JOURNAL OF COMPUTER AND SYSTEM SCIENCES 9, 355–365 (1974)

Testing Flow Graph Reducibility

R. ENDRE TARJAN*

Computer Science Division, University of California, Berkeley, California 94720

Received July 3, 1974

Many problems in program optimization have been solved by applying a technique called *interval analysis* to the flow graph of the program. A flow graph which is susceptible to this type of analysis is called *reducible*. This paper describes an algorithm for testing whether a flow graph is reducible. The algorithm uses depth-first search to reveal the structure of the flow graph and a good method for computing disjoint set unions to determine reducibility from the search information. When the algorithm is implemented on a random access computer, it requires $O(E \log^* E)$ time to analyze a graph with E edges, where $\log^* x = \min\{i \mid \log^{(i)} x \leqslant 1\}$. The time bound compares favorably with the $O(E \log E)$ bound of a previously known algorithm.

INTRODUCTION

Many code optimization methods model the flow of control in a computer program by a directed graph, called a *flow graph*. In order for some of these methods to work, the flow graph must have a special property called *reducibility*. Such methods include algorithms for finding dominators [1], finding common subexpressions [2, 3], finding active variables [4, 5], determining constant propagation [6], finding useless definitions [6], and solving other problems [7, 8]. Some interesting classes of computer programs, such as "go-to-less-programs," give rise to flow graphs which are necessarily reducible [9], and all programs may be modeled by a reducible flow graph using a process of "node splitting" [10]. However, this can be computationally expensive. We would like a fast algorithm for determining whether these optimization methods can be applied to any given program; that is, an algorithm for determining whether a flow graph is reducible.

A "reducible" flow graph is a flow graph to which a technique called "interval analysis" may be applied to determine the graph's structure. Cocke [2] and Allen [7] were the original formulators of this notion. Hecht and Ullman [9] simplified the definition of reducibility, giving two simple transformations which characterize the

* This research was partially supported by the National Science Foundation, Contract No. NSF-GJ-35604X, and by a Miller Research Fellowship.

class of reducible graphs. They also gave several structural characterizations of reducibility [11]. Hopcraft and Ullman have constructed an $O(E \log E)$ algorithm which tests a graph for reducibility according to Hecht and Ullman's definition [12], if the problem graph has E edges.

This paper gives an algorithm which is asymptotically faster than Hopcroft and Ullman's. The algorithm tests one of Hecht and Ullman's structural characterizations. It happens that the algorithm simultaneously tests the definition, which is more useful for applications. The method uses depth-first search [13, 14] to reveal the structure of the flow graph and a good set union algorithm [15, 16, 17] to test reducibility using the search information. The exact running time of the algorithm depends upon the exact running time of the set union algorithm, which is unknown. However, a good bound on this running time is known, and the reducibility algorithm requires $O(\min\{E \log^* E, V \log V + E\})$ time to test a graph with V vertices and E edges, where $\log^* x = \min\{i \mid \log^{(i)} x \leqslant 1\}$. If $E > V \log V$, the algorithm requires $O(E)$ time and is optimal to within a constant factor, since every edge must be examined to determine reducibility.

Basic Notions

To study a graph algorithm we need a model of computation and some terminology from graph theory. We will assume a random-access computer model with memory cells able to hold integers of size V if the problem graph has V vertices. All simple operations (arithmetic operations, comparisons, logical operations) require fixed times. We shall study worst-case resource requirements and shall ignore constant factors.

A directed graph $G = (\mathscr{V}, \mathscr{E})$ is an ordered pair consisting of a set of "vertices" $\mathscr{V}$ whose number is generally denoted by V and a set of "edges" $\mathscr{E}$ whose number is generally denoted by E. Each edge is an ordered pair (v, w) of distinct vertices; we say the edge (v, w) "leaves" v and "enters" w. If $G_1 = (\mathscr{V}_1, \mathscr{E}_1)$ is a graph and $\mathscr{V}_1 \subseteq \mathscr{V}$, $\mathscr{E}_1 \subseteq \mathscr{E}$, then G_1 is a "subgraph" of G. If $\mathscr{V}_2 = \mathscr{V}$ and $G_2 = (\mathscr{V}_2, \mathscr{E} \cap \mathscr{V}_2 \times \mathscr{V}_2)$, G_2 *is the* "subgraph of G induced by the vertices $\mathscr{V}_2$."

A sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ in G is a "path" from v_1 to v_n. This path "contains" vertices $v_1, \ldots, v_n$ and "avoids" all other vertices. There is a path of no edges from every vertex to itself. Vertex w is "reachable" from vertex v if there is a path from v to w. A graph is "strongly connected" if every vertex is reachable from every other. A "flow graph" (G, s) is a graph with a distinguished vertex s such that every vertex is reachable from s. Vertex v "dominates" vertex w in flow graph (G, s) if $v \neq w$ and every path from s to w contains v.

A "(directed, rooted) tree" (T, r) is a flow graph such that r has no entering edges and every other vertex has exactly one entering edge. Vertex r is called a "root" of T. We write $v \rightarrow w$ if there is an edge (v, w) in T; in this case v is the "father" of w and w is a "son" of v. We write $v \overset{*}{\rightarrow} w$ if there is a path from v to w in T; v is an "ancestor" of w and w is a "descendant" of v. (Every vertex is an ancestor and a descendant of

itself.) If T_1 is a tree and T_1 is a subgraph of T, T_1 is called a "subtree" of T. If T is a subgraph of a directed graph E and T contains all the vertices of G then T is a "spanning tree" of G.

If T is a tree rooted at r, a preorder numbering [18] of the vertices of T is any numbering which can be generated by the following algorithm.

```
begin
  procedure PREORDER(v); begin
    number v greater than any previously numbered vertex;
    comment if v = r it may be numbered arbitrarily;
    for w such that v → w do PREORDER(w);
  end;
  PREORDER(r);
end;
```

LEMMA 1. *Let $ND(v)$ be the number of descendants of each vertex v in a tree T. If T has V vertices and they are numbered from 1 to V in preorder, then $v \overset{*}{\to} w$ iff $v \leqslant w < v + ND(V)$.*

Proof. See [14].

Let (G, s) be a flow graph, and let T be a spanning tree of G rooted at s which has a preorder numbering. T is a "depth-first spanning tree" (DFST) if the edges in G, but not in T, can be partitioned into three sets:

(1) A set of edges (v, w) with $w \overset{*}{\to} v$ in T, called *cycle arcs*;

(2) A set of edges (v, w) with $v \overset{*}{\to} w$ in T, called *forward arcs*;

(3) A set of edges (v, w) with neither $v \overset{*}{\to} w$ nor $w \overset{*}{\to} v$, and $w < v$, called *cross arcs*.

A DFST is so named because it can be generated by starting at s and carrying out a depth-first search of G, numbering the vertices in increasing order as they are reached during the search. A properly implemented algorithm using depth-first search requires $O(V + E)$ time to generate a DFST, number the vertices in preorder, calculate $ND(v)$ for each vertex, and find the sets of cycle arcs, forward arcs, and cross arcs [13, 14]. Figures 1 and 2 show the application of depth-first search to a flow graph.

Flow Graphs and Reducibility

Let v and $w \neq s$ be two vertices in a flow graph (G, s). By "collapsing w into v," we mean forming a new graph G' from G by deleting vertex w and its incident edges, adding an edge (v, x) for each deleted edge (w, x) with $x \neq v$ and (v, x) not already an

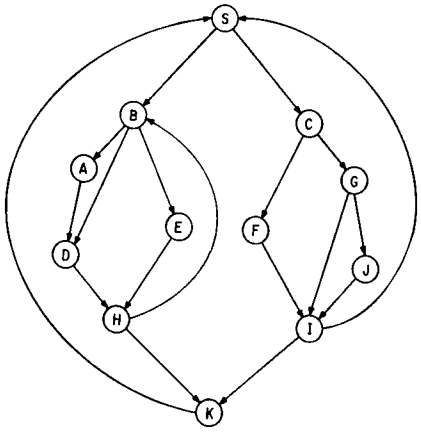

FIG. 1. A flow graph. Is this graph reducible?

edge, and adding an edge (x, v) for each deleted edge (x, w) with $x \neq v$ and (x, v) not already an edge. G' is obviously a flow graph.

If G' is formed from G by several collapsing operations, each vertex v' in G' corresponds to a set of vertices in G; namely, those collapsed into v'. If (v, w) is an edge of G such that v is collapsed into v' in G' and w is collapsed into w', then either $v' = w'$ or (v', w') is an edge of G'. In this case (v', w') in G' corresponds to (v, w) in G.

Consider the following transformation.

