A Fast Algorithm for Finding Dominators in a Flowgraph

THOMAS LENGAUER and ROBERT ENDRE TARJAN
Stanford University

A fast algorithm for finding dominators in a flowgraph is presented. The algorithm uses depth-first search and an efficient method of computing functions defined on paths in trees. A simple implementation of the algorithm runs in $O(m \log n)$ time, where m is the number of edges and n is the number of vertices in the problem graph. A more sophisticated implementation runs in $O(m\alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of Ackermann's function.

Both versions of the algorithm were implemented in Algol W, a Stanford University version of Algol, and tested on an IBM 370/168. The programs were compared with an implementation by Purdom and Moore of a straightforward $O(mn)$-time algorithm, and with a bit vector algorithm described by Aho and Ullman. The fast algorithm beat the straightforward algorithm and the bit vector algorithm on all but the smallest graphs tested.

Key Words and Phrases: depth-first search, dominators, global flow analysis, graph algorithm, path compression
CR Categories: 4.12, 4.34, 5.25, 5.32

1. INTRODUCTION

The following graph problem arises in the study of global flow analysis and program optimization [2, 6]. Let $G = (V, E, r)$ be a flowgraph[1] with start vertex r. A vertex v *dominates* another vertex $w \neq v$ in G if every path from r to w contains v. Vertex v is the *immediate dominator* of w, denoted $v = idom(w)$, if v dominates w and every other dominator of w dominates v.

THEOREM 1 [2, 6]. *Every vertex of a flowgraph $G = (V, E, r)$ except r has a unique immediate dominator. The edges $\{(idom(w), w) \mid w \in V - \{r\}\}$ form a directed tree rooted at r, called the* dominator tree *of G, such that v dominates w if and only if v is a proper ancestor of w in the dominator tree. See Figures* 1 *and* 2.

We wish to construct the dominator tree of an arbitrary flowgraph G. If G represents the flow of control of a computer program which we are trying to

[1] Appendix A contains the graph-theoretic terminology used in this paper.

The research of the first author was partly supported by the German Academic Exchange Service. The research of the second author was partly supported by the National Science Foundation under Grant MCS75-22870 and by the Office of Naval Research under Contract N00014-76-C-0688.
Authors' address: Computer Science Department, Stanford University, Stanford, CA 94305.

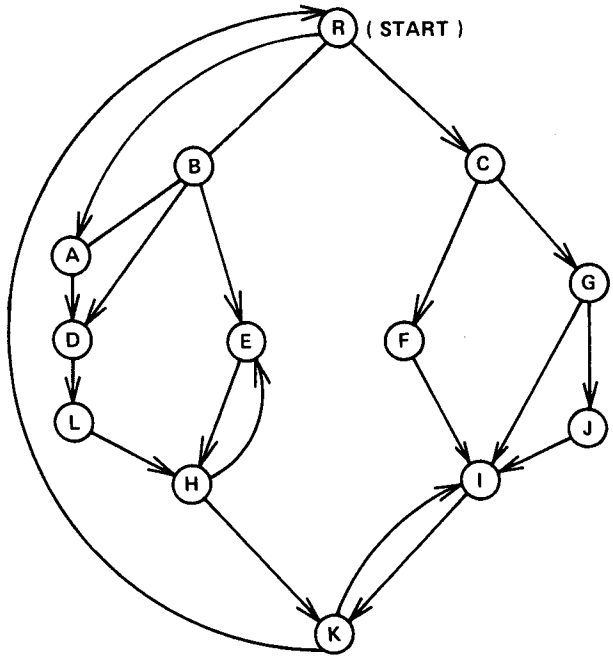

Fig. 1. A flowgraph

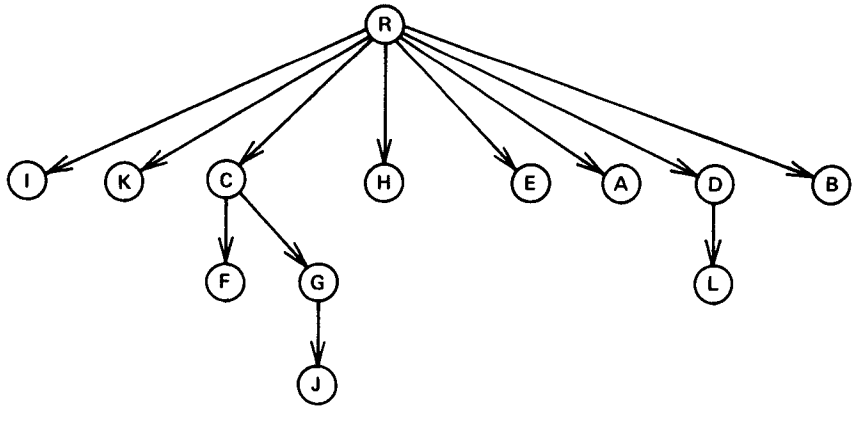

Fig. 2. Dominator tree of flowgraph in Fig. 1

optimize, then the dominator tree provides information about what kinds of code motion are safe. For further details see [2, 6].

Aho and Ullman [2] and Purdom and Moore [17] describe a straightforward algorithm for finding dominators. For each vertex $v \neq r$, we carry out the following step.

General Step. Determine, by means of a search from r, the set S of vertices reachable from r by paths which avoid v. The vertices in $V - \{v\} - S$ are exactly those which v dominates.

Knowing the set of vertices dominated by each vertex, it is an easy matter to construct the dominator tree.

To analyze the running time of this algorithm, let us assume that G has m edges and n vertices. Each execution of the general step requires $O(m)$ time, and the algorithm performs $n - 1$ executions of the general step; thus the algorithm requires $O(mn)$ time total.

Aho and Ullman [3] describe another simple algorithm for computing dominators. This algorithm manipulates bit vectors of length n. Each vertex v has a bit vector which encodes a superset of the dominators of v. The algorithm makes several passes over the graph, updating the bit vectors during each pass, until no further changes to the bit vectors occur. The bit vector for each vertex v then encodes the dominators of v.

This algorithm requires $O(m)$ bit vector operations per pass for $O(n)$ passes, or $O(nm)$ bit vector operations total. Since each bit vector operation requires $O(n)$ time, the running time of the algorithm is $O(n^2m)$. This bound is pessimistic, however; the constant factor associated with the bit vector operations is very small, and on typical graphs representing real programs the number of passes is small (on reducible flowgraphs [3] only two passes are required [4]).

