFDH}(L) \leqq (2+1/m) \cdot \mathrm{OPT}(L)$ for all lists L in which no rectangle has width exceeding $1/m$, and this bound is also tight. Theorem 5 yields the corresponding absolute bound $\mathrm{SF}(L) \leqq (3+1/(m+1)) \cdot \mathrm{OPT}(L)$, and although this bound is not known to be tight for any fixed value of m, both lower and upper bounds approach 3 as m goes to infinity. Thus for all sufficiently large m, Split-Fit is worse than First-Fit in absolute ratio even though it is better asymptotically. The case for $m = 1$ is less clear. Our proof of Theorem 5 can be tightened to yield a bound of $\mathrm{SF}(L) \leqq \frac{3}{2}\mathrm{OPT}(L) + \frac{3}{2}$ in this case, by observing that in region T each block B_i contains a single regular item of height H_i, and so there is no wasted space above regular items in this region. This yields an absolute bound of $\mathrm{SF}(L) \leqq 3 \cdot \mathrm{OPT}(L)$, but the best lower bound we can prove is $2.7 \cdot \mathrm{OPT}(L)$ and we suspect that this, the same bound as for FFDH, is the correct answer. Sleator [10] has found a method similar to Split-Fit with an absolute bound of $2.5 \cdot \mathrm{OPT}(L)$.

The question arises: Are there any techniques which might be used to help reduce the wasted space in practice? Certain techniques are worth noting.

The first of these is that, instead of packing every block in a left-to-right manner, one might alternately pack blocks from left-to-right and then from right-to-left. Then the tallest rectangle in the block will be above the shortest, rather than the tallest, rectangle in the preceding block, and some reduction in total height might be achieved by dropping each rectangle until it touches some rectangle below it. Another approach would be to allow more general packings within blocks, say by allowing new, shortened blocks to be created in the space above the regular items in a block, which otherwise would remain empty.

However, the essentially one-dimensional nature of the worst case examples mentioned in the paper show that the asymptotic performance bounds cannot be improved by these modifications, and so one can only hope that they will do better in practice.

One possible modification of our basic model deserves mention. Suppose that, in packing, we are allowed to rotate rectangles by 90° if we so wish. We have not yet analyzed this variant in detail, but we note that Theorems 1 and 3, depending as they do solely on area-arguments for their proof, continue to hold even if $\mathrm{OPT}(L)$ is interpreted to be the minimum height packing with such rotations allowed. If we are to make use of the possibility of rotations in our algorithmic packings, one appealing heuristic would be to rotate all rectangles so that their width is no larger than their height, and then apply one of our standard algorithms which, according to Theorems 3 and 5, yield better bounds for smaller widths. Our result for squares (Theorem 4) indicates the limits of this approach for algorithm FFDH, but there are no doubt many interesting questions left to be answered.

Acknowledgment. The authors thank Brenda Baker for her many helpful comments and suggestions. In particular, she greatly simplified our proofs of Theorems 2 and 4, reducing the additive constants in each, and inspired us to derive similar improvements in the additive constants for Theorems 3 and 5, thus allowing us to make the above remarks about absolute bounds.

REFERENCES

[1] B. Baker, E. G. Coffman and R. L. Rivest, *Orthogonal packings in two dimensions*, this Journal, this issue, p. 846ff.

[2] B. Baker, personal communication.

[3] E. G. Coffman, ed., *Computer and Job/Shop Scheduling Theory*, John Wiley, New York, 1976.

[4] M. R. Garey, R. L. Graham and D. S. Johnson, *Performance guarantees for scheduling algorithms*, Oper. Res., 26 (1978), pp. 3–21.

[5] M. R. Garey, R. L. Graham, D. S. Johnson and A. C. Yao, *Resource constrained scheduling as generalized bin packing*, J. Comb. Th., 21 (1976), pp. 257–298.

[6] M. R. Garey and D. S. Johnson, *Approximation algorithms for combinatorial problems: an annotated bibliography* in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 41–52.

[7] ———, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[8] I. Golan, personal communication.

[9] 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. 299–325.

[10] D. Sleator, *A 2.5 times optimal algorithm for packing in two dimensions*, Info. Proc. Letters, 10 (1980), pp. 37–40.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0904-0014 $01.00/0

A POLYNOMIAL TIME ALGORITHM FOR SOLVING SYSTEMS OF LINEAR INEQUALITIES WITH TWO VARIABLES PER INEQUALITY*

BENGT ASPVALL† AND YOSSI SHILOACH‡

Abstract. We present a constructive algorithm for solving systems of linear inequalities (LI) with at most two variables per inequality. The time complexity of the algorithm is $O(mn^3I)$ on a random access machine, where m is the number of inequalities, n the number of variables, and I the size of the binary encoding of the input. The LI problem is of importance in complexity theory because it is polynomial time (Turing) equivalent to linear programming. The subclass of LI treated in this paper is of practical interest in mechanical verification systems.

Key words. linear inequalities, linear programming, polynomial time, algorithm, complexity, loop residue, hidden inequality

1. Introduction.

In this paper, we give a polynomial time algorithm for solving systems of linear inequalities where each inequality contains at most two variables. We start this chapter by introducing the problem and relating it to other problems and previous results. In Section 1.2, we present the general approach to solving the problem, and in Section 1.3 we define the representation and the complexity measure to be used throughout the paper.

1.1. Linear inequalities. Given a rational $m \times n$ matrix A and a rational m-vector $\mathbf{c}$, the linear inequalities (LI) problem is to determine whether or not there exists an n-vector $\mathbf{x}$ of rational numbers such that

$$A\mathbf{x} \leq \mathbf{c}. \tag{1}$$

If such a vector $\mathbf{x}$ exists, we say that the system is *satisfiable* and that $\mathbf{x}$ is a *feasible* vector; otherwise the system is *unsatisfiable*, and no feasible vectors exist. If the system is satisfiable, one can also ask for a feasible vector $\mathbf{x}$; the algorithm presented in this paper does supply a feasible vector if the system is satisfiable.

The LI problem is of importance in complexity theory. It is well-known that the linear programming (LP) problem—where one wants to maximize a linear function subject to linear inequality constraints—is polynomial time (Turing) equivalent to the LI problem [3, pp.287–288], [4], [11]. The complexity of the LP problem was one of the foremost open problems in theoretical computer science until Leonid Khachiyan announced in *Doklady Akademiĭa Nauk SSSR*, February 1979, a polynomial time algorithm for the LI problem (see Chapter 5).

We will use LI(k) to denote the class of LI problems with at most k variables per inequality. Any instance of the LI problem can be transformed into an equivalent LI(3) instance by introducing additional variables and constraints. (By using a binary

* Received by the editors February 14, 1979, and in final form March 14, 1980. This work was performed while both authors were at the Department of Computer Science, Stanford University, Stanford, California, and was supported by National Science Foundation Grants MCS-75-22870 and MCS-77-23738, and by a Chaim Weizmann Postdoctoral Fellowship. This paper was typeset by the authors on an Alphatype CRS phototypesetter via D. Knuth's TEX and METAFONT systems.

† Department of Computer Science, Cornell University, Ithaca, New York.
‡ IBM Scientific Center, Technion City, Haifa, Israel.

encoding scheme, the coefficients of the new problem can be restricted to $\{-1, 0, +1\}$ [4].) The transformations can be done with at most a polynomial increase in the number of variables and constraints. Thus an efficient algorithm for LI(3) is also an efficient algorithm for LP and vice versa.

In this paper, we present a constructive algorithm for LI(2). The time complexity of the algorithm is $O(mn^3I)$ on a random access machine, where I is the size of the binary encoding of the input. The LI(2) problem has practical applications in, for example, mechanical verification systems [8], [9], [10].

To denote a typical constraint of an LI(2) problem, we will use

$$ax + by \leq c, \tag{2}$$

where x and y are any two variables, and a, b, and c are rational numbers. Vaughan Pratt [10] has given an $O(n^3)$ algorithm for the case where the inequalities are of the form $x - y \leq c$. Robert Shostak [14] has generalized Pratt's idea to the general LI(2) case, but his algorithm has an exponential worst-case behavior. Greg Nelson [7] has given an $O(mn^{\lceil \log_2 n \rceil + 4} \log n)$ algorithm for LI(2), by using a modified Fourier-Motzkin elimination method.

1.2. Outline of the algorithm. It is well-known that the solution space of a system of linear inequalities forms a convex polyhedron. Let $\mathcal{X}(S)$ be the projection of the solution polyhedron on the x-axis for a given system S. We can also view $\mathcal{X}(S)$ as the set of values of x for which a solution to the entire system can be constructed. If we can find $\mathcal{X}(S)$, which is a convex interval on the x-axis, we can assign any value $\tilde{x} \in \mathcal{X}(S)$ to the variable x. This reduces the number of variables by one and yields a new system of inequalities that is satisfiable if and only if S is satisfiable. Hence, if S is satisfiable, a solution can be constructed by recursively solving systems of linear inequalities with fewer variables.

Our algorithm is related to a theorem by Shostak. In [14] he shows how to construct an undirected graph from a given system of inequalities such that the system is unsatisfiable if and only if the graph has what he terms an *infeasible simple loop.* Since we use the same graph construction in our algorithm, we will describe Shostak's ideas here.

Let S be a system of inequalities of the form (2), and let v_0 be an auxiliary *zero variable* that always occurs with zero coefficient—the only variable that can do this. Without loss of generality, we can thus assume that all the inequalities contain two variables. We construct the graph $G(S) = (V, E)$ with $n + 1$ vertices and m edges (counting multiple edges) as follows: (a) For each variable x occurring in S, add a vertex named x to $G(S)$. (We will use x to denote both the variable and the vertex where no confusion can occur.) (b) For each inequality $ax + by \leq c$ in S, add an undirected edge between x and y to $G(S)$, and label the edge with the inequality (Fig. 1).

Let P be a path of $G(S)$ determined by the vertices $v_1, v_2, \ldots, v_{l+1}$ and the edges $e_1, e_2, \ldots, e_l$. We define the *triple sequence* of P as

$$\langle a_1, b_1, c_1 \rangle, \langle a_2, b_2, c_2 \rangle, \ldots, \langle a_l, b_l, c_l \rangle,$$

where, for $1 \leq i \leq l$, $a_i v_i + b_i v_{i+1} \leq c_i$ is the inequality associated with e_i. If a_{i+1} and b_i have opposite signs for $1 \leq i < l$, then P is called *admissible.* Define $\langle a_P, b_P, c_P \rangle$, the *residue* of P, as

$$\langle a_P, b_P, c_P \rangle = \langle a_1, b_1, c_1 \rangle \odot \langle a_2, b_2, c_2 \rangle \odot \cdots \odot \langle a_l, b_l, c_l \rangle, \tag{3}$$

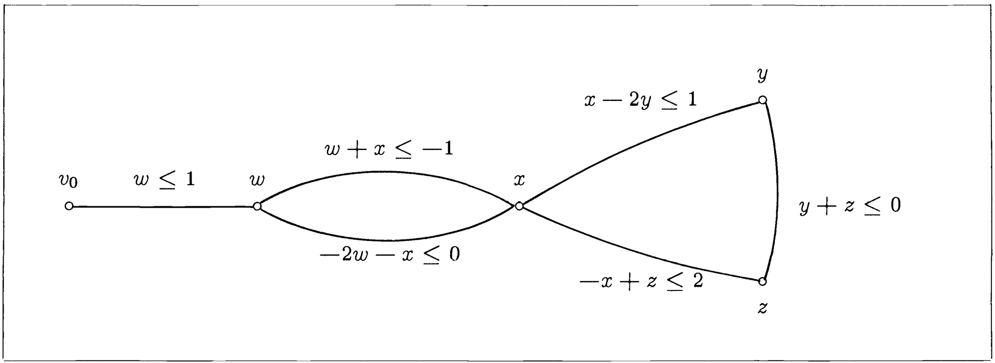


FIG. 1. $G(S)$ *for* $S = \{w \le 1, w + x \le -1, -2w - x \le 0, x - 2y \le 1, y + z \le 0, -x + z \le 2\}$.

where $\odot$ is the associative binary operator defined on triples by

$$(4) \qquad \langle a, b, c\rangle \odot \langle a', b', c'\rangle = \langle kaa', -kbb', k(ca' - c'b)\rangle \text{ and } k = a'/|a'|.$$

Intuitively, the operator $\odot$ takes two inequalities and derives a new inequality by eliminating a common variable; e.g., $ax + by \le c$ and $a'y + b'z \le c'$ imply $-aa'x + bb'z \le -(ca' - c'b)$ if $a' < 0$ and $b > 0$. Note that the residue imposes a direction on P even though the graph is undirected and that the signs of a_P and a_1 agree, as do the signs of b_P and b_l. The significance of path residues is formalized in the following lemma.