T_2: If (v, w) is the only edge entering w and $w \neq s$, collapse w into v.

A flow graph is "reducible" if and only if it can be transformed into the graph consisting only of vertex s by repeated application of T_2. This definition differs from Hecht and Ullman's definition in that they allow loops (edges of the form (v, v)) in their flow graphs and allow another transformation which deletes loops. It is easy to see that a flow graph with loops is reducible in Hecht and Ullman's sense if and only if after removing loops it is reducible in our sense.

If G' is obtained from G by repeated application of T_2, G' is a "reduction" of G. A unique graph results if T_2 is applied until it is no longer applicable [9]; thus, the order of transformations doesn't matter in a test for reducibility. To test the reducibility of (G, s), suppose we count the number of edges entering each vertex. Then we find a vertex with only one entering edge and apply T_2, collapsing the graph and updating the number of edges entering each other vertex. We repeat until we reduce the graph entirely or we get stuck. Each application of T_2 requires $O(V)$ time and reduces the number of vertices by one, so this obvious algorithm has an $O(V^2)$ time bound.

Hopcroft and Ullman have improved this algorithm to $O(E \log E)$ by applying a clever method of updating information after T_2 is applied [12]. Hopcroft and Ullman's

algorithm is only an improvement if E is small relative to V, but this is generally true in any flow graph representing a computer program.

Hecht and Ullman have given several structural characterizations of reducible flow graphs [11]. One of these gives a faster reducibility algorithm. Later we shall see that the algorithm can be used to calculate a sequence of T_2 applications which will reduce a reducible flow graph.

Let T be a DFST of a flow graph (G, s).

THEOREM 2 (Hecht and Ullman). *G is reducible iff w dominates v for each cycle arc (v, w) (relative to T).*

Proof. See [11].

For each vertex w in G, let $C(w) = \{v \mid (v, w)$ is a cycle arc$\}$ and let $P(w) = \{v \mid \exists z \in C(w)$ such that there is a path from v to z which avoids $w\}$. If there are no cycle arcs (v, w), both $C(w)$ and $P(w)$ are empty.

LEMMA 3. *G is reducible iff for all w and for all $v \in P(w)$, $w \overset{*}{\to} v$ in T.*

Proof. For any v and w, if v is not a descendant of w then there is a path in T from s to v which doesn't contain w. If there is a w and a $v \in P(w)$ such that v is not a descendant of w, this means that there is a path which avoids w from s to some vertex in $C(w)$, and G is not reducible by Theorem 2. Conversely, if G is not reducible, by Theorem 2 there is a cycle arc (v, w) and a path which avoids w from s to v. Then $s \in P(w)$ but s is not a descendant of w. Q.E.D.

Let w be the highest numbered vertex in G with an entering cycle arc. Suppose that for all $v \in P(w)$, $w \overset{*}{\to} v$ in T. Let G' be formed from G by collapsing all vertices of $P(w)$ into w.

LEMMA 4. *Every arc (v', w') in G' corresponds to an arc (v, w') of G with $v' \overset{*}{\to} v$ in T.*

Proof. Let (v, w') be an arc of G. If $w' \in P(w)$, then $v \in P(w) \cup \{w\}$. If $v \in P(w)$, then $w \overset{*}{\to} v$. Thus, either (v, w') corresponds to no arc of G', or (v, w') corresponds to an arc (v', w') of G' with $v' = v$ or $v' = w \overset{*}{\to} v$. Q.E.D.

Let T' be the subgraph of G' whose edges correspond to the edges of T.

LEMMA 5. *T', with numbering the same as that of T, is a* DFST *of G'. Cycle arcs of G' correspond to cycle arcs of G, forward arcs of G' correspond to forward arcs or cross arcs of G, and cross arcs of G' correspond to cross arcs of G.*

Proof. The subgraph of T induced by $P(w) \cup \{w\}$ is a tree. Each vertex of G', except s, obviously has exactly one edge of T' entering it, so T' is a tree. Furthermore, the numbering of T is obviously a preorder numbering of T'. Lemma 4 implies that

a cycle arc of G corresponds either to nothing or to a cycle arc of G', a forward arc of G corresponds either to nothing or to a forward arc of G', and a cross arc of G corresponds either to nothing or to a cross arc or a forward arc of G'. It follows that T' is a DFST and the rest of Lemma 5 is true. Q.E.D.