In this paper we shall describe a faster algorithm for solving the dominators problem. The algorithm uses depth-first search [9] in combination with a data structure for evaluating functions defined on paths in trees [14]. We present a simple implementation of the algorithm which runs in $O(m \log n)$ time and a more sophisticated implementation which runs in $O(m\alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of Ackermann's function [1], defined as follows. For integers $i, j \geq 0$, let $A(i, 0) = 0$ if $i \geq 0$, $A(0, j) = 2^j$ if $j \geq 1$, $A(i, 1) = A(i - 1, 2)$ if $i \geq 1$, and $A(i, j) = A(i - 1, A(i, j - 1))$ if $i \geq 1$, $j \geq 2$. Then $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor 2m/n \rfloor) > \log_2 n\}$.

The algorithm is a refinement of earlier versions appearing in [10–12]. Although proving its correctness and verifying its running time require rather complicated analysis, the algorithm is quite simple to program and is very fast in practice. We programmed both versions of the algorithm in Algol W, a Stanford University version of Algol, and tested the programs on an IBM 370/168. We compared the programs with a transcription into Algol W of the Purdom-Moore algorithm and with an implementation of the bit vector algorithm. On all but the smallest graphs tested our algorithm beat the other methods.

This paper consists of five sections. Section 2 describes the properties of depth-first search used by the algorithm and proves several theorems which imply the correctness of the algorithm. Some knowledge of depth-first search as described in [9] and [10, sec. 2] is useful for understanding this section. Section 3 develops the algorithm, using as primitives two procedures that manipulate trees. Section 4 discusses two implementations, simple and sophisticated, of these tree manipulation primitives. Some knowledge of [14, secs. 1, 2, and 5] is useful for understanding this section. Section 5 presents our experimental results and conclusions.

2. DEPTH-FIRST SEARCH AND DOMINATORS

The fast dominators algorithm consists of three parts. First, we perform a depth-first search on the input flowgraph $G = (V, E, r)$, starting from vertex r, and numbering the vertices of G from 1 to n in the order they are reached during the search. The search generates a spanning tree T rooted at r, with vertices

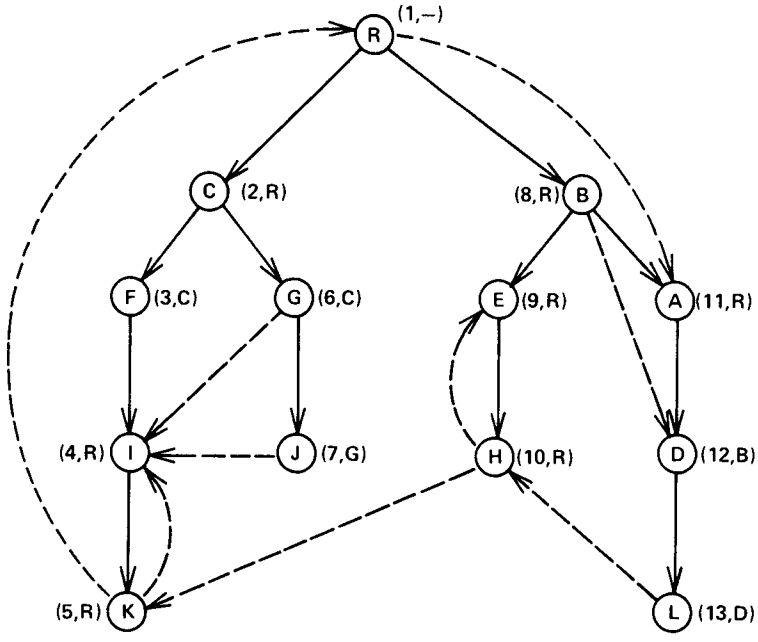

Fig. 3. Depth-first search of flowgraph in Fig. 1. Solid edges are spanning tree edges; dashed edges are nontree edges. Number in parentheses is vertex number; letter is semidominator

numbered in preorder [5]. See Figure 3. For convenience in stating our results, we shall assume in this section that all vertices are identified by number.

The following *paths lemma* is an important property of depth-first search and is crucial to the correctness of the dominators algorithm.

LEMMA 1 [9]. *If v and w are vertices of G such that $v \leq w$, then any path from v to w must contain a common ancestor of v and w in T.*

Second, we compute a value for each vertex $w \neq r$ called its *semidominator,* denoted by $sdom(w)$ and defined by

$$sdom(w) = \min\{v \mid \text{there is a path } v = v_0, v_1, \ldots, v_k = w \text{ such that } v_i > w \text{ for } 1 \leq i \leq k-1\}. \quad (1)$$

See Figure 3. Third, we use the semidominators to compute the immediate dominators of all vertices.

The semidominators have several properties which make their computation a convenient intermediate step in the dominators calculation. If $w \neq r$ is any vertex, then $sdom(w)$ is a proper ancestor of w in T, and $idom(w)$ is a (not necessarily proper) ancestor of $sdom(w)$. If we replace the set of nontree edges of G by the set of edges $\{(sdom(w), w) \mid w \in V \text{ and } w \neq r\}$, then the dominators of vertices in G are unchanged. Thus if we know the spanning tree and the semidominators, we can compute the dominators.

In the remainder of this section we prove the properties of semidominators and immediate dominators which justify the algorithm. The following three lemmas give basic relationships among the spanning tree, the semidominators, and the immediate dominators.

LEMMA 2. *For any vertex* $w \neq r$, $idom(w) \overset{+}{\rightarrow} w$.[2]

PROOF. Any dominator of w must be on the path in T from r to w. □

LEMMA 3. *For any vertex* $w \neq r$, $sdom(w) \overset{+}{\rightarrow} w$.

PROOF. Let *parent*(w) be the parent of w in T. Since (*parent*(w), w) is an edge of G, by (1) $sdom(w) \leq parent(w) < w$. Also by (1), there is a path $sdom(w) = v_0, v_1, \ldots, v_k = w$ such that $v_i > w$ for $1 \leq i \leq k - 1$. By Lemma 1, some vertex v_i on the path is a common ancestor of $sdom(w)$ and w. But such a common ancestor v_i must satisfy $v_i \leq sdom(w)$. This means $i = 0$, i.e. $v_i = sdom(w)$, and $sdom(w)$ is a proper ancestor of w. □

LEMMA 4. *For any vertex* $w \neq r$, $idom(w) \overset{+}{\rightarrow} sdom(w)$.