LEMMA 1 (Shostak). *If P is an admissible path with initial vertex x, final vertex y, and residue $\langle a_P, b_P, c_P\rangle$, then any point (i.e., assignment of rational values to variables) that satisfies the inequalities that label the edges of P satisfies $a_Px + b_Py \le c_P$.*

A path is called a *loop* if the initial and final vertices are identical. (A loop is not uniquely specified unless its initial vertex is given.) If all the intermediate vertices of a path are distinct, the path is *simple.* The reverse of an admissible path is always admissible, and the cyclic permutations of a loop are admissible if and only if a_1 and b_l have opposite signs. It follows that no admissible loop with initial vertex v_0 is permutable.

An admissible loop P with initial vertex x is *infeasible* if $a_P + b_P = 0$ and $c_P < 0$, since by Lemma 1 any solution of S must satisfy the unsatisfiable loop inequality $(a_P + b_P)x \le c_P$. Thus if $G(S)$ has an infeasible loop, the system of inequalities S is unsatisfiable. However, the converse is not true in general. We say that two systems of linear inequalities S and T are *equivalent* if they have the same solution polyhedron. Next we show how to extend S to an equivalent system S' such that $G(S')$ has an infeasible simple loop if and only if S is unsatisfiable.

For each vertex x of $G(S)$, and for each admissible simple loop P of $G(S)$ with $a_P + b_P \ne 0$ and initial vertex x, add a new inequality $(a_P + b_P)x \le c_P$ to S. We will call the new system S' the *Shostak extension* of S.

THEOREM 1 (Shostak). *Let S' be the Shostak extension of S. The system of inequalities S is satisfiable if and only if $G(S')$ has no infeasible simple loop.*

This theorem can easily be used to design an algorithm for LI(2). However, since the number of simple cycles in a graph on n vertices can be exponential in n, the worst-case behavior of the algorithm can be exponential in the number of variables. We will now outline the method we use in order to avoid examining all the cycles separately.

Assume that $G(S)$ is a graph for S. For each variable x define

$$(5)\quad \begin{aligned} xmin &= \max\{ \frac{c_P}{b_P} \mid P \text{ is an admissible path from } v_0 \text{ to } x \text{ in } G(S) \text{ and } b_P < 0 \}, \\ xmax &= \min\{ \frac{c_P}{b_P} \mid P \text{ is an admissible path from } v_0 \text{ to } x \text{ in } G(S) \text{ and } b_P > 0 \}, \end{aligned}$$

where we define $\max\{\ \} = -\infty$ and $\min\{\ \} = \infty$. Intuitively, $x \geq xmin$ is the most restrictive lower bound on x that we can derive using a chain of inequalities in S, where all but the first have two variables; a similar statement holds for $xmax$. Let $xmin^{(k)}$ and $xmax^{(k)}$ be defined in the same way as $xmin$ and $xmax$ but with P additionally restricted to length at most k. Thus $xmin^{(\infty)} = xmin$ and $xmax^{(\infty)} = xmax$.

A graph $G(S)$ is said to be *closed* for S if $[xmin, xmax] = \mathcal{X}(S)$ for all variables x or if $[xmin, xmax] = \mathcal{X}(S) = \emptyset$ (the empty set) for at least one variable x. Given a system S of linear inequalities, let T be a system of linear inequalities with the same variables as S. We say that the system T is a *closure* of S if the following is true: (a) The two systems S and T are equivalent, and (b) the graph $G(T)$ is closed for T. Thus if we can find a closure T of S in polynomial time and if there is a way to compute $[xmin, xmax]$ from $G(T)$ in polynomial time, we can construct a solution to S in polynomial time. In Section 3.1 we show that the Shostak extension S' is a closure of S and $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$, so $G(S')$ can in fact be used to reduce S to a smaller system. However, constructing $G(S')$ takes exponential time in the worst case.

We construct in polynomial time a modified extension S^* of S. The extension S^* will be a closure of S. Furthermore, we have $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$. This enables us to compute $[xmin^{(n)}, xmax^{(n)}] = \mathcal{X}(S^*) = \mathcal{X}(S)$ from $G(S^*)$ in polynomial time. (We assume that a solution exists; if no solution exists, it must be detected eventually.)

In the construction of the Shostak extension S' from $G(S)$, many redundant inequalities are added to S. These inequalities will not be added in the construction of S^*. From the graph $G(S)$, we will compute the non-redundant (i.e., the most restrictive) inequalities $x \geq xlow$ and $x \leq xhigh$ for each variable x using a binary search technique. In order to do this, we maintain, for each variable x of S, an interval $[xlow\!\downarrow, xlow\!\uparrow]$ such that either $xlow \in [xlow\!\downarrow, xlow\!\uparrow]$ or $xlow = -\infty$. Similarly, we maintain an interval $[xhigh\!\downarrow, xhigh\!\uparrow]$ such that either $xhigh \in [xhigh\!\downarrow, xhigh\!\uparrow]$ or $xhigh = \infty$. Initially, the intervals will be set to $[-\lambda, \lambda]$, where λ can be computed from the input.

The algorithm will guess values for the variables one at a time. A guessed value of x will be "pushed" through $G(S)$ in a breadth-first manner. This will give new—but not necessarily true—upper and lower bounds on the variables. By analysing the outcome of each guess, it is possible to chop the interval for either $xlow$ or $xhigh$ by at least half. There will also be a way to decide if a new and more restrictive bound on x can be derived. If this is not the case, the intervals can be coalesced into single points. After a finite—and in fact polynomial—number of guesses, we will be able to determine the true values of $xlow$ and $xhigh$ for any variable x of S.

Chapter 2 is devoted to the computation of $xlow$ and $xhigh$. After computing the values of $xlow$ and $xhigh$ for each variable x of S from the graph $G(S)$, we construct the extension S^* from S by adding the inequalities $x \geq xlow$ and $x \leq xhigh$. Given $G(S^*)$, it is rather straightforward to compute $xmin^{(n)}$ and $xmax^{(n)}$ using a breadth-first search. This is explained in detail in Chapter 3, where we give the LI(2)-Algorithm. In Chapter 4 we analyze the complexity of the algorithm and show that the number of guesses needed is bounded by a polynomial in the input size.

1.3. Representation and complexity measure. Our algorithm for LI(2) is polynomial in the size of the input. In order to establish exactly what this means, we have to say a few words about the way the input is represented and how the complexity is measured.

We assume that an instance of LI(2) is described as a string of inequalities of the form $ax + by \leq c$. Each rational number is represented as an ordered pair of integers and each variable by an integer between 1 and n. All integers are written in binary notation, and just enough additional symbols are allowed to delimit the input unambigously. The *input size* is the total length of the string describing a given instance.

Throughout this paper, we will use a random access machine (RAM) model. (For a detailed description see [1, pp.5–14].) The complexity measure will be the worst-case time using a uniform cost criterion, i.e., all elementary arithmetic operations and comparisons take one unit of time. However, we want to establish that the algorithm is polynomial also in the Turing machine sense. The following theorem (see, e.g., [1, p.33]) relates the complexity for the two machine models.

THEOREM 2. *The random access machine under logarithmic cost and the Turing machine are polynomially related models.*

In Chapter 4, where we examine the complexity of the algorithm, we will consider the logarithmic cost criterion as well. We will show that the intermediate results do not "blow up", establishing that the LI(2) problem is solvable in polynomial time on a Turing machine.

2. Finding the extension S^*. In this chapter we show how to find S^* from $G(S)$ using a binary search technique. We start by classifying different kinds of admissible loops and examining their behavior under guesses. In Section 2.2, we describe how to push a guessed value for a variable through $G(S)$. In Section 2.3, we construct the extension S^* and the graph $G(S^*)$ using the results from the previous sections.

2.1. Behavior of loops under guesses. From a linear inequality $ax + by \leq c$, we can derive an upper bound on y if $b > 0$ or a lower bound if $b < 0$, and the bounds depend on x if $a \neq 0$. Let us see how this can be employed in $G(S)$. Let P be an admissible path from x to y $(x, y \neq v_0)$ with residue $\langle a_P, b_P, c_P \rangle$. If we assign a value to x, we can get a constant bound on y from $a_P x + b_P y \leq c_P$. The residue of P is uniquely defined, since the operator $\odot$ is associative. Hence the bound on y is unique with respect to P. For another path P' we might get another bound on y. The trouble is that we do not want to compute the residues for all admissible paths between the two vertices x and y.

Let us ignore this problem for the moment and turn our attention to admissible loops, since they play a fundamental role in the algorithm. By making a guess for a variable x at one end of an admissible loop, we get a bound on the same variable at the other end of the loop. We now show how the guess and the result are related to the residue of the loop.

Let P be an admissible loop with residue $\langle a_P, b_P, c_P \rangle$ and initial and final vertex $x \neq v_0$. We call $a_P x + b_P x \leq c_P$ the *hidden inequality* of the loop. If $a_P + b_P \neq 0$, we can—and often will—write the hidden inequality of P as either $x \leq h$ or $x \geq h$, where $h = c_P/(a_P + b_P)$. Without knowing the residue of P, we can obtain information about its hidden inequality by guessing a value g for x, pushing the guess around P

(as described in the next section) and examining the result. There are four classes of admissible loops, denoted $\overset{-}{-}$, $\overset{+}{+}$, $\mp$, and $\pm$, and distinguished by the signs of a_P and b_P (the upper sign corresponds to the sign of a_P). We will now classify their behavior under guesses.

Let $h = c_P/(a_P + b_P)$, $r = (c_P - a_P g)/b_P$, and $\mu = -a_P/b_P$, so that r can be written as $r = \mu g + (1 - \mu)h$. The hidden inequality for a $\overset{+}{+}$ loop is of the form $x \leq h$. If we guess the value g for x, we get as the result the inequality $x \leq r$. The situation for $\overset{-}{-}$ loops is similar, but the inequalities are in the opposite directions. For these loops, we have $\mu < 0$, so either $g < h < r$, $r < h < g$, or $g = h = r$. Since the guess and the result are on opposite sides of h, we will call them the *flipping* loops (Figs. 2 and 3).

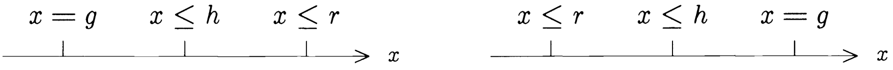


FIG. 2. *Flipping* $\overset{+}{+}$ *loops.*

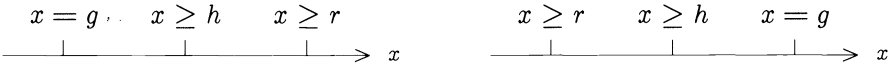


FIG. 3. *Flipping* $\overset{-}{-}$ *loops.*

The $\mp$ loops are slightly more complicated. We always get a result of the form $x \leq r$, but we have to distinguish three different cases for the hidden inequality: (a) If $a_P + b_P > 0$, we have $x \leq h$ and $0 < \mu < 1$, so the result lies in the interval between g and h (the *converging* case, Fig. 4a). (b) If $a_P + b_P < 0$, we have $x \geq h$ and $\mu > 1$, so h and r are on opposite sides of g (the *diverging* case, Fig. 4b). (c) If $a_P + b_P = 0$, the hidden inequality is $0 \leq c_P$; we get $r < g$ if $c_P < 0$ (i.e., the hidden inequality is a contradiction) and $r \geq g$ otherwise (the *contradiction* and *tautology* cases, Fig. 4c).

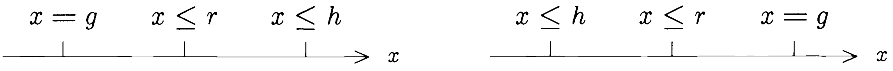


FIG. 4a. *Converging case for* $\mp$ *loops.*

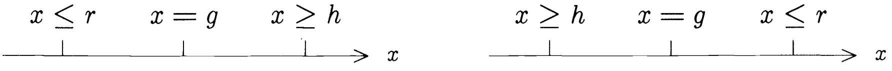


FIG. 4b. *Diverging case for* $\mp$ *loops.*

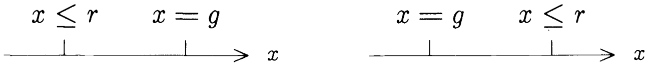


FIG. 4c. *Contradiction (left) and tautology cases for* $\mp$ *loops.*

TABLE 1.

Class of loop	Hidden ineq.	Result	Relations between g, r, and h	Figure
$\ddagger$	$x \leq h$	$x \leq r$	$g < h < r$, $g = h = r$, or $r < h < g$	2
$=$	$x \geq h$	$x \geq r$	$g < h < r$, $g = h = r$, or $r < h < g$	3
$\mp$, $a_P + b_P > 0$	$x \leq h$	$x \leq r$	$g < r < h$, $g = h = r$, or $h < r < g$	4a
$\mp$, $a_P + b_P < 0$	$x \geq h$	$x \leq r$	$r < g < h$, $g = h = r$, or $h < g < r$	4b
$\mp$, $a_P + b_P = 0$	$0 \leq c_P$	$x \leq r$	$r < g$ or $g \leq r$	4c
$\pm$, $a_P + b_P < 0$	$x \geq h$	$x \geq r$	$g < r < h$, $g = h = r$, or $h < r < g$	5a
$\pm$, $a_P + b_P > 0$	$x \leq h$	$x \geq r$	$r < g < h$, $g = h = r$, or $h < g < r$	5b
$\pm$, $a_P + b_P = 0$	$0 \leq c_P$	$x \geq r$	$g < r$ or $r \leq g$	5c

The $\pm$ loops are similar to the $\mp$ loops in their behavior. The difference is that the inequalities go in the opposite directions. Thus the result is always $x \geq r$, and again there are three cases for the hidden inequality (Figs. 5a,b,c).