LEMMA 6. *For any vertex $x < w$, let $P'(x)$ and $C'(x)$ be defined in G' relative to T' as $P(x)$ and $C(x)$ were defined in G relative to T. Then $x \overset{*}{\to} y$ in T' for all $y \in P'(x)$ iff $x \overset{*}{\to} y$ in T for all $y \in P(x)$.*

Proof. Suppose there is a y in $P'(x)$ such that y is not a descendant of x in T'. Then y is not a descendant of x in T. Furthermore, there is an x-avoiding path p' in G' from y to some z' such that (z', x) is a cycle arc of G'. By Lemma 5, (z', x) corresponds to some cycle arc (z, x) in G. The subgraph of G induced by the vertices $P(w) \cup \{w\}$ is strongly connected. Thus, there is an x-avoiding path p from y to z in G consisting of arcs corresponding to arcs of p' and possibly some arcs between vertices in $P(w) \cup \{w\}$. Thus $y \in P(x)$.

Conversely, suppose there is a y in $P(x)$ such that y is not a descendant of x in T. If $y \notin P(w)$, then $y \in P'(x)$ and y is not a descendant of x in T'. If $y \in P(w)$, then w is not a descendant of x in T', and $w \in P'(x)$. Q.E.D.

Lemmas 3–6 lead to an efficient algorithm for testing the reducibility of a flow graph (G, s). Let T be a DFST of (G, s) and let $w_1 > w_2 > \cdots > w_n$ be the vertices of G with entering cycle arcs. We calculate $P(w_1)$. If there is some nondescendant of w_1 in $P(w_1)$ we stop; otherwise we collapse the vertices of $P(w_1)$ into w_1 to form a graph G' and calculate $P'(w_2)$ in G'. If $P'(w_2)$ contains a nondescendant of $P'(w_2)$ we stop; otherwise we form G'' by collapsing $P'(w_2)$ into w_2 and calculate $P''(w_3)$ in G''. We continue in this way until we know that G is not reducible or we run out of cycle arcs, in which case G is reducible.

The only tricky part of this algorithm is forming G', G'', and so on by collapsing vertices. We use a disjoint set union algorithm described in [15, 16, 17]. A set with name v will contain v and all vertices collapsed into v in the current graph. Initially there will be V sets, each with the name of the single vertex it contains. The function FIND(x) will return the name of the set containing x (that is, the vertex which corresponds to x in the current graph). The procedure UNION(A, B, C) will compute the union of sets A and B (destroying A and B), and give the new set the name C.

The reducibility algorithm appears below in algolic notation. It is important to remember that by Lemma 4 the first vertex of any edge never changes during the collapsing operations. In addition to testing reducibility, the algorithm calculates, for each vertex x, the first of $w_1, w_2, \ldots, w_n$ into which x is collapsed. This value is called HIGHPT(x) and is defined to be zero if x is never collapsed. HIGHPT(x) will be used to construct a sequence of transformations T_2 to reduce a reducible graph.

procedure REDUCE(G, s); *begin*

a: construct DFST of G using depth-first search, numbering vertices from 1 to V in preorder and calculating $ND(v)$ for each vertex v;

b: *for* $v = 1$ *until* V *do begin*

make lists of all cycle arcs, forward arcs, and cross arcs entering vertex v; construct a set named v containing v as its only element; HIGHPT(v) := 0; *end*;

c: *for* $w = V$ *step* -1 *until* 1 *do begin*

comment P will contain vertices in $P(w)$ as defined in the current graph. Q will contain vertices in P whose entering edges haven't been examined;

$P := \varnothing$;

d: *for* each cycle arc (v, w) entering w *do* add FIND(v) to P;

$Q := P$;

comment now we construct $P(w)$ by exploring backward from vertices in Q; *while* $Q \neq \varnothing$ *do begin*

select a vertex $x \in Q$ and delete it from Q;

e: *for* each forward arc, tree arc, or cross arc (y, x) entering x

do begin

comment all cycle arcs entering x have already been collapsed;

$y' :=$ FIND(y);

if $w > y'$ or $w + ND(w) \leqslant y'$ *then go to* N;

if $y' \notin P$ and $y' \neq w$ *then* add y' to P and to Q;

if HIGHPT(y') = 0 *then* HIGHPT(y') := w;

end;

end;

comment now $P = P(w)$ in the current graph;

for $x \in P$ *do* UNION(x, w, w);

comment this collapsing may produce duplicate edges which are not deleted, but this does not affect the algorithm;

end;