PROOF. By Lemmas 2 and 3, $idom(w)$ and $sdom(w)$ are proper ancestors of w. The path consisting of the tree path from r to $sdom(w)$ followed by a path $sdom(w) = v_0, v_1, \ldots, v_k = w$ such that $v_i > w$ for $1 \leq i \leq k - 1$ (which must exist by (1)) avoids all proper descendants of $sdom(w)$ which are also proper ancestors of w. It follows that $idom(w)$ is an ancestor of $sdom(w)$. □

LEMMA 5. *Let vertices* v, w *satisfy* $v \overset{+}{\rightarrow} w$. *Then* $v \overset{*}{\rightarrow} idom(w)$ *or* $idom(w) \overset{*}{\rightarrow} idom(v)$.

PROOF. Let x be any proper descendant of $idom(v)$ which is also a proper ancestor of v. By Theorem 1 and Corollary 1, there is a path from r to v which avoids x. By concatenating this path with the tree path from v to w, we obtain a path from r to w which avoids x. Thus $idom(w)$ must be either a descendant of v or an ancestor of $idom(v)$. □

Using Lemmas 1–5, we obtain two results which provide a way to compute immediate dominators from semidominators.

THEOREM 2. *Let* $w \neq r$. *Suppose every* u *for which* $sdom(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} w$ *satisfies* $sdom(u) \geq sdom(w)$. *Then* $idom(w) = sdom(w)$.

PROOF. By Lemma 4, it suffices to show that $sdom(w)$ dominates w. Consider any path p from r to w. Let x be the last vertex on this path such that $x < sdom(w)$. If there is no such x, then $sdom(w) = r$ dominates w. Otherwise, let y be the first vertex following x on the path and satisfying $sdom(w) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$. Let $q = (x = v_0, v_1, v_2, \ldots, v_k = y)$ be the part of p from x to y. We claim $v_i > y$ for $1 \leq i \leq k - 1$. Suppose to the contrary that some v_i satisfies $v_i < y$. By Lemma 1, some v_j with $i \leq j \leq k - 1$ is an ancestor of y. By the choice of x, $v_j \geq sdom(w)$, which means $sdom(w) \overset{*}{\rightarrow} v_j \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$, contradicting the choice of y. This proves the claim.

The claim together with the definition of semidominators implies that $sdom(y) \leq x < sdom(w)$. By the hypothesis of the theorem, y cannot be a proper descendant of $sdom(w)$. Thus $y = sdom(w)$ and $sdom(w)$ lies on the path p. Since the path selected was arbitrary, $sdom(w)$ dominates w. □

THEOREM 3. *Let* $w \neq r$ *and let* u *be a vertex for which* $sdom(u)$ *is minimum among vertices* u *satisfying* $sdom(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} w$. *Then* $sdom(u) \leq sdom(w)$ *and* $idom(u) = idom(w)$.

[2] Throughout this paper the notation "$x \overset{*}{\rightarrow} y$" means that x is an ancestor of y in the spanning tree T generated by the depth-first search, and "$x \overset{+}{\rightarrow} y$" means $x \overset{*}{\rightarrow} y$ and $x \neq y$.

PROOF. Let z be the vertex such that $sdom(w) \rightarrow z \overset{*}{\rightarrow} w$. *Then* $sdom(u) \leq sdom(z) \leq sdom(w)$.

By Lemma 4, $idom(w)$ is an ancestor of $sdom(w)$ and thus a proper ancestor of u. Thus by Lemma 5 $idom(w) \overset{*}{\rightarrow} idom(u)$. To prove $idom(u) = idom(w)$, it suffices to prove that $idom(u)$ dominates w.

Consider any path p from r to w. Let x be the last vertex on this path satisfying $x < idom(u)$. If there is no such x, then $idom(u) = r$ dominates w. Otherwise, let y be the first vertex following x on the path and satisfying $idom(u) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$. Let $q = (x = v_0, v_1, v_2, \ldots, v_k = y)$ be the part of p from x to y. As in the proof of Theorem 2, the choice of x and y implies that $v_i > y$ for $1 \leq i \leq k - 1$. Thus $sdom(y) \leq x$. Since $idom(u) \leq sdom(u)$ by Lemma 4, we have $sdom(y) \leq x < idom(u) \leq sdom(u)$.

Since u has the smallest semidominator among vertices on the tree path from z to w, y cannot be proper descendant of $sdom(w)$. Furthermore, y cannot be both a proper descendant of $idom(u)$ and an ancestor of u, for if this were the case the path consisting of the tree path from r to $sdom(y)$ followed by a path $sdom(y) = v_0, v_1, \ldots, v_k = y$ such that $v_i > y$ for $1 \leq i \leq k - 1$ followed by the tree path from y to u would avoid $idom(u)$; but no path from r to u avoids $idom(u)$.

Since $idom(u) \overset{*}{\rightarrow} v \overset{+}{\rightarrow} u \rightarrow w$ and $idom(u) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$, the only remaining possibility is that $idom(u) = y$. Thus $idom(u)$ lies on the path from r to w. Since the path selected was arbitrary, $idom(u)$ dominates w. □

COROLLARY 1. *Let* $w \neq r$ *and let* u *be a vertex for which* $sdom(u)$ *is minimum among vertices* u *satisfying* $sdom(w) \overset{+}{\rightarrow} u \rightarrow w$. *Then*

$$idom(w) = \begin{cases} sdom(w) & \text{if } sdom(w) = sdom(u), \\ idom(u) & \text{otherwise.} \end{cases} \tag{2}$$

PROOF. Immediate from Theorems 2 and 3. □

The following theorem provides a way to compute semidominators.

THEOREM 4. *For any vertex* $w \neq r$,

$$sdom(w) = min(\{v \mid (v, w) \in E \text{ and } v < w\} \cup \{sdom(u) \mid u > w \text{ and there is an edge } (v, w) \text{ such that } u \overset{*}{\rightarrow} v\}).$$

PROOF. Let x equal the right-hand side of (3). We shall first prove that $sdom(w) \leq x$. Suppose x is a vertex such that $(x, w) \in E$ and $x < w$. By (1), $sdom(w) \leq x$. Suppose on the other hand $x = sdom(u)$ for some vertex u such that $u > w$ and there is an edge (v, w) such that $u \overset{*}{\rightarrow} v$. By (1) there is a path $x = v_0, v_1, \ldots, v_j = u$ such that $v_i > u > w$ for $1 \leq i \leq j - 1$. The tree path $u = v_j \rightarrow v_{j+1} \rightarrow \cdots \rightarrow v_{k-1} = v$ satisfies $v_i \geq u > w$ for $j \leq i \leq k - 1$. Thus the path $x = v_0, v_1, \ldots, v_{k-1} = v, v_k = w$, satisfies $v_i > w$ for $1 \leq i \leq k - 1$. By (1), $sdom(w) \leq x$.