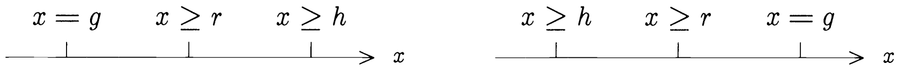


FIG. 5a. *Converging case for $\pm$ loops.*

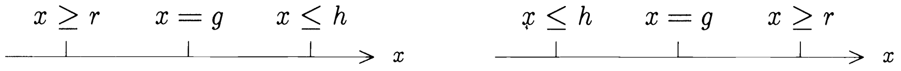


FIG. 5b. *Diverging case for $\pm$ loops.*

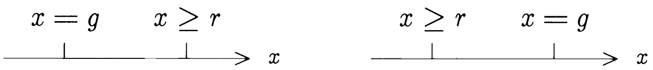


FIG. 5c. *Contradiction (left) and tautology cases for $\pm$ loops.*

For the converging and diverging cases of the $\mp$ loops and $\pm$ loops, we have ignored the possibility that $g = r$. In this case, we must have $h = g$ as for the $=$ loops and $\ddagger$ loops. Table 1 summarizes the loop results for reference in subsequent sections. Note that the directions of the hidden inequality and the resulting inequality disagree only for diverging loops.

2.2. Pushing guesses through $G(S)$. In Section 2.1, we saw that we could obtain partial information about the hidden inequality of a given admissible loop by guessing a value for a variable, "pushing" it around the loop, and examining the result for the same variable. We now describe how to "push" the guessed value through $G(S)$ in order to obtain the resulting inequality.

The hidden inequalities that we want to find are those that give the most restrictive lower and upper bounds for each variable; all other hidden inequalities are redundant. The algorithm will find the non-redundant hidden inequalities after examining only a polynomial number of all the hidden inequalities in $G(S)$.

Clearly we do not want to "push" a guessed value for x around each admissible simple loop with initial vertex x—there might be exponentially many. We will instead "push" the guessed value in a breadth-first way with at most n stages. This allows us to find a new and more restrictive hidden inequality involving x if one exists.

We call an edge e labeled $ay + bz \leq c$ a *positive* edge for y if $a > 0$ and a *negative* edge for y if $a < 0$. Note that the same edge can, for example, be positive for y and negative for z. Given a lower bound on y, we can derive a bound (lower or upper) on z using a positive edge for y. We say that the vertex y *sends* the lower bound on y over the edge e; the edge *transfers* this bound on y into a bound on z, which is then *received* by the vertex z. Similarly, a negative edge for y can transfer only an upper bound on y into a bound (lower or upper) on z.

A guessed value g for x will be spread like a rumor through $G(S)$. Whenever a vertex $y \neq x$ receives a new and more restrictive lower (or upper) bound, the vertex y *records* it as the current lower (or upper) bound on y. In the next stage of the algorithm, y sends this new bound out over all its positive (or negative) edges.

ALGORITHM 1 (The Grapevine). The input to the algorithm is the graph $G(S)$ and a guessed value g for x. The algorithm finds the most restrictive lower and upper bounds on x, which can be obtained from g using admissible loops of length at most n and with x occurring only as the initial and final vertex. The algorithm stores enough information so that the loops corresponding to the most restrictive lower and upper bounds can be reconstructed.

Step 1. [Send guess from x.] Let $i \leftarrow 1$. Transfer the guessed value g over all edges incident to x. For each vertex $y \neq x$, record the most restrictive lower and upper bounds received on y; record also the edges over which they were transferred together with the current stage number 1. (If the same bound was received over several edges, record one of them.)

Step 2. [Termination?] If $i < n$, set $i \leftarrow i + 1$ and go to Step 3. Otherwise the algorithm terminates and returns the current lower and upper bounds on x as the result.

Step 3. [Stage i.] For each vertex $y \neq x$, do the following: (a) If the currently most restrictive lower bound on y was recorded during stage $i — 1$, send it over all its positive edges. (b) If the currently most restrictive upper bound on y was recorded during stage $i — 1$, send it over all its negative edges.

Step 4. [Record new bounds.] For each vertex y, do the following: If a new, and more restrictive, lower (or upper) bound on y was received during stage i, record it as the current lower (or upper) bound; record also the edge that transferred the new bound together with the current stage number i. (If the same bound was received over several edges, record one of them.) Go to Step 2. □

Later we will need to trace the loop that gave the most restrictive lower or upper bound on x. This can be done simply by tracing the loop backwards. The algorithm stores for each vertex y the edges over which the lower and upper bounds were received. Since only a lower (upper) bound on y can have been sent out over a positive (negative) edge for y, there is no ambiguity between lower and upper bounds when tracing the loop backwards.

We call an admissible loop P of length at most n, with x occurring only as the initial and final vertex, and with hidden inequality $(a_P + b_P)x \leq c_P$ a *lower* loop for x if $b_P < 0$ and an *upper* loop for x if $b_P > 0$. Thus a lower loop is either a $\overset{-}{-}$ loop or a $\overset{+}{-}$ loop, and an upper loop is either a $\overset{+}{+}$ loop or a $\overset{-}{+}$ loop. Transferring a guess around a lower loop for x gives a lower bound on x as the result. Similarly, by using an upper loop for x, we get an upper bound on x. A lower (resp. an upper) loop for x is *optimal* with respect to g if for all other lower (resp. upper) loops P' for x we have $(c_{P'} - a_{P'}g)/b_{P'} \leq (c_P - a_P g)/b_P$ (resp. $(c_{P'} - a_{P'}g)/b_{P'} \geq (c_P - a_P g)/b_P$). Note that an admissible simple loop is either a lower or an upper loop for some vertex. The proof of the following lemma is straightforward and left to the reader.

LEMMA 2. *Algorithm 1 returns no lower (upper) bound on x if no lower (upper) loop for x exists. Otherwise there exists an optimal lower (upper) loop P for x with respect to g, and Algorithm 1 returns $x \geq r$ ($x \leq r$), where $r = (c_P - a_P g)/b_P$.*

2.3. Constructing S^* using binary search. In this section, we show how to construct the extension S^* of S without explicitly examining all admissible simple loops as is done in the construction of S', the Shostak extension of S. In the construction of S', an exponential number of inequalities involving the variable x may be added to S. However, all but at most two of these inequalities are redundant. We define

$$\begin{aligned} xlow &= \max\{\, h_P \mid P \text{ is a lower loop for } x \text{ with hidden inequality } x \geq h_P \,\}, \\ xhigh &= \min\{\, h_P \mid P \text{ is an upper loop for } x \text{ with hidden inequality } x \leq h_P \,\}. \end{aligned} \tag{6}$$

By using a binary search technique and the results from the two previous sections, we can compute the values of $xlow$ and $xhigh$ for all vertices x without examining all admissible simple loops. To obtain the extension S^*, we then add, for each variable x, the two inequalities $x \geq xlow$ and $x \leq xhigh$ to S.

It is now an easy task to construct $G(S^*)$ as follows: For each variable x, add two edges between x and v_0 to $G(S)$ and label the edges $x \geq xlow$ and $x \leq xhigh$ respectively. The graph $G(S^*)$ will be a subgraph of $G(S')$ in the following sense: (a) There is an edge between x and y ($x, y \neq v_0$) in $G(S^*)$ if and only if there is an edge between x and y in $G(S')$ with the same label. (b) If e is an edge between x and v_0 in $G(S^*)$ with label $x \geq c$, then there exists an edge e' between x and v_0 in $G(S')$ with label $x \geq c'$, where $c \geq c'$. (c) If e is an edge between x and v_0 in $G(S^*)$ with label $x \leq c$, then there exists an edge e' between x and v_0 in $G(S')$ with label $x \leq c'$, where $c \leq c'$.

In order to find the value of $xlow$, we maintain, for each variable x of S, an interval $[xlow\!\downarrow, xlow\!\uparrow]$ such that either $xlow \in [xlow\!\downarrow, xlow\!\uparrow]$ or $xlow = -\infty$. Similarly, we maintain an interval $[xhigh\!\downarrow, xhigh\!\uparrow]$ such that either $xhigh \in [xhigh\!\downarrow, xhigh\!\uparrow]$ or $xhigh = \infty$. Initially, the intervals will be set to $[-\lambda, \lambda]$, where λ can be computed from the input as explained in Section 4.1. We should stress that the interval $[xmin, xmax]$ might be nonempty although S is an infeasible system, but at least for one variable the interval will be empty.

The intervals $[xlow\!\downarrow, x^low\!\uparrow]$ and $[xhigh\!\downarrow, xhigh\!\uparrow]$ will be identical for some number of steps of the binary search, and we will use Algorithm 2 to chop their joint interval by at least half in each iteration. If the intervals become non-identical, we switch to Algorithm 3 and continue the search. The intervals will either be identical or have at most one point in common. We will always take the midpoint of an interval as a guess, and we use Algorithm 1 to provide the most restrictive lower and upper bounds with respect to this guess.

ALGORITHM 2 (Chop Joint Interval). The input to the algorithm is $G(S)$, a variable x, and an initial bound λ such that either $xlow \in [-\lambda, \lambda]$ or $xlow = -\infty$, and either $xhigh \in [-\lambda, \lambda]$ or $xhigh = \infty$. In each iteration the algorithm chops the joint interval for $xlow$ and $xhigh$ by at least half. The algorithm terminates when either an infeasible loop is found or the two intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ become non-identical. In the latter case $xlow\uparrow = xhigh\downarrow$, $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$, and $xhigh \in [xhigh\downarrow, xhigh\uparrow]$ or $xhigh = \infty$.

```
Step 1. [Initialize.] Let $xlow\downarrow \leftarrow -\lambda$ and $xhigh\uparrow \leftarrow \lambda$.
Step 2. [Distribute guess.] Let $g \leftarrow (xlow\downarrow + xhigh\uparrow)/2$. Use Algorithm 1 to find
        $x \geq r$ and $x \leq r'$—the most restrictive lower and upper bounds on $x$ with
        respect to $g$.
Step 3. [Chop or split?] If $x \geq r$ and $r > g$, go to Step 4. Otherwise, if $x \leq r'$ and
        $r' < g$, go to Step 5. Otherwise, go to Step 6.
Step 4. [Use lower result.] Trace the loop giving the bound $x \geq r > g$ and compute
        its hidden inequality. If $x \geq h$ and $h > g$, let $xlow\downarrow \leftarrow h$; else if $x \leq h$ and
        $h < g$, let $xhigh\uparrow \leftarrow h$; else terminate the algorithm due to an infeasible loop.
        Go to Step 2.
Step 5. [Use upper result.] Trace the loop giving the bound $x \leq r' < g$ and compute
        its hidden inequality. If $x \leq h$ and $h < g$, let $xhigh\uparrow \leftarrow h$; else if $x \geq h$ and
        $h > g$, let $xlow\downarrow \leftarrow h$; else terminate the algorithm due to an infeasible loop.
        Go to Step 2.
Step 6. [Split interval and terminate.] Let $xlow\uparrow \leftarrow xhigh\downarrow \leftarrow g$. Terminate the
        algorithm and return the intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ as the
        result. □
```

The following lemma will be very important in the proofs of correctness for Algorithms 2 and 3.

LEMMA 3. *The following two statements are equivalent:*

(a) *Algorithm 1 returns either $x \geq r > g$ or $x \leq r < g$.*

(b) *In $G(S)$ there exists a lower loop for x with hidden inequality $x \geq h > g$, or an upper loop for x with hidden inequality $x \leq h < g$, or an infeasible loop of length at most n with initial vertex x.*

Proof. From the behavior of different classes of loops (Table 1) and Lemma 2, it is easy to see that (a) implies (b). We will therefore only show that (b) implies (a).

If there exists a lower loop P for x in $G(S)$ with hidden inequality $x \geq h > g$, we know from Table 1 that we can derive either $x \geq r > g$ or $x \leq r < g$ from P. Since Algorithm 1 finds an optimal lower loop P' for x with respect to g, either $x \geq r' \geq r > g$ or $x \leq r' \leq r < g$ must be returned. The proof for an upper loop for x is similar.