Y: *comment* if program arrives here G is reducible, take appropriate action; *go to* DONE;

N: *comment* if program arrives here G is not reducible, take appropriate action;

DONE: *end*;

It is easy to prove by induction on the number of vertices with entering cycle arcs that this procedure correctly tests the reducibility of G. This follows from Lemmas 3–6. Steps a and b require $O(V + E)$ time. Each cycle arc of G is examined exactly once in Step d. Once a vertex becomes an element of P it is collapsed into some other vertex and its entering vertices are never reexamined. Thus, each forward arc, tree arc, or cross arc is examined exactly once in Step e. Each vertex becomes an element of P at most once during all executions of loop c. It follows that loop c requires $O(V + E)$ time plus time for $O(V)$ UNION's and $O(E)$ FIND's. The total time of the algorithm is dominated by the time for the set operations, which is $O(\min\{V \log V + E, E \log^* E\})$ by the results in [14, 16, 17]. The algorithm obviously requires $O(V + E)$ storage space.

Reducing a Reducible Graph

The algorithm above is nonconstructive, but we can use HIGHPT(v) to construct a sequence of transformations T_2 which will reduce a reducible graph G. We can assign numbers, called SNUMBER's to the vertices of G so that tree arcs (v, w), satisfy SNUMBER(v) $<$ SNUMBER(w) and cross arcs (v, w) also satisfy SNUMBER(v) $<$ SNUMBER(w). This can be done during the depth-first search of G[12], and corresponds to traversing the spanning tree of G using depth-first search and proceeding to highest numbered vertices first. Suppose we apply the reducibility algorithm and with each vertex v we associate the pair (HIGHPT(v), SNUMBER(v)). When the algorithm is finished, we order the vertices so that a vertex labeled (x_1, y_1) appears before a vertex labeled (x_2, y_2) if and only if $x_1 > x_2$ or $x_1 = x_2$ and $y_1 < y_2$. This order of vertices is called reduction order. We can carry out this sorting in $O(V)$ time using a two-pass radix sort [19].

LEMMA 7. *If G is reducible, then we may collapse the vertices of G in reduction order using T_2.*

Proof. We prove Lemma 7 by induction on the number of vertices collapsed. Suppose all the vertices up to v in reduction order may be collapsed. This creates a graph G' which is a reduction of G. Consider vertex v. If v is not the start vertex, a single tree arc enters v in G. If G contains a cycle arc (u, w) with $v \overset{*}{\to} w$, all vertices x on the tree path from u to w will have been collapsed before v, since HIGHPT(x) $\geqslant w$, and HIGHPT(v) $< v \leqslant w$. If G contains a forward arc (u, w) with $v \overset{*}{\to} w$, then HIGHPT(w) $\leqslant u$ by Lemmas 3 and 6. Furthermore HIGHPT(x) $\geqslant$ HIGHPT(w) and SNUMBER(x) $\leqslant$ SNUMBER(w) for all vertices x on the tree path from u to w. It follows that all vertices on the tree path from u to w have been collapsed before w. Suppose G contains a cross arc (u, w) with $v \overset{*}{\to} w$. Let x be the highest numbered common ancestor of u and w. Then HIGHPT(w) $\leqslant x$ by Lemmas 3 and 6, and all vertices y on the tree paths from x to w and from x to u satisfy HIGHPT(y) $\geqslant$ HIGHPT(w) and SNUMBER(y) $\leqslant$ SNUMBER(w). Thus all vertices on these tree

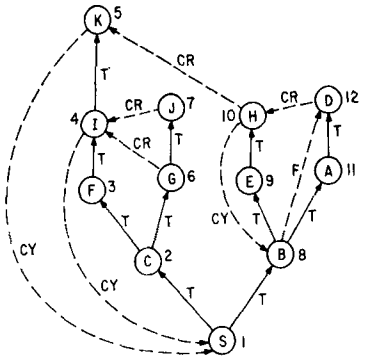

FIG. 2. Depth-first search of the graph in Fig. 1. Vertices are numbered in search order. Tree arcs are labelled *T*, cycle arcs *CY*, forward arcs *F*, and cross arcs *CR*.

paths have been collapsed before w. It follows that in G' vertex v can have only one edge entering it, and we may collapse v. The lemma holds in general by induction.

Q.E.D.