It remains for us to prove that $sdom(w) \geq x$. Let $sdom(w) = v_0, v_1, \ldots, v_k = w$ be a simple path such that $v_i > w$ for $1 \leq i \leq k - 1$. If $k = 1$, $(sdom(w), w) \in E$, and $sdom(w) < w$ by Lemma 3. Thus $sdom(w) \geq x$. Suppose on the other hand that $k > 1$. Let j be minimum such that $j \geq 1$ and $v_j \overset{*}{\rightarrow} v_{k-1}$. Such a j exists since $k - 1$ is a candidate for j.

We claim $v_i > v_j$ for $1 \leq i \leq j - 1$. Suppose to the contrary that $v_i \leq v_j$ for some i in the range $1 \leq i \leq j - 1$. Choose the i such that $1 \leq i \leq j - 1$ and v_i is

minimum. By Lemma 1, $v_i \dashrightarrow v_j$, which contradicts the choice of j. This proves the claim.

The claim implies $sdom(w) \geq sdom(v_j) \geq x$. Thus whether $k = 1$ or $k > 1$, we have $sdom(w) \geq x$, and the theorem is true. □

3. A FAST DOMINATORS ALGORITHM

In this section we develop an algorithm which uses the results in Section 2 to find dominators. Earlier versions of the algorithm appear in [10–12]; the version we present is refined to the point where it is as simple to program as the straightforward algorithm [2, 7] or the bit vector algorithm [3, 4], similar in speed on small graphs, and much faster on large graphs.

The algorithm consists of the following four steps.

Step 1. Carry out a depth-first search of the problem graph. Number the vertices from 1 to n as they are reached during the search. Initialize the variables used in succeeding steps.
Step 2. Compute the semidominators of all vertices by applying Theorem 4. Carry out the computation vertex by vertex in decreasing order by number.
Step 3. Implicitly define the immediate dominator of each vertex by applying Corollary 1.
Step 4. Explicitly define the immediate dominator of each vertex, carrying out the computation vertex by vertex in increasing order by number.

Our implementation of this algorithm uses the following arrays.

Input

succ(v): The set of vertices w such that (v, w) is an edge of the graph.

Computed

parent(w): The vertex which is the parent of vertex w in the spanning tree generated by the search.
pred(w): The set of vertices v such that (v, w) is an edge of the graph.
semi(w): A number defined as follows:
- (i) Before vertex w is numbered, *semi*(v) = 0.
- (ii) After w is numbered but before its semidominator is computed, *semi*(w) is the number of w.
- (iii) After the semidominator of w is computed, *semi*(w) is the number of the semidominator of w.

vertex(i): The vertex whose number is i.
bucket(w): A set of vertices whose semidominator is w.
dom(w): A vertex defined as follows:
- (i) After step 3, if the semidominator of w is its immediate dominator, then *dom*(w) is the immediate dominator of w. Otherwise *dom*(w) is a vertex v whose number is smaller than w and whose immediate dominator is also w's immediate dominator.
- (ii) After step 4, *dom*(w) is the immediate dominator of w.

Rather than converting vertex names to numbers during step 1 and converting numbers back to names at the end of the computation, we have chosen to refer to vertices as much as possible by name. Arrays *semi* and *vertex* incorporate all that we need to know about vertex numbers. Array *semi* serves a dual purpose, representing (though not simultaneously) both the number of a vertex and the number of its semidominator. As well as saving storage space, this device allows us to simplify the computation of semidominators by combining the two cases of Theorem 4 into one.

Here is an Algol-like version of step 1.

```
step1:  n := 0;
        for each v ∈ V do pred(v) := ∅; semi(v) := 0 od;
        DFS(r);
```

Step 1 uses the recursive procedure DFS, defined below, to carry out the depth-first search. When a vertex v receives a number i, the procedure assigns $semi(v) := i$ and $vertex(i) := v$.

```
procedure DFS(vertex);
  begin
    semi(v) := n := n + 1;
    vertex(n) := v;
    comment initialize variables for steps 2, 3, and 4;
    for each w ∈ succ(v) do
      if semi(w) = 0 then parent(w) := v; DFS(w) fi;
      add v to pred(w) od
  end DFS;
```

After carrying out step 1, the algorithm carries out steps 2 and 3 simultaneously, processing the vertices $w \neq r$ in decreasing order by number. During this computation the algorithm maintains an auxiliary data structure which represents a forest contained in the depth-first spanning tree. More precisely, the forest consists of vertex set V and edge set $\{(parent(w), w) \mid \text{vertex } w \text{ has been processed}\}$. The algorithm uses one procedure to construct the forest and another to extract information from it:

LINK(v, w): Add edge (v, w) to the forest.

EVAL(v): If v is the root of a tree in the forest, return v. Otherwise, let r be the root of the tree in the forest which contains v. Return any vertex $u \neq r$ of minimum $semi(u)$ on the path $r \overset{*}{\rightarrow} v$.

To process a vertex w, the algorithm computes the semidominator of w by applying Theorem 4. The algorithm assigns $semi(w) := \min\{semi(\text{EVAL}(v)) \mid (v, w) \in E\}$. After this assignment, $semi(w)$ is the number of the semidominator of w. To verify this claim, consider any edge $(v, w) \in E$. If v is numbered less than w, then v is unprocessed, which means v is the root of a tree in the forest and $semi(v)$ is the number of v. Thus $semi(\text{EVAL}(v))$ is the number of v. If v is numbered greater than w, then v has been processed and is not a root. Thus EVAL(v) returns a vertex u among vertices numbered greater than w satisfying $u \overset{*}{\rightarrow} v$ whose semidominator has the minimum number, and $semi(\text{EVAL}(v))$ is the number of u's semidominator. This means that the algorithm performs exactly the minimization specified in Theorem 4.

After computing $semi(w)$, the algorithm adds w to $bucket(vertex(semi(w)))$ and adds a new edge to the forest using LINK$(parent(w), w)$. This completes step 2 for w. The algorithm then empties $bucket(parent(w))$, carrying out step 3 for each vertex in the bucket. Let v be such a vertex. The algorithm implicitly computes the immediate dominator of v by applying Corollary 1. Let $u = \text{EVAL}(v)$. Then u is the vertex satisfying $parent(w) \overset{+}{\rightarrow} u \overset{*}{\rightarrow} v$ whose semidominator has minimum number. If $semi(u) = semi(v)$, then $parent(w)$ is the immediate dominator of v and the algorithm assigns $dom(v) := parent(w)$. Otherwise u and v have the same

dominator and the algorithm assigns $dom(v) = u$. This completes step 3 for v.