If there exists an infeasible loop P of length at most n and with initial vertex x, we know, from the behavior of the contradiction case for $\mp$ loops and $\pm$ loops, that either $x \geq r > g$ or $x \leq r < g$ can be derived from P. Since P is either a lower or an upper loop for x, Algorithm 1 must return either $x \geq r' \geq r > g$ or $x \leq r' \leq r < g$ by Lemma 2. □

THEOREM 3. *In each iteration Algorithm 2 chops the joint interval for xlow and xhigh by at least half. Algorithm 2 terminates either because an infeasible loop has been found or the intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ have become non-identical. In the latter case $xlow\uparrow = xhigh\downarrow$, $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$, and $xhigh \in [xhigh\downarrow, xhigh\uparrow]$ or $xhigh = \infty$.*

Proof. It is easy to see that each time the algorithm returns to Step 2 the interval $[xlow\downarrow, xhigh\uparrow]$ has been chopped by at least half. Furthermore, the new endpoint corresponds to a bound on x that has been obtained from a new hidden inequality. Since the algorithm has to go to Step 2 in each iteration, and since there are only finitely many lower and upper loops of length at most n in $G(S)$, the algorithm must terminate. Clearly, if the algorithm terminates in Step 6, the two intervals for *xlow* and *xhigh* have only one point in common.

Let us now show that if the algorithm terminates in Step 4, then an infeasible loop has been found. Suppose the algorithm terminates in Step 4 without having found an infeasible loop. The loop that gave the result $x \geq r > g$ must then have either $x \geq h > g$ or $x \leq h < g$ as its hidden inequality according to Lemma 3. But in this case we do not terminate the algorithm, so we have a contradiction. The proof for termination in Step 5 is similar.

Finally, we show that if the algorithm terminates in Step 6, then either $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$. Suppose to the contrary that $xlow \notin [xlow\downarrow, xlow\uparrow]$ and $xlow \neq -\infty$. Then either $-\infty < xlow < xlow\downarrow$ or $xlow > xlow\uparrow = g$, where g is the last guess used in Step 2. By assumption $xlow \geq xlow\downarrow$ or $xlow = -\infty$ after Step 1. The only statements that change $xlow\downarrow$ are $xlow\downarrow \leftarrow h$ in Steps 4 and 5, and at those points we know that $x \geq h$. Thus the first case is not possible. If $xlow > xlow\uparrow = g$ there must be a lower loop with hidden inequality $x \geq h > g$. According to Lemma 3, Algorithm 1 must return either $x \leq r < g$ or $x \geq r > g$. But in this case we do not go to Step 6 from Step 3, so we have a contradiction. Hence we conclude that $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$. We omit the corresponding proof for *xhigh*; it is analogous to the one for *xlow*. □

By using Algorithm 2, we can compute an interval $[xlow\downarrow, xlow\uparrow]$ such that $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$, and $xhigh \geq xlow\uparrow$. We now show how to use this result to compute the correct value of *xlow* with an iterative technique similar to the one used in Algorithm 2. The main difference is the termination criterion. Guessing $xlow\downarrow$, the left endpoint of the interval for *xlow*, allows us to determine if there exists a lower loop with hidden inequality $x \geq h > xlow\downarrow$. If this is not the case, we can coalesce the interval $[xlow\downarrow, xlow\uparrow]$ into the single point $xlow\downarrow$.

The idea behind the termination test is actually the same as the idea for determining how to chop an interval. If we guess the midpoint of an interval, the derived inequality tells us whether the interval can be chopped by at least half; if we guess an endpoint, it tells us whether it can be chopped at all.

ALGORITHM 3 (Chop Lower Interval). The input to the algorithm is the graph $G(S)$, a variable x, and an interval $[xlow\downarrow, xlow\uparrow]$ such that $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$. Furthermore, $xlow\uparrow \leq xhigh$ is assumed. In each iteration, the algorithm chops the interval $[xlow\downarrow, xlow\uparrow]$ by at least half. The algorithm terminates if either an infeasible loop is found or *xlow* is determined to be either $xlow = xlow\downarrow$ or $xlow = -\infty$. In the latter case the point $xlow\downarrow$ is returned.

```
Step 1. [Chop or coalesce interval?] Let $g \leftarrow xlow\downarrow$. Use Algorithm 1 to find $x \geq r$
        and $x \leq r'$—the most restrictive lower and upper bounds on $x$ with respect
        to $g$. If $x \geq r$ and $r > g$, or $x \leq r'$ and $r' < g$, go to Step 3.
Step 2. [Coalesce interval and terminate.] Terminate the algorithm and return $xlow\downarrow$
        as the result.
Step 3. [New guess.] Let $g \leftarrow (xlow\downarrow + xlow\uparrow)/2$. Use Algorithm 1 to find $x \geq r$ and
        $x \leq r'$—the most restrictive lower and upper bounds on $x$ with respect to $g$.
Step 4. [Which end to chop?] If $x \geq r$ and $r > g$, go to Step 5. Otherwise, if $x \leq r'$
        and $r' < g$, go to Step 6. Otherwise, set $xlow\uparrow \leftarrow g$ and go to Step 3.
Step 5. [Use lower result.] Trace the loop giving the bound $x \geq r > g$ and compute
        its hidden inequality. If $x \geq h$ and $h > g$, let $xlow\downarrow \leftarrow h$ and go to Step 1.
        Otherwise, terminate the algorithm due to an infeasible loop.
Step 6. [Use upper result.] Trace the loop giving the bound $x \leq r' < g$ and compute
```

its hidden inequality. If $x \geq h$ and $h > g$, let $xlow\!\downarrow \leftarrow h$ and go to Step 1. Otherwise, terminate the algorithm due to an infeasible loop. □

LEMMA 4. *Algorithm 3 terminates; furthermore, in each iteration the interval* $[xlow\!\downarrow, xlow\!\uparrow]$ *is chopped by at least half.*

Proof. The test in Step 1 guarantees (according to Lemma 3 and the assumption that $xhigh \geq xlow\!\uparrow$) that whenever we get to Step 3 there exists either a lower loop for x with hidden inequality $x \geq h > xlow\!\downarrow$ or an infeasible loop of length at most n with initial vertex x. If there is an infeasible loop or h is in the right half of $[xlow\!\downarrow, xlow\!\uparrow]$, this will be detected in Step 4 (according to Lemma 3), and the algorithm will proceed to Step 5 or 6. Otherwise the right half of $[xlow\!\downarrow, xlow\!\uparrow]$ will be chopped, and the algorithm will return to Step 3. Thus h must fall in the right half of $[xlow\!\downarrow, xlow\!\uparrow]$ after finitely many executions of Steps 3 and 4, and the algorithm will proceed to Step 5 or 6.

To show that Algorithm 3 terminates, it remains to be shown that it cannot return to Step 1 infinitely many times. It is easy to see that each time the algorithm returns to Step 1, the interval $[xlow\!\downarrow, xlow\!\uparrow]$ has been chopped by at least half. Furthermore, the new left endpoint corresponds to a bound on x that has been obtained from a new hidden inequality. Since there are only finitely many lower loops in $G(S)$, the algorithm must terminate. □

THEOREM 4. *If there exist an infeasible loop of length at most n with initial vertex x in $G(S)$, then Algorithm 3 finds one and terminates. Otherwise the algorithm terminates and returns $xlow\!\downarrow$ such that either $xlow = xlow\!\downarrow$ or $xlow = -\infty$.*

Proof. From Lemma 4, we know that Algorithm 3 terminates. Let us first show that if the algorithm terminates in Step 5, an infeasible loop has been found. Suppose the algorithm terminates in Step 5 without having found an infeasible loop. The loop that gave the result $x \geq r > g$ must then have either $x \leq h < g$ or $x \geq h > g$ as its hidden inequality according to Lemma 3. By assumption $xlow\!\uparrow \leq xhigh$ at the beginning of the algorithm and $xlow\!\uparrow$ is never increased, so $x \leq h < g$ cannot be the case. If $x \geq h > g$ is the case in Step 5, we do not terminate the algorithm. Hence we have a contradiction. Termination in Step 6 is handled similarly.

We now show that if there exists an infeasible loop of length at most n with initial vertex x, then the algorithm terminates in Step 5 or 6. Suppose to the contrary that the algorithm terminates in Step 2 despite the fact that such a loop exists. Let g be the last guess used in Step 1. Lemma 3 tells us that Algorithm 1 must return either $x \geq r > g$ or $x \leq r < g$ in Step 1. But if this is the case, we do not go to Step 2. Hence we have a contradiction.

It remains to be shown that if the algorithm terminates in Step 2, then either $xlow = xlow\!\downarrow$ or $xlow = -\infty$. In order to prove this, we need the following claim, which is proved below.

- *Whenever the algorithm gets to Step 1, we have either $xlow \in [xlow\!\downarrow, xlow\!\uparrow]$ or $xlow = -\infty$.*

Let us assume that the algorithm terminates in Step 2, but neither $xlow = xlow\!\downarrow$ nor $xlow = -\infty$ is true. From the claim above, we conclude that $xlow > xlow\!\downarrow = g$, where g is the last guess used in Step 1. Thus there must be a lower loop P for x with hidden inequality $x \geq h > g$. Algorithm 1 would therefore, according to Lemma 3, return either $x \leq r < g$ or $x \geq r > g$ in Step 1. But in this case we do not go to Step 2, so we have a contradiction.

Proof of claim. By assumption, the claim is true the first time the algorithm reaches Step 1. Suppose the claim is violated for the first time when the algorithm returns to Step 1 the ith time. Then either $-\infty < xlow < xlow\downarrow$ or $xlow > xlow\uparrow$. The only statements that change $xlow\downarrow$ are $xlow\downarrow \leftarrow h$ in Steps 5 and 6. At those points we know that $x \geq h$, so the first case is not possible. Thus $xlow > xlow\uparrow$, which implies that there exists a lower loop P for x with hidden inequality $x \geq h > xlow\uparrow$. Hence $xlow\uparrow$ must have been erroneously changed since the previous execution of Step 1. The only statement that changes $xlow\uparrow$ is $xlow\uparrow \leftarrow g$ in Step 4, where g is the last guess used in Step 3. From existence of the lower loop P with hidden inequality $x \geq h > xlow\uparrow = g$, we know that Algorithm 1 must (according to Lemma 3) return either $x \leq r < g$ or $x \geq r > g$ as the result in Step 3. But then we would not change $xlow\uparrow$ in Step 4, so we have a contradiction. □

After using Algorithm 3, we know that either $xlow = xlow\downarrow$ or $xlow = -\infty$, but we do not know which alternative is the true one. However, if λ is sufficiently large, so that initially $xlow = -\infty$ or $xlow \in (-\lambda, \lambda)$, then $xlow = -\infty$ if and only if $xlow\downarrow = \lambda$.

We now know how to compute the correct value of $xlow$. The algorithm for computing $xhigh$ is almost identical to Algorithm 3 and is therefore omitted. Given the values of $xlow$ and $xhigh$ for each variable x of S, we can construct the extension S^* and the graph $G(S^*)$ as described at the beginning of this section.

3. The LI(2)-algorithm. The LI(2)-Algorithm is presented and proved to be correct in this chapter. We start by showing that the Shostak extension S' is a closure of S. In Section 3.1, we also show that $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$ for $G(S')$. In Section 3.2, we use these results to show that the extension S^* is a closure of S and that $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$ holds for $G(S^*)$. This allows us to present the polynomial algorithm for LI(2).

3.1. The Shostak extension S' is a closure of S. In this section, we show that S' is a closure of S and that $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$ for $G(S')$. If an infeasible simple loop in $G(S)$ is found during the construction of S', we know that S is unsatisfiable (Theorem 1) and define $[xmin^{(n)}, xmax^{(n)}]$ to be equal to the empty set $\emptyset$. The following lemma is immediate from the proof of Theorem 1.

LEMMA 5 (Shostak). *Let $xmin$ and $xmax$ be defined with respect to $G(S')$, let x be any variable in S, and let $\tilde{x}$ be any value such that $\tilde{x} \in [xmin, xmax]$. The system of inequalities S is satisfiable if and only if $S \bigcup \{x \leq \tilde{x}, x \geq \tilde{x}\}$ (i.e., S with $x = \tilde{x}$) is satisfiable.*

We call an algorithm for LI(2) *constructive* if it supplies a feasible vector whenever the system of inequalities is satisfiable. Lemma 5 does not directly lead to a constructive algorithm for LI(2). We now give three preliminary lemmas that allow us to prove Lemma 9—the constructive version of Lemma 5.

LEMMA 6. *Let S be satisfiable. The Shostak extension S' is a closure of S.*

Proof. In order to establish the lemma, we have to prove that S and S' are equivalent systems and that $G(S')$ is closed for S'. From Lemma 1 and the construction of the extension S', it is easy to see that S and S' are equivalent.

From Lemma 5, we have $[xmin, xmax] \subseteq \mathcal{X}(S)$. Thus $G(S')$ is closed for S' if we can show that $\mathcal{X}(S) \subseteq [xmin, xmax]$. Let $\tilde{x} \in \mathcal{X}(S)$. If $xmin = -\infty$, then obviously $\tilde{x} \geq xmin$. If $xmin > -\infty$, we know from the way S' is constructed that the inequality $x \geq xmin$ can be derived from S and that therefore any solution must satisfy it. Thus $\tilde{x} \geq xmin$. In the same way, we can show that $\tilde{x} \leq xmax$; hence $\mathcal{X}(S) \subseteq [xmin, xmax]$. □

LEMMA 7. *If there is an admissible path P from v_0 to x in $G(S')$, there is an* *admissible simple path Q from v_0 to x in $G(S')$ such that the sign of b_Q agrees with the sign of b_P.*

Proof. Given b_P, let Q be a shortest admissible path from v_0 to x in $G(S')$ such that the sign of b_Q agrees with the sign of b_P. We claim that Q is simple. Suppose to the contrary that Q is not simple. By the admissibility of Q, the intermediate vertices of Q are distinct from v_0. Thus Q can be expressed as $Q_1Q_2Q_3$, the concatenation of three admissible paths Q_1, Q_2, and Q_3, where Q_2 is a simple loop. Let $\langle a_i, b_i, c_i\rangle$, $1 \leq i \leq 3$, be the residues of the three paths and let $\langle a_Q, b_Q, c_Q\rangle$ be the residue of Q. We have two cases to consider, depending on whether Q_2 is permutable.