Figure 3 gives HIGHPT values, SNUMBER's, and a reduction order for the graph in Fig. 2.

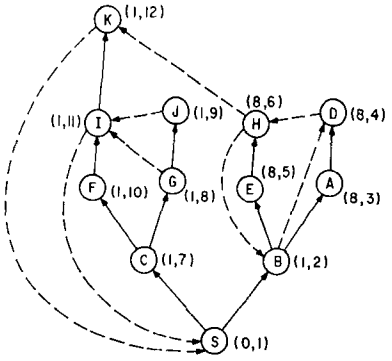

FIG. 3. HIGHPT and SNUMBER values (in parentheses) for the graph in Fig. 2. A reduction order is *A, D, E, H, B, C, G, J, F, I, K*.

CONCLUSIONS

This paper has presented an algorithm with an almost-linear time bound for determining whether a flow graph is reducible. The algorithm may be used to determine a way to reduce the graph if such a way exists. The method uses depth-first search and a good algorithm for computing disjoint set unions, and it improves upon a

previously published algorithm for determining reducibility. The algorithm may be used as a basic subroutine for various code optimization procedures [1–8]. Many of these procedures use nonlinear algorithms, some of which may be improvable using the methods applied here.

ACKNOWLEDGMENT

The author wishes to thank Professor Jeffrey Ullman for his helpful suggestions, which simplified the proof of the algorithm considerably.

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, On finding lowest common ancestors in trees, *in* "Proc. 5th Annual ACM Symposium on Theory of Computing," May 1972, pp. 253–263.
2. J. COCKE, Global common subexpression elimination, *SIGPLAN Notices* **5** (1970), 20–24.
3. J. D. ULLMAN, Fast algorithms for the elimination of common subexpressions, *Acta Informatica* **2** (1974), 191–213.
4. R. KENNEDY, A global flow analysis algorithm, *Internat. J. Comput. Math.* **3** (1971), 5–16.
5. M. SHAEFFER, "A Mathematical Theory of Global Program Optimization," Prentice–Hall, Englewood Cliffs, NJ, 1973.
6. A. V. AHO AND J. D. ULLMAN, "The theory of parsing, translation and compiling, Vol. II: Compiling," Prentice–Hall, Englewood Cliffs, NJ, 1973.
7. F. E. ALLEN, Control flow analysis, *SIGPLAN Notices* **5** (1970), 1–19.
8. F. E. ALLEN, Program optimization, *in* "Annual Review in Automatic Programming," Vol. 5, Pergammon Press, New York, 1969.
9. M. S. HECHT AND J. D. ULLMAN, Flow graph reducibility, *SIAM J. Comput.* **1** (1972), 188–202.
10. J. COCKE AND R. E. MILLER, Some analysis techniques for optimizing computer programs, *in* "Proc. 2nd International Conf. on System Sciences," Honolulu, Hawaii, 1969, pp. 143–146.
11. M. S. HECHT AND J. D. ULLMAN, "Characterizations of Reducible Flow Graphs," TR-118, Computer Science Laboratory Department of Electrical Engineering, Princeton University, NJ, 1973.
12. J. E. HOPCROFT AND J. D. ULLMAN, An $n \log n$ algorithm for detecting reducible graphs, *in* "Proc. 6th Annual Princeton Conf. on Inf. Sciences and Systems," Princeton, NJ, 1972, pp. 119–122.
13. R. E. TARJAN, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1** (1972), 146–160.
14. R. E. TARJAN, Finding dominators in directed graphs, *SIAM J. Comput.*, in press.
15. M. FISCHER, Efficiency of equivalence algorithms, *in* "Complexity of Computer Computations" (R. E. Miller and J. W. Thatcher, Eds.), pp. 153–168, Plenum Press, New York, 1972.
16. J. E. HOPCROFT AND J. D. ULLMAN, Set merging algorithms, *SIAM J. Comput.* **2** (1973), 294–304.

17. R. TARJAN, "On the efficiency of a good but not linear disjoint set union algorithm," TR 72-148, Department of Computer Science, Cornell University, Ithaca, NY, 1972.
18. D. KNUTH, "The Art of Computer Programming, Vol. 1: Fundamental Algorithms," pp. 315–347. Addison–Wesley, Reading, MA, 1968.
19. D. KNUTH, "The Art of Computer Programming, Vol. 3: Sorting and Searching," pp. 170–180, Addison–Wesley, Reading, MA, 1973.