Here is an Algol-like version of steps 2 and 3 which uses LINK and EVAL.

```
        comment initialize variables;
        for i := n by −1 until 2 do
          w := vertex(i);
step2:  for each v ∈ pred(w) do
            u := EVAL(v); if semi(u) < semi(w) then semi(w) := semi(u) fi od;
          add w to bucket(vertex(semi(w)));
          LINK(parent(w), w);
step3:  for each v ∈ bucket(parent(w)) do
            delete v from bucket(parent(w));
            u := EVAL(v);
            dom(v) := if semi(u) < semi(v) then u
                      else parent(w) fi od od;
```

Step 4 examines vertices in increasing order by number, filling in the immediate dominators not explicitly computed by step 3. Here is an Algol-like version of step 4.

```
step4:  for i := 2 until n do
            w := vertex(i);
            if dom(w) ≠ vertex(semi(w)) then dom(w) := dom(dom(w)) fi od;
          dom(r) := 0;
```

This completes our presentation of the algorithm except for the implementation of LINK and EVAL. Figure 4 illustrates how the algorithm works.

Figure 4(a) is a snapshot of the graph just before vertex A is processed. Two edges (B, A) and (R, A) enter vertex A, giving 8 (the number of B) and 1 (the number of R) as candidates for *semi*(A). The algorithm assigns *semi*(A) := 1, places A in *bucket*(R), and adds edge (B, A) to the forest. Then the algorithm empties *bucket*(B), which contains only D. EVAL(D) produces A as the vertex on the path $B \overset{+}{\rightarrow} A \overset{*}{\rightarrow} D$ with minimum *semi*. Since *semi*(A) = 1 < 8 = *semi*(D), *idom*(A) = *idom*(D) and the algorithm assigns *dom*(D) = A.

Figure 4(b) is a snapshot of the graph just before vertex I is processed. Four edges (F, I), (G, I), (J, I), and (K, I) enter vertex I, giving 3 (the number of F), 2 (*semi*(G)), 2 (*semi*(G)), and 1 (*semi*(K)), respectively, as candidates for *semi*(I). The algorithm assigns *semi*(I) = 1, places I in *bucket*(R), and adds edge (F, I) to the forest. Then the algorithm empties *bucket*(F), which contains nothing.

Appendix B contains a complete Algol-like version of the algorithm, including variable declarations and initialization. Using Theorem 4 and Corollary 1, it is not hard to prove that after execution of the algorithm, $dom(v) = idom(v)$ for each vertex $v \neq r$, assuming that LINK and EVAL perform as claimed. The running time of the algorithm is $O(m + n)$ plus time for $n - 1$ LINK and $m + n - 1$ EVAL instructions.

4. IMPLEMENTATION OF LINK AND EVAL

Two ways to implement LINK and EVAL, one simple and one sophisticated, are provided in [14]. We shall not discuss the details of these methods here, but merely provide Algol-like implementations of LINK and EVAL which are adapted from [14].

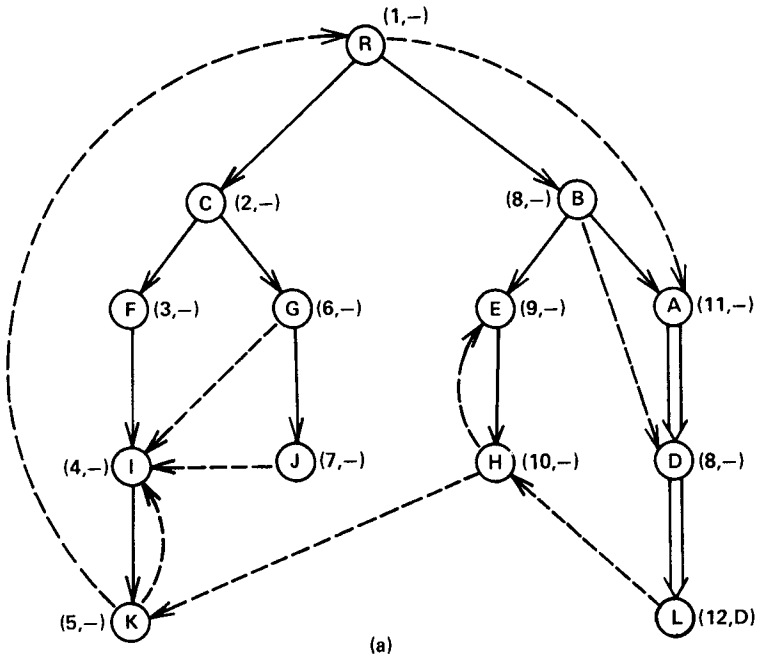

Fig. 4(a). Snapshot just before processing vertex A. Double lines denote edges in forest. Number in parentheses is *semi*; letter in parentheses is *dom*

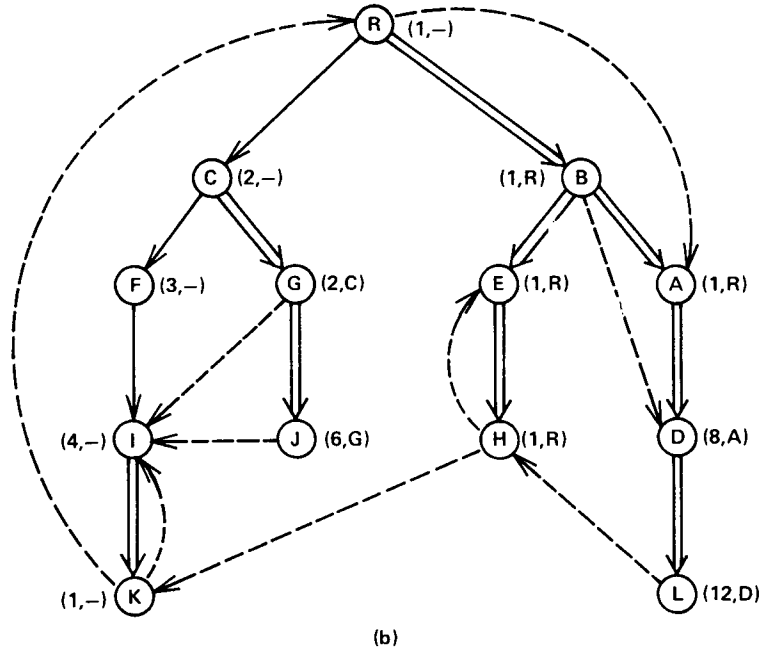

Fig. 4(b). Snapshot just before processing vertex I

The simple method uses *path compression* to carry out EVAL. To represent the forest built by the LINK instructions (henceforth called the *forest*), the algorithm uses two arrays, *ancestor* and *label*. Initially $ancestor(v) = 0$ and $label(v) = v$ for each vertex v. In general $ancestor(v) = 0$ only if v is a tree root in the forest; otherwise $ancestor(v)$ is an ancestor of v in the forest.