(a) If Q_2 is permutable, then $a_2b_2 < 0$ (i.e., a_2 and b_2 are of opposite signs). Since Q is admissible this implies that $b_1a_3 < 0$, so $Q' = Q_1Q_3$ is also admissible. Let $a_lx + b_ly \leq c_l$ be the inequality labelling the last edge of Q and recall that the signs of b_Q and b_l agree. Since $a_lx + b_ly \leq c_l$ also labels the last edge of Q', the signs of b_Q and $b_{Q'}$ agree, which contradicts the choice of Q as a shortest admissible path.

(b) If Q_2 is not permutable, then $a_2b_2 > 0$. By the definition of S', there exists an edge e labelled $(a_2 + b_2)y \leq c_2$ from v_0 to y in $G(S')$, where y is the initial vertex of Q_2. Since Q is admissibile, we have $b_2a_3 < 0$, which implies that $Q' = eQ_3$ is also admissible. The two paths Q and Q' both end with the same edge, so the signs of b_Q and $b_{Q'}$ agree. Thus, we have again a contradiction to the choice of the path Q. □

LEMMA 8. *If $G(S')$ has no infeasible simple loops, then $xmin^{(n)} = xmin$ and $xmax^{(n)} = xmax$ for each variable x.*

Proof. We show that $xmin^{(n)} = xmin$; the proof of the other case is similar. Trivially, $xmin^{(n)} \leq xmin$; it remains to be shown that $xmin^{(n)} \geq xmin$. If there is no admissible path from v_0 to x in $G(S')$ with $b_P < 0$, then $xmin = xmin^{(n)} = -\infty$. Let us therefore assume that such paths exist and let P be one for which $c_P/b_P = xmin$. According to Lemma 7, there exists an admissible path Q of length at most n from v_0 to x in $G(S')$ such that the signs of b_P and b_Q agree. Let Q be one for which $c_Q/b_Q = xmin^{(n)}$.

Add a new edge e between x and v_0 to $G(S')$ and label the new edge $x \leq xmin^{(n)}$. The only admissible loops of length at most n formed by adding this edge are of the form $Q'e$ (or eQ'), where Q' is an admissible path of length at most $n-1$ from v_0 to x with $b_{Q'} < 0$. From the edge e, we have $x \leq xmin^{(n)} = c_Q/b_Q$; from the path Q', we have $x \geq c_{Q'}/b_{Q'}$, where $c_Q/b_Q \geq c_{Q'}/b_{Q'}$ by the definition of Q and Q'. This implies that the hidden inequality $0 \leq c_Q/b_Q - c_{Q'}/b_{Q'}$ of $Q'e$ is a tautology. By assumption $G(S')$ did not contain any infeasible simple loops, and adding the edge e did not introduce any; thus the modified graph has no infeasible simple loops. It follows from Theorem 1 and Lemma 1 that $x \leq xmin^{(n)} = c_Q/b_Q$ and $x \geq xmin = c_P/b_P$ must be satisfiable simultaneously. We therefore have $xmin \leq xmin^{(n)}$. □

LEMMA 9. *Let S' be the Shostak extension of S. If S is satisfiable, then we have $[xmin^{(n)}, xmax^{(n)}] = \mathcal{X}(S)$ for each variable x; otherwise, there exists a variable x such that $[xmin^{(n)}, xmax^{(n)}] = \mathcal{X}(S) = \emptyset$.*

Proof. From Lemmas 6 and 8 it follows that this lemma is true when $G(S')$ has no infeasible simple loops, so let us assume that $G(S')$ has an infeasible simple loop P. Thus S is unsatisfiable (Theorem 1) and therefore $\mathcal{X}(S) = \emptyset$. We have two cases to consider depending on whether the initial vertex of P is v_0. If it is, then there exists a vertex x such that $x \geq xmin^{(n)}$, $x \leq xmax^{(n)}$, and $xmin^{(n)} > xmax^{(n)}$; thus $[xmin^{(n)}, xmax^{(n)}] = \emptyset$. Otherwise, $G(S)$ has an infeasible simple loop and by definition $[xmin^{(n)}, xmax^{(n)}] = \emptyset$. □

3.2. Constructing a solution. We will now show how to compute $xmin^{(n)}$ and $xmax^{(n)}$ from the graph $G(S^*)$ and how to use them to construct a feasible solution assuming that one exists. If the system of inequalities S is unsatisfiable, then no feasible vector exists and we will detect that during our construction. Before we describe the algorithm, we need the following theorem.

THEOREM 5. *Let $xmin^{(n)}$ and $xmax^{(n)}$ be defined with respect to $G(S^*)$. If S is satisfiable, then $[xmin^{(n)}, xmax^{(n)}] = \mathcal{X}(S)$ for each variable x; otherwise, there exists a variable x such that $[xmin^{(n)}, xmax^{(n)}] = \mathcal{X}(S) = \emptyset$.*

Proof. From Lemma 1 and the construction of the extension $G(S^*)$, it is easy to see that S^* and S are equivalent systems.

From the definition (6) of $xlow$ and $xhigh$, we see that the inequalities $x \geq xlow$ and $x \leq xhigh$ are as restrictive as any hidden inequality of an admissible simple loop. Thus the inequalities added to S in the construction of S^* are at least as restrictive as those added to S in the construction of S'. Hence, if $xmin^{(n)}$ and $xmax^{(n)}$ are defined with respect to $G(S^*)$, then $[xmin^{(n)}, xmax^{(n)}]$ is a subinterval of the corresponding interval defined with respect to $G(S')$. But $x \geq xmin^{(n)}$ and $x \leq xmax^{(n)}$ can be derived from S^*, so they must clearly be satisfied in any feasible solution. The theorem thus follows from Lemma 9. □

In the algorithms presented in Chapter 2, the auxiliary zero variable v_0 has never had any significance since it always occurs with zero coefficient. Thus all inequalities in S with only one variable have been ignored so far. Now is the time for v_0 to play its role.

The algorithm to compute $xmin^{(n)}$ and $xmax^{(n)}$ will be quite similar to Algorithm 1. It works on $G(S^*)$ instead of $G(S)$. The algorithm starts by sending the guess $v_0 = 0$ (any other value will do) from v_0 and recording the most restrictive lower and upper bounds received at vertices adjacent to v_0. What this intuitively means is the following: If x is adjacent to v_0, the lower (or upper) bound on x received is the most restrictive lower (or upper) bound obtained from the original inequalities with only one variable and from the inequality $x \geq xlow$ (or $x \leq xhigh$) added in the construction of $G(S^*)$. The algorithm then proceeds to transfer new and more restrictive bounds on variables other than v_0 in a breadth-first way with n stages.

ALGORITHM 4 (The Projector). The input to the algorithm is the graph $G(S^*)$. The algorithm finds $xmin^{(n)}$ and $xmax^{(n)}$ for each variable $x \neq v_0$.

Step 1. [Send guess from v_0.] Let $i \leftarrow 1$. Transfer the value $g = 0$ over all edges incident to v_0. For each vertex $x \neq v_0$, record the most restrictive lower and upper bounds received on x.

Step 2. [Termination?] If $i < n$, set $i \leftarrow i + 1$ and go to Step 3. Otherwise, the algorithm terminates and returns for each variable $x \neq v_0$ the current lower and upper bound on x as the result.

Step 3. [Stage i.] For each vertex $x \neq v_0$ do the following: (a) If the currently most restrictive lower bound on x was recorded during stage $i - 1$, send it over all its positive edges. (b) If the currently most restrictive upper bound on x was recorded during stage $i - 1$, send it over all its negative edges. (c) Record new, and more restrictive, bounds on x. Go to Step 2. □

LEMMA 10. *Algorithm 4 computes $xmin^{(n)}$ and $xmax^{(n)}$ for each variable $x \neq v_0$.*

The proof of Lemma 10 is essentially the same as the proof of Lemma 2 and is omitted. Having found the values of $xmin^{(n)}$ and $xmax^{(n)}$, we can use Theorem 5 to construct a feasible solution if one exists. The theorem and the definition of $\mathcal{X}(S)$ tell us that if x is any variable of S and $\tilde{x}$ is any value such that $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}]$, then S is satisfiable if and only $S \bigcup \{x \leq \tilde{x}, x \geq \tilde{x}\}$ is satisfiable. Adding the two inequalities $x \leq \tilde{x}$ and $x \geq \tilde{x}$ forces x to be equal to $\tilde{x}$ in any solution, so we have the following constructive algorithm to decide whether S is satisfiable.

ALGORITHM 5 (LI(2)-Algorithm). The input to the algorithm is the system of inequalities S. The algorithm determines whether S is satisfiable. If S is satisfiable, the algorithm supplies a feasible vector.

Step 1. [Construct $G(S)$.] Construct the graph $G(S)$ for S as described in Section 1.2. Compute λ from S as described in Section 4.1. Mark all variables *unassigned.*

Step 2. [Construct S^*.] Use Algorithms 1, 2, and 3 to compute $xlow$ and $xhigh$ for each variable $x \neq v_0$. Add the corresponding inequalities $x \geq xlow$ and $x \leq xhigh$ to S.

Step 3. [Construct $G(S^*)$.] For each variable $x \neq v_0$, add two edges between x and v_0 to $G(S)$ and label the edges $x \geq xlow$ and $x \leq xhigh$ respectively.

Step 4. [Compute $\mathcal{X}(S)$.] Use Algorithm 4 to compute $[xmin^{(n)}, xmax^{(n)}]$ for each variable $x \neq v_0$.

Step 5. [Terminate?] If all variables are marked *assigned,* terminate the algorithm and return the constructed feasible vector. Otherwise, let x be any variable marked *unassigned.*

Step 6. [Assign value to x.] If the interval $[xmin^{(n)}, xmax^{(n)}]$ is empty, terminate the algorithm and return *unsatisfiable.* Otherwise, mark x *assigned* and let $x \leftarrow \tilde{x}$, where $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}]$.

Step 7. [Reduce the system.] Add two edges between x and v_0 to $G(S^*)$ and label the edges $x \geq \tilde{x}$ and $x \leq \tilde{x}$ respectively. Go to Step 4. □

4. Complexity.

In this chapter we analyze the complexity of the LI(2)-Algorithm. We first show how to compute λ, which is needed as an initial bound in the algorithm and also enters into the analysis. In the same section, we show that the algorithm runs in $O(mn^3|I|)$ time on a random access machine, where $|I|$ is the input size. We then turn to the Turing machine model in Section 4.2 and show that the algorithm is also polynomial time on a Turing machine.

4.1. Random access machine model.

In this section we show that the LI(2)-Algorithm runs in polynomial time on a random access machine. We start by showing how to compute λ, which is needed in the algorithm and also enters into the complexity analysis.

Let I denote an instance of LI(2), and let $|I|$ be the length of the string encoding I. Let κ be the largest absolute value of an integer used to represent the rational coefficients in the input. Clearly, $\log_2 \kappa < |I|$.

We will now determine λ such that either $xlow \in (-\lambda/2, \lambda/2)$ or $xlow = -\infty$ (the reason for dividing by two will be explained below). If $xlow \in (-\lambda/2, \lambda/2)$, then by (6) there exists a lower loop P for x with hidden inequality $x \geq h_P = c_P/(a_P + b_P)$. Since a lower loop is of length at most n, it follows from (3), the definition of κ, and the fact that the coefficients are rational numbers that $a_P = 0$ or $\kappa^{-n} \leq |a_P| \leq \kappa^n$, $b_P = 0$ or $\kappa^{-n} \leq |b_P| \leq \kappa^n$, and $c_P = 0$ or $\kappa^{-3n} \leq |c_P| \leq n\kappa^n$. Since $a_P + b_P \neq 0$, we have $\kappa^{-2n} \leq |a_P + b_P| \leq 2\kappa^n$. Thus, $\kappa^{-4n}/2 \leq |h_P| = |c_P/(a_P + b_P)| \leq n\kappa^{3n}$ if $h_P \neq 0$. Let $\lambda = 3n\kappa^{3n}$; we have either $xlow \in (-\lambda/2, \lambda/2)$ or $xlow = -\infty$. Obviously, either $xhigh \in (-\lambda/2, \lambda/2)$ or $xhigh = \infty$ for the same choice of λ.