The algorithm maintains the labels so that they satisfy the following property. Let v be any vertex, let r be the root of the tree in the forest containing v, and let $v = v_k, v_{k-1}, \ldots, v_0 = r$ be such that $ancestor(v_i) = v_{i-1}$ for $1 \leq i \leq k$. Let x be a vertex such that $semi(x)$ is minimum among vertices $x \in \{label(v_i) \mid 1 \leq i \leq k\}$. Then

$$x \text{ is a vertex such that } semi(x) \text{ is minimum among vertices } x \text{ satisfying } r \xrightarrow{+} x \dashrightarrow v. \tag{3}$$

To carry out LINK(v, w), the algorithm assigns $ancestor(w) := v$. To carry out EVAL(v), the algorithm follows ancestor pointers to determine the sequence $v = v_k, v_{k-1}, \ldots, v_0 = r$ such that $ancestor(v_i) = v_{i-1}$ for $1 \leq i \leq k$. If $v = r$, v is returned. Otherwise, the algorithm performs a *path compression* by assigning $ancestor(v_i) := r$ for i from 2 to k, simultaneously updating labels to maintain (3) as follows: If $semi(label(v_{i-1})) < semi(label(v_i))$, then $label(v_i) := label(v_{i-1})$. Then $label(v)$ is returned. Here is an Algol-like procedure for EVAL.

```
vertex procedure EVAL(v);
  if ancestor(v) = 0 then EVAL := v
    else COMPRESS(v); EVAL := label(v) fi;
```

Recursive procedure COMPRESS, which carries out the path compression, is defined by

```
procedure COMPRESS(v);
  comment this procedure assumes ancestor(v) ≠ 0;
  if ancestor(ancestor(v)) ≠ 0 then
    COMPRESS(ancestor(v));
    if semi(label(ancestor(v))) < semi(label(v)) then
      label(v) := label(ancestor(v)) fi;
    ancestor(v) := ancestor(ancestor(v)) fi;
```

The time required for $n - 1$ LINKs and $m + n - 1$ EVALs using this implementation is $O(m \log n)$ [14]. Thus the simple version of the dominators algorithm requires $O(m \log n)$ time.

The sophisticated method uses path compression to carry out the EVAL instructions but implements the LINK instruction so that path compression is carried out only on *balanced* trees. See [14]. The sophisticated method requires two additional arrays, *size* and *child*. Initially $size(v) = 1$ and $child(v) = 0$ for all vertices v. Here are Algol-like implementations of EVAL and LINK using the sophisticated method. These procedures are adapted from [14].

```
vertex procedure EVAL(v);
  comment procedure COMPRESS used here is identical to that in the
    simple method.
  if ancestor(v) = 0 then EVAL := label(v)
    else COMPRESS(v);
      EVAL := if semi(label(ancestor(v))) ≥ semi(label(v))
                then label(v) else label(ancestor(v)) fi fi;
```

```
procedure LINK(v, w);
  begin
    comment this procedure assumes for convenience that
      size(0) = label(0) = semi(0) = 0;
    s := w;
    while semi(label(w)) < semi(label(child(s))) do
      if size(s) + size(child(child(s))) ≥ 2*)size(child(s)) then
        parent(child(s)) := s; child(s) := child(child(s))
      else size(child(s)) := size(s);
        s := parent(s) := child(s) fi od;
    label(s) := label(w);
    size(v) := size(v) + size(w);
    if size(v) < 2*size(w) then s, child(v) := child(v), s fi;
    while s ≠ 0 do parent(s) := v; s := child(s) od
  end LINK;
```

With this implementation, the time required for $n - 1$ LINKs and $m + n - 1$ EVALs is $O(m\alpha(m, n))$, where α is the functional inverse of Ackerman's function [1] defined in the Introduction. Thus the sophisticated version of the dominators algorithm requires $O(m\alpha(m, n))$ time.

5. EXPERIMENTAL RESULTS AND CONCLUSIONS

We performed extensive experiments in order to qualitatively compare the actual performance of our algorithm with that of the Purdom-Moore algorithm [7] and that of the bit vector algorithm. We translated both versions of our algorithm as contained in Appendix B into Algol W and ran the programs on a series of randomly generated program flowgraphs.

Table I and Figures 5 and 6 illustrate the results. The sophisticated version beat the simple version on all graphs tested. The relative difference in speed was between 5 and 25 percent increasing with increasing n. It is important to note that the running times of the algorithms are insensitive to the way the test graphs are selected; for fixed m and n the running times vary very little on different graphs, whether the graphs are chosen randomly or by some other method. This is also true for the Purdom-Moore algorithm.

Table I. Running Times in 10^{-3} Sec of the Simple and Sophisticated Versions of the Fast Algorithm (Three Graphs for Each Value of n)

	Simple		Sophisticated			Simple		Sophisticated	
n	Min	Max	Min	Max	n	Min	Max	Min	Max
10	2.0	2.1	1.9	2.0	200	46.4	47.2	36.2	36.4
20	4.3	4.4	3.7	3.9	300	70.1	72.3	55.0	55.7
30	6.2	6.8	5.5	5.8	400	98.5	101	74.7	78.1
40	8.0	8.8	7.1	7.6	500	123	125	92.0	93.7
50	10.5	11.4	8.9	9.6	600	150	152	110	120
60	12.4	13.4	10.9	11.6	700	176	181	130	137
70	14.6	15.4	12.6	13.1	800	214	217	158	167
80	17.4	18.6	14.5	15.6	900	238	244	173	188
90	20.0	20.2	16.7	16.8	1000	263	268	192	206
100	22.4	22.7	18.0	19.3					