From the previous chapters, we know that the LI(2)-Algorithm terminates, but we do not know how many iterations are performed in Algorithms 2 and 3. We now bound the total number of iterations in the two algorithms. Let $x \geq h = c_P/(a_P+b_P)$ and $x \geq h' = c_{P'}/(a_{P'}+b_{P'})$ be the hidden inequalities of two lower loops for x. By using our previous bounds on a_P, b_P, and c_P (again together with the fact that they are rational numbers), we find that h and h' must be equal if $|h - h'| < (\kappa^{-4n}/2)^2$. Clearly, this argument does not depend on the fact that we have two lower loops—it holds as well for two upper loops, or one upper and one lower loop. Let $\epsilon = \kappa^{-8n}/4$. If h and h' correspond to lower and/or upper loops for some variable, then $h \neq h'$ implies $|h - h'| \geq \epsilon$.

We know that in each iteration of Algorithm 2 the joint interval for $xlow$ and $xhigh$ is chopped by at least half and that the new endpoint is obtained from a hidden inequality of a lower or an upper loop. Thus, if the interval is of size less than ϵ, the algorithm must terminate. Initially the interval is of size 2λ, so the maximum number of iterations in Algorithm 2 is $\lceil \log_2(2\lambda/\epsilon)\rceil = O(\log n + n \log \kappa) = O(n|I|)$.

Let us now turn to Algorithm 3. From Lemma 4, we know that each time Algorithm 3 returns to Step 3 the interval $[xlow\downarrow, xlow\uparrow]$ has been chopped by at least half and that there exists a hidden inequality $x \geq h$ with $h > xlow\downarrow$. We also know that $xlow\downarrow$ has been obtained from a hidden inequality of a lower loop for x unless $xlow\downarrow = -\lambda$. By the choice of λ, we can only have $xlow\downarrow = -\lambda$ the first time the algorithm gets to Step 3. This follows since initially the interval $[xlow\downarrow, xlow\uparrow]$ is of size at most λ. Therefore $g = (xlow\downarrow + xlow\uparrow)/2 \leq -\lambda/2$ in Step 3 if $xlow\downarrow = -\lambda$. By definition $h > -\lambda/2$. Thus h lies in the right half of $[xlow\downarrow, xlow\uparrow]$, and $xlow\downarrow \leftarrow h$ will be executed in Step 5 or 6 during the first iteration.

We conclude that Algorithm 3 returns to Step 3 at most $\lceil \log_2(\lambda/\epsilon))\rceil = O(\log n + n \log \kappa) = O(n|I|)$ times; the same bound holds when computing $xhigh$. Thus the total number of iterations to compute $xlow$ and $xhigh$ is at most $O(n|I|)$. It is straightforward to see that Algorithm 1 takes $O(mn)$ time to perform the n stages of the breadth-first pushing of the guess. For both Algorithms 2 and 3 the amount of work in each iteration is dominated by the call of Algorithm 1. Hence the time to compute $xlow$ and $xhigh$ is at most $O(mn^2|I|)$.

Let us now bound the amount of time necessary to reduce the number of variables by one using Algorithm 5. The time to construct the graph $G(S)$ from S is $O(m+n)$. Since there are n different variables the total time to compute $xlow$ and $xhigh$ for all variables x (i.e., to find S^*) is $O(mn^3|I|)$. We can then construct $G(S^*)$ from $G(S)$ in $O(n)$ time. To compute $xmin^{(n)}$ and $xmin^{(n)}$ for each variable x requires one call of Algorithm 4, which is of complexity $O(mn)$ (the same as Algorithm 1). The remaining steps of Algorithm 5 take $O(m+n)$ time, so the total time to reduce the number of variables by one is at most $O(mn^3|I|)$. Since the construction of the extension S^* only has to be done once and the total contribution to the running time from the other steps is $O(mn^2)$, we have the following theorem.

THEOREM 6. *The time complexity of the LI(2)-Algorithm is $O(mn^3|I|)$ on a random access machine with uniform cost criterion.*

4.2. Turing machine model. Since the Turing machine model is polynomially related to the RAM model under the logarithmic cost criterion, we will start by examining the complexity of the algorithm on a RAM under logarithmic cost. In order to simplify the discussion, we assume that all the coefficients in the input are integers; otherwise, we multiply through by their least common denominator γ ($\log_2 \gamma < |I|$). We will also make one assumption on the way we choose $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}]$ in Step 7 of the LI(2)-Algorithm. Let $\tilde{x}$ be a finite endpoint of $[xmin^{(n)}, xmax^{(n)}]$ if one exists; otherwise, let $\tilde{x} = 0$. (Without any restriction, one could choose a value $\tilde{x}$ that requires an exponential number of bits to represent.)

The number of memory cells required by the LI(2)-Algorithm on a RAM is at most $O(m + n^2)$, where the non-linear term comes from Step 4 of Algorithm 1. We will now bound the size of the numbers that can occur. It is enough to consider results from operations involving multiplication. From (3) it follows that a_P, b_P, and c_P are integers if the coefficients in the input are integers. Since we only consider paths of length at most n in the algorithm, we know that a_P, b_P, and c_P are bounded in magnitude by $n\kappa^n$. Thus the bounds on the variables obtained from lower or upper loops can be represented as pairs of integers whose magnitudes are bounded by $n\kappa^n$. The only other intermediate results obtained using multiplications are the bounds resulting from guesses in Algorithm 1 and in Algorithm 4. A guessed value can always be represented by a pair of integers whose magnitudes are bounded by λ. It is easily seen that the results obtained can be represented as pairs of integers whose magnitudes are bounded by $n\kappa^n\lambda$. We conclude that all operands are of size $O(\log m + \log n + n \log \kappa + \log \lambda + \log \gamma) = O(n|I|)$ bits. Thus the LI(2)-Algorithm runs in polynomial time on a RAM under the logarithmic cost criterion, and by Theorem 2 we have the following theorem.

THEOREM 7. *The time complexity of the LI(2)-Algorithm is polynomial in the size of the input on a Turing machine.*

5. Conclusions. We have presented a new technique for tackling the LI(2) problem. As we have pointed out, extending our method to systems of linear inequalities with three variables per inequality will yield an algorithm for LP. Therefore this extension is apt to be quite hard to find. Another possible extension is to allow a fixed number of inequalities with more than two variables.

Our main concern has been that the algorithm runs in polynomial time, so we have not mentioned several short-cuts that can reduce the practical running time. One of them is to use inequalities of the forms $x \geq c$ and $x \leq c'$, which are given in the system S, as initial bounds for $xlow\downarrow$ and $xhigh\uparrow$ instead of the theoretically derived λ. Other modifications that make the algorithm more practical and reduce the time bound to $O(mn^2|I|)$ are discussed in [2]. For example, it is possible to avoid computing $xlow$ and $xhigh$ and instead compute $xmin$ and $xmax$ directly using an iterative method similar to the one presented in this paper. One advantage of this approach is that a value $\tilde{x} \in [xmin\uparrow, xmax\downarrow]$ can be substituted for the variable x without first having to compute the exact values of $xmin$ and $xmax$.

Recently, Leonid Khachiyan has determined the complexity of the LI problem by presenting a polynomial time algorithm [6]. His algorithm is based on the method of shrinking ellipsoids by Shor [12], [13] and Judin and Nemirovskii [5]. In the worst case, Khachiyan's algorithm requires $O(n^3(m + n^2)I)$ arithmetic operations ($+$, $-$, $\times$, $/$, and $\sqrt{\ }$); each operation is carried out to $O(nI)$ bits of accuracy. Khachiyan's algorithm is a beautiful theoretical discovery; however, early reports indicate that it is not a practical algorithm.

Although the LI(2) algorithm and Khachiyan's algorithm use different approaches, they are both iterative methods using the fact that the input coefficients are rational numbers. (For the LI(2) algorithm, this fact is used only to establish the time bound and not validity; i.e., the algorithm is valid in exact real arithmetic). This means that a solution vector cannot have arbitrarily large entries (unless they are unbounded); compare λ in the presented algorithm with Khachiyan's initial hypersphere. Furthermore, the rational coefficients add a certain "discreteness" to the problem; compare ϵ with Khachiyan's lower bound on the volume of the solution space.

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] B. ASPVALL, *Efficient algorithms for certain satisfiability and linear programming problems*, Ph. D. dissertation, Dept. of Computer Science, Stanford University, 1980.

[3] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman and Company, San Francisco, 1979.

[4] A. ITAI, *Two-commodity flow*, J. Assoc. Comput. Mach., 25 (1978), pp. 596–611.

[5] D. B. JUDIN AND A. S. NEMIROVSKII, *Informational complexity and effective methods for the solution of convex extremal problems*, Ekonomika i Matematicheskie Metody, 12 (1976), pp. 357–369; [English translation in Matekon: Translations of Russian and East European Matematical Economics, 13:3 (1977), 25–45].

[6] L. G. KHACHIYAN, *A polynomial algorithm in linear programming*, Doklady Akademiîa Nauk SSSR Novaia Seriîa, 244 (1979), pp. 1093–1096 [English translation in Soviet Mathematics Doklady, 20 (1979), pp. 191–194].

[7] C. G. NELSON, *An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem*, Technical Report AIM-319, Dept. of Computer Science, Stanford University, 1978.

[8] C. G. NELSON AND D. C. OPPEN, *Simplification by cooperating decision procedures*, ACM Transactions on Programming Systems and Languages, 1 (1979), pp. 245–257.

[9] D. C. OPPEN, *Convexity, Complexity, and Combinations of Theories*, Theoret. Comput. Sci., to appear.

[10] V. R. PRATT, *Two easy theories whose combination is hard*, unpublished manuscript (1977).

[11] S. P. REISS AND D. P. DOBKIN, *The complexity of linear programming*, Theoret. Comput. Sci., 11 (1980), pp. 1–18.

[12] N. Z. SHOR, *Convergence rate of the gradient descent method with dilatation of the space*, Kibernetika, No. 2 (1970), pp. 80–85 [English translation in Cybernetics, 6 (1970), pp. 102–108].

[13] N. Z. SHOR, *Cut-off method with space extension in convex programming problems*, Kibernetika, No. 1 (1977), pp. 94–95 [English translation in Cybernetics, 13 (1977), pp. 94–96].

[14] R. SHOSTAK, *Deciding linear inequalities by computing loop residues*, Proc. Fourth Workshop on Automatic Deduction, Austin, Texas, 1979, pp. 81–89; J. Assoc. Comput. Mach., to appear.

SIAM J. COMPUT.
Vol. 9, No. 4, November 1980

0097-5397/80/0902-0015$01.00/0

ORTHOGONAL PACKINGS IN TWO DIMENSIONS*

BRENDA S. BAKER†, E. G. COFFMAN, JR.† AND RONALD L. RIVEST‡

Abstract. We consider problems of packing an arbitrary collection of rectangular pieces into an open-ended, rectangular bin so as to minimize the height achieved by any piece. This problem has numerous applications in operations research and studies of computer operation. We devise efficient approximation algorithms, study their limitations, and derive worst-case bounds on the performance of the packings they produce.

Key words. two-dimensional packing, bin packing, resource constrained scheduling

1. Introduction. Efficiently packing sets of rectangular figures into a given rectangular area is a problem with widespread application in operations research. Thus, one is inclined to attribute the scarcity of results on this problem, and others of its type, to inherent difficulty rather than to lack of importance. Motivated by the intractability of these problems, we define and analyze certain approximation algorithms. These algorithms are natural in the sense that they would probably be among the first to occur to anyone wishing to design simple, fast procedures for determining easily computed packings. The analysis of these algorithms leads to bounds on the performance of approximate packings relative to the best achievable.

In the remainder of this section we define the model to be studied and introduce notation. At that point we examine in more detail the applications which are served by the model, and we review the literature bearing on this and similar models. In § 2 the main results of the paper are presented and proved. Concluding remarks and a discussion of open problems are given in § 3.

As illustrated in Fig. 1, we consider an "open-ended" rectangle, R, of width w and a collection of rectangles, also called pieces, organized into a list $L=(p_1,p_2,\cdots,p_n)$. Each piece is defined by an ordered pair $p_i=(x_i,y_i), 1\leqq i\leqq n$, corresponding to the horizontal (x_i) and vertical (y_i) dimensions of the rectangle.

We are concerned with the packing or assignment of the pieces in L into R so as to minimize the *height*, h, of the packing; i.e., the maximum height, measured from the bottom edge of R, of the space occupied by any piece in the packing (see Fig. 1). In addition to the implicit requirement that the spaces occupied by distinct pieces be disjoint, we restrict attention to packings that are *orthogonal* and *oriented*. An orthogonal packing is one in which every edge of every rectangle is parallel to either the bottom edge or the vertical edges of R. An orthogonal packing is also oriented if the rectangles are regarded strictly as ordered pairs; i.e., a rectangle (x_i,y_i) must be packed in such a way that the edges of length x_i are parallel to the bottom edge of R. Thus, rotations of 90° (which preserve orthogonality) are not allowed.