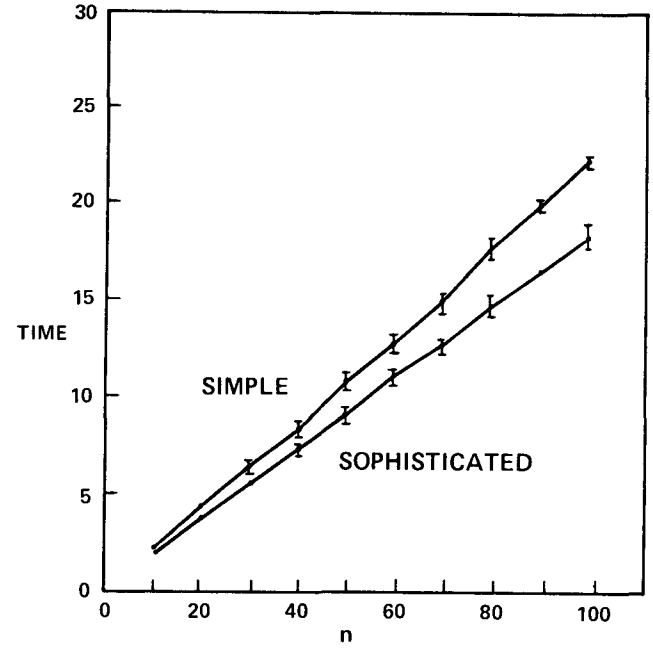

Fig. 5. Running times in 10^{-3} sec of the simple and sophisticated versions of the fast algorithm

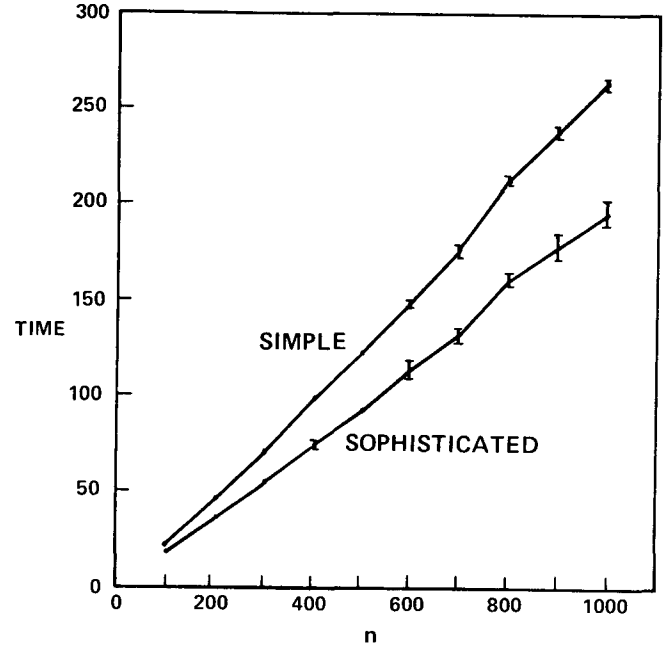

Fig. 6. Running times in 10^{-3} sec of the simple and sophisticated versions of the fast algorithm

We transcribed the Purdom-Moore algorithm into Algol W and ran it and the sophisticated version of our algorithm on another series of program flowgraphs. Table II and Figure 7 show the results. Our algorithm was faster on all graphs tested except those with $n = 8$. The Purdom-Moore algorithm rapidly became noncompetitive as n increased. The tradeoff point was about $n = 10$.

We implemented the bit vector algorithm using a set of procedures for manipulating multiprecision bit vectors. (Algol W allows bit vectors only of length 32 or less.) Table III gives the running time of this algorithm on the second series of test graphs, and Figure 8 compares the running times of the bit vector algorithm and the sophisticated version of our algorithm. The speed of the bit vector algorithm varied not only with m and n, but also with the number of passes required (two, three, or four on our test graphs). However, the bit vector method was always slower than our algorithm.

There are several ways in which the bit vector algorithm can be made more competitive. First, the bit vector procedures can be inserted in-line to save the overhead of procedure calls. We made this change and observed a 33–45-percent speedup. The corresponding change in the fast algorithm, inserting LINK and EVAL in-line, produced a 20-percent speedup. These changes made the bit vector algorithm almost as fast as our algorithm on graphs of less than 32 vertices, but on larger graphs the bit vector algorithm remained substantially slower than our algorithm. See Tables I and IV and Figure 9.

Second, the bit vector procedures can be written in assembly language. To provide a fair comparison with the fast algorithm, it would be necessary to write LINK and EVAL in assembly language. We did not try this approach, but we

Table II. Running Times in 10^{-3} Sec of the Purdom-Moore Algorithm and the Sophisticated Version of the Fast Algorithm (Three Graphs for Each Value of n)

	Sophisticated		In-line sophisticated		Purdom-Moore	
n	Min	Max	Min	Max	Min	Max
8	1.7	1.7	1.4	1.5	1.3	1.4
16	3.0	3.2	2.5	2.6	4.6	4.7
24	4.4	4.5	3.6	3.7	10.1	10.3
32	5.8	6.1	4.7	4.8	18.4	18.6
40	7.4	7.6	6.0	6.1	29.4	29.6
48	8.8	9.2	7.0	7.4	40.8	42.5
56	10	11	8.0	8.8	56.5	58.2
64	12	13	9.3	10.0	74.3	75.5
72	13.2	13.8	10.3	10.9		
80	14.9	15.1	11.8	12.0		
88	16.5	17.4	13.0	13.9		
96	17.7	17.9	14.0	14.5		
104	19.3	20.4	15.4	16.4		
112	19.9	20.6	15.9	16.7		
120	22.3	23.4	17.7	19.0		
128	23.5	23.8	18.7	19.2		

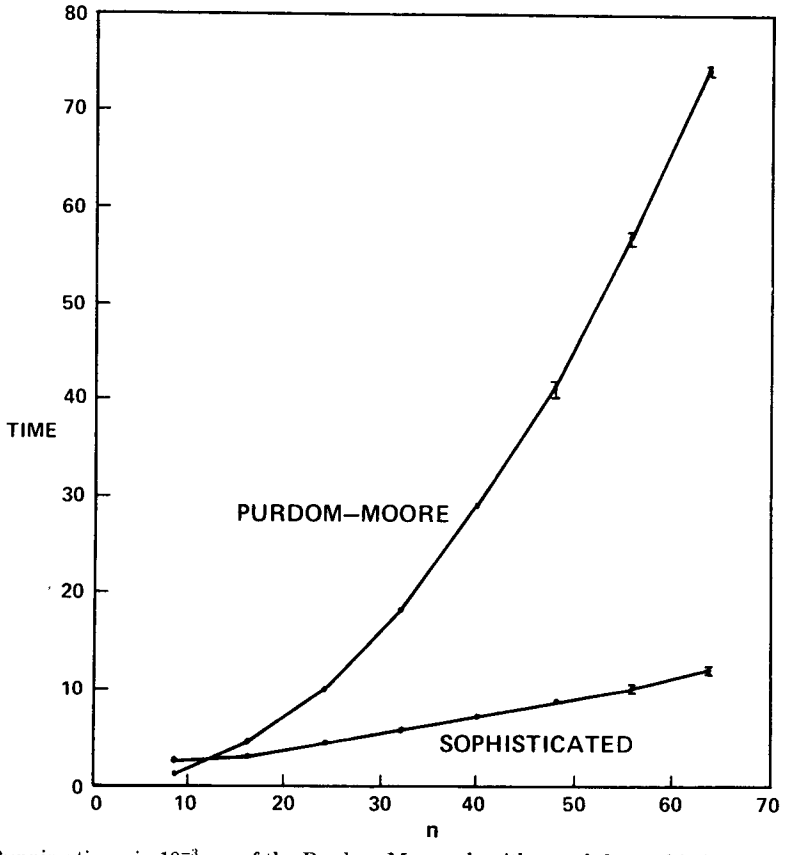

Fig. 7. Running times in 10^{-3} sec of the Purdom-Moore algorithm and the sophisticated version of the fast algorithm