Returning to applications we see that our model applies to industrial or commercial situations in which objects are to be packed on floors, shelves, truck beds, etc., where concern is limited to the objects in two prespecified dimensions. Another important application concerns systems containing a shared resource. A prime exam-

*Received by the editors July 3, 1978, and in final form March 28, 1980.

†Bell Laboratories, Murray Hill, New Jersey 07974.
‡Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139.

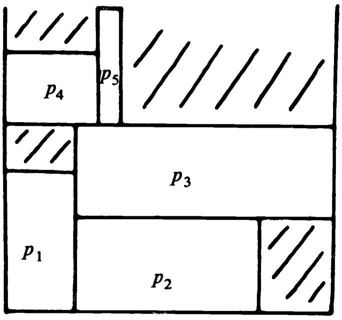
ple is the main memory resource in multiprogrammed computer systems. In such systems a number of tasks compete for a resource which they can share, but only within the limit provided by the total amount of resource available.

This application of the model was defined almost 20 years ago by E. F. Codd [1] in a study of multiprogramming systems. More recently, Garey and Graham [2] considered a related problem oriented to multiprocessor systems. In their study arbitrary numbers of processors and additional resources were considered. The analysis focused on worst-case bounds on the ratios of schedule-lengths (packing heights) for arbitrary lists; approximation algorithms were not considered. Moreover, the model of resources is basically different: Whenever an amount of the resource is available, no matter how it is configured, it can be used to satisfy any demand no greater than this amount; i.e., fragmentation of the resource is not a consideration.

Little else appears to have been published which bears on the packing problem we have defined. Erdös and Graham [3] have shown that orthogonal packings of squares into rectangles are not always optimum; i.e., there exist examples for which all orthogonal packings have greater height than the minimum achievable by exploiting the ability to rotate the squares. Based on earlier work of Meir and Moser [6], Kleitman and Krieger have considered the problem of finding a smallest rectangle into which a collection of squares can be packed [4], [5]. Specifically, they prove that a $\sqrt{2} \times 2/\sqrt{3}$ rectangle is always sufficient to pack a set of squares whose cumulative area is unity, and that no rectangle of smaller area can have this property.

Even when $x_i = x_j$ for all i and j, our packing problem is intractable; it can be shown that it becomes the NP-complete make-span minimization problem [7]. Hence, we are moved to consider fast heuristics and how closely the packings they produce approach optimum packings. For this purpose we define the following class of packing algorithms, to be called *bottom-up left-justified* (or simply BL) algorithms. (Recall that pieces must be packed so as to preserve oriented, orthogonal packings.) Each such algorithm packs the pieces one at a time as they are drawn in sequence from the list L. When a piece is packed into R it is first placed into the lowest possible location, and then it is left-justified at this vertical position in R. In the sequel R will also be referred to as a bin.

Fig. 1 shows a BL packing. Note that from a combinatorial point of view our problem remains essentially unchanged if we replace left-justification by right-justification and consider BR packings instead. Note also that two BL algorithms differ only in the ordering of L.



$L = (p_1, p_2, p_3, p_4, p_5)$

x_i: 3 8 11 4 1

y_i: 6 4 4 3 5

FIG 1. *Two-dimensional packing.*

2. Performance bounds for BL packings. We shall see that the basic BL algorithm, using a poorly ordered list L, can perform arbitrarily badly relative to an optimization algorithm. Thus, it is natural to inquire about the improvement possible by ordering L on the basis of some simple measure of piece size. Some obvious orderings to consider are increasing height, decreasing height, increasing width, and decreasing width. With the proper ordering the improvement can indeed be striking, as we shall see. However, with a badly chosen ordering, we can be just as poorly off as before. In particular, this will be true if we order pieces by increasing width (or decreasing height).

As a matter or notation let h_{BL} and h_{OPT} denote the respective heights of a BL and optimum packing of a list L which will always be clear by context.

THEOREM 1. *For any $M>0$, there exists a list of pieces ordered by increasing width such that $h_{\mathrm{BL}}/h_{\mathrm{OPT}}>M$.*

Proof. We shall define a class of lists which proves this result. First, let $k\geqq 2$ be given and define $r_i=\max\{m|i\equiv 0 \bmod k^m\}$, $i>0$. Thus, $r_i=1$ if i is a multiple of 4 but not 16, $r_i=2$ if i is a multiple of 16 but not 64, $r_i=3$ if i is a multiple of 64 but not 256, etc.; $r_i=0$ if i is not a multiple of 4.

Let the bin width be $w=k^k$ and let $s=k^{k-1}$. Rectangles are packed in the order given. Along the bottom row we pack k^k unit-width pieces, the ith of which has height $1-r_i\varepsilon$, where ε is much smaller than 1. The remaining rectangles all have unit height but widths in the order given by

$$\begin{array}{ll} s \text{ of width } 1 & \text{(the second row)} \\ s/k \text{ of width } k & \text{(the third row)} \\ s/k^2 \text{ of width } k^2 & \text{(the fourth row)} \\ \vdots & \\ s/k^{k-1}=1 \text{ of width } k^{k-1} & \text{(the } k+1\text{)st row).} \end{array}$$

Since $r_i=0$ for i not a multiple of k, one obtains the "notching" structure illustrated in the first row of Fig. 2; i.e., every kth piece is lower than the intervening $k-1$ pieces of unit height. Thus, the $s=k^{k-1}$ unit squares of the second row are placed on top of every kth piece of the first row. Now every kth piece of the second row corresponds to every (k^2)th piece of the first row and reaches a height at most $2-2\varepsilon$ which is less than that of the intervening $k-1$ pieces of the second row, all at height $2-\varepsilon$. Thus, the third row left-justifies pieces of width k over each of the lower pieces of the second row. Note that a width exceeding $k-1$ is necessary so that the piece width exceeds the width of spaces in the second row.

A similar pattern applies to the heights reached in the third row: The height reached by every kth piece is determined by that of every (k^3)th piece of the first row. Thus, in the third row every kth piece achieves a height at most $3-3\varepsilon$ while the remaining pieces all achieve a height of $3-2\varepsilon$. It follows that the fourth row left-justifies pieces of width k^2 over each of the lower pieces of the third row. In general, then, the spaces in the $(j+1)$st row $(j\geqq 1)$ all have width $k^j(k-1)$, the s/k^j pieces have width k^j, and every kth piece reaches a height less than the others. Thus, when the next s/k^{j+1} pieces of width k^{j+1} are added, they are placed on top of every

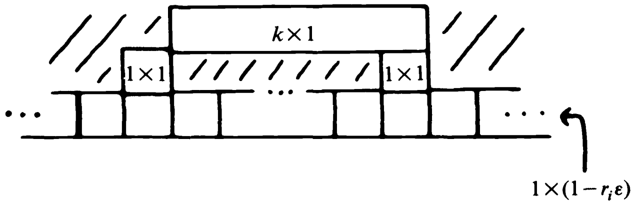


FIG 2. *The increasing widths example.*

kth piece in the $(j+1)$st row and each abuts the piece to the left in the $(j+1)$st row. Overall, the pattern looks like Fig. 2, drawn for $k=4$. (Distinctions of $O(\varepsilon)$ are not all represented, because of scaling.)

Since there are $k+1$ rows, we see that $h_{\text{BL}} = k+1-O(k\varepsilon)$. But a different packing can be found which packs rows 2 through $k+1$ into one row of height 1. (Note that the sum of widths of all pieces in rows 2 through $k+1$ is $k^{k-1} \times 1 + k^{k-2} \times k + \cdots + 1 \times k^{k-1} = k^k = w$.) Therefore, an optimum packing has a height not exceeding 2. Hence, we obtain a BL packing at least $k/2$ times higher than an optimum packing. Since k is arbitrary, the result follows. □

A dramatic improvement in the performance of BL packings is obtained when the list of rectangles is ordered by decreasing width. In fact, the ratio of BL to optimum packing height is guaranteed to be no worse than 3 when L is in decreasing order by width. For the case of squares, where decreasing width is equivalent to decreasing height, the bound is further reduced to 2. First, we shall show that the bounds of 3 and 2 can be approached as closely as desired; thus, these bounds are best possible.

THEOREM 2. *For any $\delta > 0$ there exists a list L of rectangles ordered by decreasing width such that the* BL *packing gives a height h_{BL} for which*

$$\frac{h_{\text{BL}}}{h_{\text{OPT}}} > 3 - \delta. \tag{1}$$

If the pieces are restricted to squares then an L can be found such that for any $\delta > 0$

$$\frac{h_{\text{BL}}}{h_{\text{OPT}}} > 2 - \delta. \tag{2}$$

Proof. We shall prove the second result first, since the first result is but a slight modification.

The list proving (2) corresponds to the "checkerboard" packing in Fig. 3. The pieces are all either unit squares or approximately 2×2 squares. In particular, the larger squares are disposed on the bottom of the bin with the dimensions stepping down by ε from piece to piece. Hence the assignment of pieces in the second row must be made from right to left according to the bottom-up rule. Since two unit squares exceed the dimension of any larger square and squares are left-justified, only one unit square is placed on each large square. Except for the first and last pieces, this type of assignment repeats on the second row since the "holes" in the second row all have width less than 1 and squares to the right are lower than squares to the left. In general,

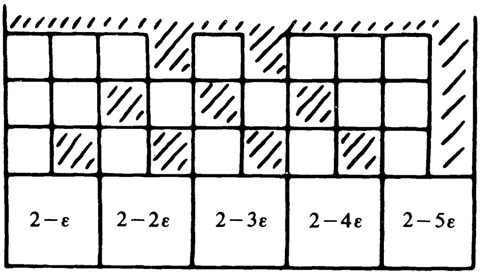


FIG 3. *The checkerboard example.*

the ith row of unit squares alternates holes and pieces except for at most the initial and final $i-1$ pieces of the row. Note that an optimum packing can be found which, except for possibly the last row, is within $O(\varepsilon)$ of being fully occupied.

The edge effects inhibiting the waste of half the space in the BL packing consist of

1. the row of larger pieces on the bottom, and
2. the triangular-shaped solidly packed collections of squares on the left and right of the packing.

Holding piece sizes constant, the influence of the first edge effect is reduced by increasing the height of the packing, while the second is attenuated by widening the bin. Let k be two greater than the number of rows of unit squares. If the width of the bin is selected to be k^2, then the area of the bottom row and side edge effects is $O(k^2)$. Thus, ignoring $O(\varepsilon)$ terms, we can find a list such that

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}}=\frac{k^3}{k^3/2+O(k^2)}.$$

In the limit $k\rightarrow\infty$, we have the bound of 2.

For the case of rectangles, it is only necessary to augment the list for Fig. 3 by adding as a new, last piece a rectangle of unit width and a height which equals the height of the optimum packing corresponding to the new list. Omitting the details, the BL packing will correspond to Fig. 3 with the new piece placed on top. It is easy to verify that the height ratio can now be made to approach 3 as closely as desired. □

THEOREM 3. *Let L be a list of rectangles ordered by decreasing widths. Then*

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}}\leqq 3. \tag{3}$$

This bound is best possible in the sense of Theorem 2.

Proof. Let h^* denote the height of the lower edge of a tallest piece whose upper edge is at height h_{BL}. If y denotes the height of this piece, then $h_{\mathrm{BL}}=y+h^*$. Let A denote the region of the bin up to height h^*.

Suppose we can show that A is at least half occupied. Then we have $h_{\mathrm{OPT}}\geqq\max\{y,h^*/2\}$; hence, $y>h^*/2$ implies

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}}\leqq\frac{y+h^*}{y}<\frac{y+2y}{y}=3,$$

and if $y \leqq h^*/2$, we have

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}} \leqq \frac{h^*/2+h^*}{h^*/2} = 3.$$

The result will thus be proved. It remains to show that A is at least half occupied.

Any horizontal cut or line through A can be partitioned into alternating segments corresponding to cuts through unoccupied and occupied areas of the BL packing. We shall show that the sum of the occupied segments is at least the sum of the unoccupied segments. For convenience we may restrict ourselves to lines which do not coincide with the (upper or lower) edges of any piece. Since the set of such lines is of measure zero, ignoring them will not influence our claim that A is at least half occupied.

Initially, consider the partition of a given line just prior to when the first rectangle, say q, is assigned with a lower edge at a height exceeding the height, h, of the line. The piece q need not be in A; its existence is guaranteed by the fact that there is a piece packed above A. We claim that at that point in the assignment sequence the line is "half occupied."

First, bottom-up packing implies that all lines must cut through at least one piece. Second, all lines must cut through a piece abutting the left bin edge. For suppose not; then the left-most piece, say q', cut by the line must abut another piece, say q'', to the left and entirely below the line. Thus, the length, x, of the unoccupied, initial segment of the line must be at least the width of q''. But since q'' was packed prior to q, the width of q must be less than that of q'' and hence less than x. Since at the point in time we are considering, no piece has been assigned entirely above the line, the space vertically above the initial segment must be completely unoccupied. Thus, we have the contradiction that q would have fit into the space above q'' in such a way that its lower edge is at a height less than h.

Now consider any segment, S, of the line which cuts through an unoccupied space. Let p be the piece bordering S on the left. Since when q is assigned, it is placed above the line, q must be wider than the length of S. (Once again, at the time q is assigned its height could not prevent its placement in a sufficiently wide unoccupied space cut by the line.) But q is packed later than p; consequently, p is at least as wide as q. It follows that for each segment representing unoccupied space along the line there is a longer segment representing occupied space immediately to its left. Clearly, for any given line, the sum of the segment lengths corresponding to unoccupied space must be monotonically nonincreasing as the packing sequence progresses. Therefore, the line continues to be at least half occupied. Finally, "integration" over the height of A verifies that A is at least half full. □

COROLLARY 1. *If L in the statement of Theorem 3 consists only of squares, then*

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}} \leqq 2.$$

This bound is best possible in the sense of Theorem 2.

Proof. First, define A' as the area extending from height y to height $h_{\mathrm{BL}}-y$, where y is the size of the tallest rectangle (now square) assigned above A in Theorem 2. Let p denote this square. As in Theorem 2, $A' \subset A$ is shown to be at least half occupied. But now, if w denotes the width of the bin, we observe that the cumulative occupied area of the upper and lower $w \times y$ slabs of the packing is at least wy and hence they are (when considered together) half occupied. This follows from the facts that p is at most as large as any square on the bottom of the bin, the bottom of the bin is full except possibly for a space at the right end, and the area of p must exceed y times the width of this space. Hence, the entire packing is at least half occupied from 0 to h_{BL}. It follows immediately that $h_{\mathrm{BL}} \leqq 2h_{\mathrm{OPT}}$. □

We have seen that a BL algorithm can yield reasonably good packings when the list of pieces is sorted into decreasing order by width. That is, Theorem 3 shows that the bin height used is no more than three times the optimal bin height, and if the pieces are squares, then the packing can be no more than twice as high.

A natural question to ask next is, "For every set of pieces, is there *some* ordering of those pieces into a list such that the BL rule, when applied to that list, yields an optimal packing?" Our checkerboard example, which showed that a list of squares sorted into decreasing order by size can use up to twice as much space as an optimal packing, can be packed optimally by the BL algorithm if the list is sorted into *increasing* order by size. While it might be difficult in practice to actually determine an ordering for which the BL rule produces an optimum packing, it would be comforting to know that one was not excluding the possibility of finding an optimum packing by considering only bottom-up packings.

Unfortunately, there are sets of pieces for which no BL packing is optimal. That is, no matter what ordering is used, the BL algorithm will produce a suboptimal packing. In fact we shall present an example using squares only, which demonstrates that an optimal packing can be as little as $\frac{11}{12}$ the height of the best bottom-up packing.

THEOREM 4. *There exist sets of squares such that the ratio of the bin height used by the best bottom-up packing to that of an optimum packing is at least* $12/(11+\varepsilon)$ *for any sufficiently small* $\varepsilon > 0$.

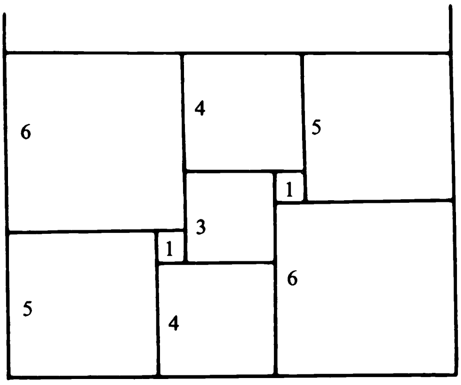
Proof. Consider the set of squares of sizes (6,6,5,5,4,4,3,1,1) and a rectangle of width 15. An optimum packing, of height 11, is shown in Fig. 4. We first demonstrate that (up to obvious left-right symmetries) this is the only optimum packing, and then we modify the example slightly to obtain the theorem.

Since Fig. 4 is a tight packing, any optimum packing must have height 11. For an arbitrary optimum packing consider the 15×11 rectangle A that it packs to be divided into 15 disjoint 1×11 vertical slabs. Let the type of a slab be an ordered list of the sizes of the squares that the slab intersects. The only possible types are:

(a) 6-5
(b) 6-4-1
(c) 6-3-1-1
(d) 5-5-1
(e) 5-4-1-1
(f) 4-4-3

Let a denote the number of slabs of type (a), etc. The following equations must then hold, for a,b,c,d,e,f nonnegative integers:

$$\begin{aligned}
a+b+c+d+e+f&=15\\
a+b+c&=12\\
a+2d+e&=10\\
b+e+2f&=8\\
c+f&=3\\
b+2c+d+2e&=2
\end{aligned}$$



$w = 15$

FIG 4. *Optimum vs. best* BL *packings.*

The first equation reflects the fact that there is a total of 15 slabs; the remaining five equations account for the presence of the squares of sizes 6,5,4,3,1 respectively. For example, the second equation states that the presence of exactly two 6×6 squares requires 12 slabs of types with 6's in them. By the last equation we have that either $c=0$ or $c=1$. Choosing $c=1$ yields a contradiction, and $c=0$ gives us the unique solution to the above equations:

$$\begin{aligned} &a = 10 \\ &b = 2 \\ &c = d = e = 0 \\ &f = 3. \end{aligned}$$

Any solution having the slab types in the above numbers must look like either Fig. 4 or its reflection. This can be proved by observing that (since $a=10$), each 5×5 is entirely above or below a 6×6. Take a particular 5×5 and consider the slab which intersects the adjacent 6×6 but does not contain the 5×5. This must be of type 6-4-1 or 6-5 (using the other five). The 6-5 possibility can't happen (since there would be a 6×3 unfilled space next to the other 6×6), and the pieces of the 6-4-1 must occur in the order 6,1,4 to prevent an unfillable gap between the 4×4 and the edge of A. The 4-4-3 slabs must come next: the pieces must be in the order 4,3,4 to leave room for the 6×6. Finally, the 6-4-1 and 6-5 slabs complete the picture. Thus, Fig. 4 represents the only way to pack the given squares into a 15×11 rectangle.

We now modify things so that (i) the 3×3 now has size $(3+\varepsilon) \times (3+\varepsilon)$ and (ii) the bin now has width $15+\varepsilon$. The preceding proof shows that (to within ε) the packing in Fig. 4 is still optimum. However, we note that in the modified packing there must be gaps on the bottom row between the 5×5 and 4×4 and between the 4×4 and 6×6 which add up to ε; otherwise the 1×1 and $(3+\varepsilon) \times (3+\varepsilon)$ will not be able to fit on top of the 4×4. Since no BL rule can produce those gaps, the optimum packing of Fig. 4 is unachievable. The best that such an algorithm can do is a packing of height 12. Thus we have

$$\frac{h_{\text{BL}}}{h_{\text{OPT}}} \geqq \frac{12}{11+\varepsilon}$$

for this example. □

As a final technical result we make an observation also made by R. E. Tarjan; viz., that the example of Theorem 1 can be modified to show that ordering the list by decreasing height can also lead to packings that are arbitrarily bad relative to the optimum.

THEOREM 1′. *For any $M>0$, there exists a list of pieces ordered by decreasing height such that*

$$\frac{h_{\mathrm{BL}}}{h_{\mathrm{OPT}}}>M.$$

Proof. Let $k>1$ be given and let the bin width be $w=k^k$. We shall define the BL packings proving the theorem by specifying the pieces row by row. The first two rows will each consist of w unit-width pieces. We shall define sequences $\{\delta_i\}$ and $\{\delta_i'\}$ such that the heights of pieces in the first row are given by

$$a_i=1+\delta_i, \qquad 1\leqq i\leqq w,$$

and the heights of pieces in the second row are, indexed from left to right,

$$b_i=1+\delta_i', \qquad 1\leqq i\leqq w.$$

The δ_i and δ_i' sequences are defined below so that $a_1<a_t+b_t$ and

$$a_1\geqq a_2\geqq \cdots \geqq a_t\geqq b_t\geqq b_{t-1}\geqq \cdots \geqq b_1\geqq 0. \tag{4}$$

Thus, the pieces will be packed in the order given by (4), the piece of height b_i will be on top of the piece of height a_i, $1\leqq i\leqq w$, and the cumulative heights achieved in the second row will be $h_i=2+\delta_i+\delta_i'$, $1\leqq i\leqq w$.

As in the proof of Theorem 1, define $r_i=\max\{m\,|\,k^m \text{ divides } i\}$. Thus, if i is a multiple of k but not k^2, then $r_i=1$; if i is a multiple of k^2 but not k^3, then $r_i=2$; etc. Clearly, $r_i=0$ if i is not a multiple of k. Note that $\max_{1\leqq i\leqq w}\{r_i\}=r_w=k$. Next, define

$$\delta_i=1-i/2w, \qquad i\leqq i\leqq w,$$

$$\delta_i'=i/2w-r_i\varepsilon, \qquad 1\leqq i\leqq w,$$

where $0<\varepsilon<1/2wk$, and hence $\delta_i'\geqq 0$, $1\leqq i\leqq w$. Note that $\delta_i>\delta_{i+1}$, $1\leqq i<w$, and $\delta_w=\frac{1}{2}\geqq\delta_w'=\frac{1}{2}-r_w\varepsilon$. Since $\delta_{i+1}'-\delta_i'=1/2w-(r_{i+1}-r_i)\varepsilon\geqq 1/2w-k\varepsilon>0$, the ordering in (4) follows. Moreover, the cumulative heights in the second row are $h_i=3-r_i\varepsilon$. Note that these heights have the same notching effect as the heights of the first row of pieces in Theorem 1.

Let $s=w/k=k^{k-1}$. As in Theorem 1 the remaining rectangles will be of height 1, with widths in the order given by:

$$\begin{array}{ll} s & \text{of width } 1, \\ \dfrac{s}{k} & \text{of width } k, \\ \dfrac{s}{k^2} & \text{of width } k^2, \\ & \vdots \\ \dfrac{s}{k^{k-1}}=1 & \text{of width } k^{k-1}. \end{array}$$

The pieces pack in a pattern similar to that of Theorem 1, using a total height of $k+3-O(k\varepsilon)$.

Note that a different packing could pack all the pieces in rows 3 and above into one row of height 1, and the remaining pieces into two rows of height 3 as in the above packing. Thus an optimum packing has a height no greater than 4. Therefore, the bottom-up packing is at least $k/4$ times higher than an optimum packing. Since k is arbitrary, the result follows. □

3. Conclusions. This paper is but a beginning in the study of fast, effective approximation algorithms for packing pieces in two dimensions. We have seen that, although performance of these algorithms can be very poor, simple measures such as ordering on piece size can produce algorithms with much more reasonable performance. Indeed, with such algorithms it appears that worst-case performance can only be approached by essentially pathological cases.

Subsequent to the work [8] on which this paper is based, considerable activity has arisen in two-dimensional bin-packing. Performance bounds have been found for so-called level-oriented algorithms [9], [10], which in terms of worst-case performance are superior to the bottom-up algorithms. Also, examples have been found [11] which show that a best BL packing can be as much as 5/4 worse than the optimum packing.

Many open problems related to our models remain for future research. Questions that should be resolved are those connected with the specialization to squares and those arising from list orderings we have not considered. For example, what is a tight bound for the special case of increasing squares? Another question concerns the implementation of the BL algorithms. The complexity of such algorithms for decreasing widths is open. How does one efficiently maintain the structure of available space as the packing sequence progresses?

REFERENCES

[1] E. G. Codd, *Multiprogram scheduling*, Comm. of the ACM, Parts 1 and 2, 3 (1960), pp. 347–350, Parts 3 and 4, 3 (1960), pp. 413–418.

[2] M. R. Garey and R. L Graham, *Bounds on multiprocessing scheduling with resource constraints*, this Journal, 4 (1975), pp. 187–200. (See also Chap. 4 of [7]).

[3] P. Erdős and R. L. Graham, *On packing squares with equal squares*, Tech. Rep. STAN-CS-75-483, Computer Science Dept., Stanford University, March, 1975.

[4] D. J. Kleitman and M. M. Krieger, *An optimal bound for two-dimensional bin-packing*, Proc. 16th Annual Symposium on Foundations of Computer Science, Los Angeles, Oct. 1975.

[5] D. Kleitman and M. Krieger, *Packing squares in rectangles; I*, Annals of the New York Academy of Sciences, Vol. 175, Article 1, pp. 253–262, July, 1970.

[6] A. Meir and L. Moser, *On the packing of squares and cubes*, Journal of Combinatorial Theory, 5 (1968), pp. 126–134.

[7] J. D., Ullman, *Complexity of sequencing problems*, Computer and Job-shop Scheduling Theory, E. G. Coffman, Jr. (ed), John Wiley and Sons, New York, 1975.

[8] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest, *Orthogonal packings in two dimensions*, Proc. Allerton Conf., 1978 University of Illinois, Champaign-Urbana.

[9] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson and R. E. Tarjan, *Performance bounds for level-oriented two-dimensional packing algorithms*, this Journal, this issue, pp. 808–826.

[10] Daniel D. K. D. B. Sleator, *A 2.5 times optimal algorithm for packing in two dimensions*, Inform. Process. Lett. 10 (1980), pp. 37–40.

[11] D. Brown, private communication.