Table III. Running Times in 10^{-3} Sec and Number of Passes of the Bit Vector Algorithm (Three Graphs for Each Value of n)

	Bit vector					
n	Time	Passes	Time	Passes	Time	Passes
8	3.2	3	3.4	3	3.4	3
16	6.3	3	6.3	3	6.4	3
24	9.3	3	9.4	3	9.5	3
32	12.4	3	12.4	3	15.7	4
40	12.8	2	12.9	2	17.3	3
48	20.9	3	20.9	3	21.0	3
56	24.3	3	24.3	3	24.3	3
64	27.9	3	28.2	3	28.2	3
72	25.6	2	35.1	3	35.5	3
80	28.6	2	39.2	3	39.6	3
88	43.7	3	43.8	3	44.1	3
96	46.6	3	47.7	3	47.7	3
104	40.6	2	41.0	2	56.0	3
112	43.9	2	43.9	2	61.3	3
120	65.9	3	66.0	3	66.6	3
128	70.5	3	71.3	3	91.5	4

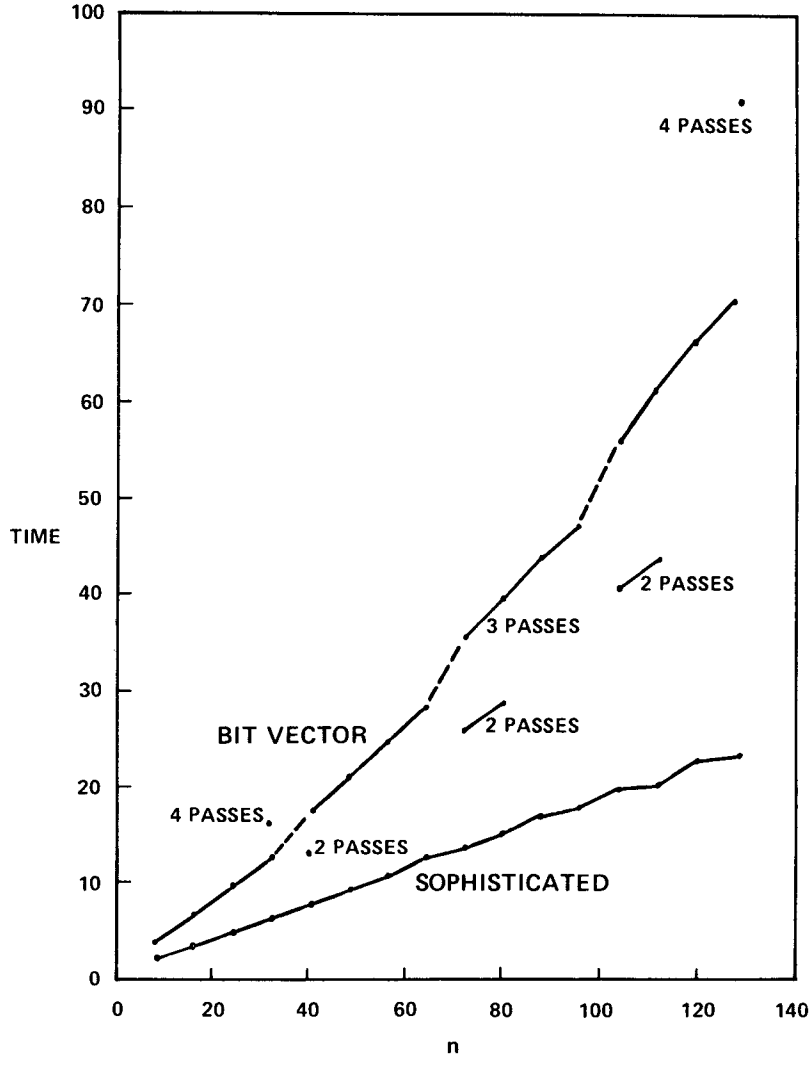

Fig. 8. Running times in 10^{-3} sec of the bit vector algorithm and the sophisticated version of the fast algorithm

believe that the fast algorithm would still beat the bit vector algorithm on graphs of moderate size.

Third, use of the bit vector algorithm can be restricted to graphs known to be reducible. On a reducible graph only one pass of the bit vector algorithm is necessary, because the only purpose served by the second pass is to prove that the bit vectors do not change, a fact guaranteed by the reducibility of the graph. We believe that a one-pass in-line bit vector algorithm would be competitive with the fast algorithm on reducible graphs of moderate size, but only if one ignores the time needed to test reducibility.

The bit vector algorithm has two disadvantages not possessed by the fast algorithm. First, it requires $O(n^2)$ storage, which may be prohibitive for large

Table IV. Running Times in 10^{-3} Sec and Number of Passes of the In-Line Bit Vector Algorithm (Three Graphs for Each Value of n)

	In-line bit vector					
n	Time	Passes	Time	Passes	Time	Passes
8	1.8	3	1.8	3	1.9	3
16	3.3	3	3.4	3	3.4	3
24	4.9	3	5.0	3	5.1	3
32	6.4	3	6.5	3	7.9	4
40	7.7	2	7.7	2	10.1	3
48	12.1	3	12.2	3	12.4	3
56	14.2	3	14.2	3	14.2	3
64	16.1	3	16.3	3	16.3	3
72	16.8	2	22.4	3	22.7	3
80	18.4	2	24.7	3	24.8	3
88	27.1	3	27.5	3	27.8	3
96	29.5	3	29.6	3	29.8	3
104	27.1	2	27.2	2	38.1	3
112	30.4	2	30.8	2	41.5	3
120	44.0	3	44.1	3	44.3	3
128	46.5	3	46.9	3	60.6	4

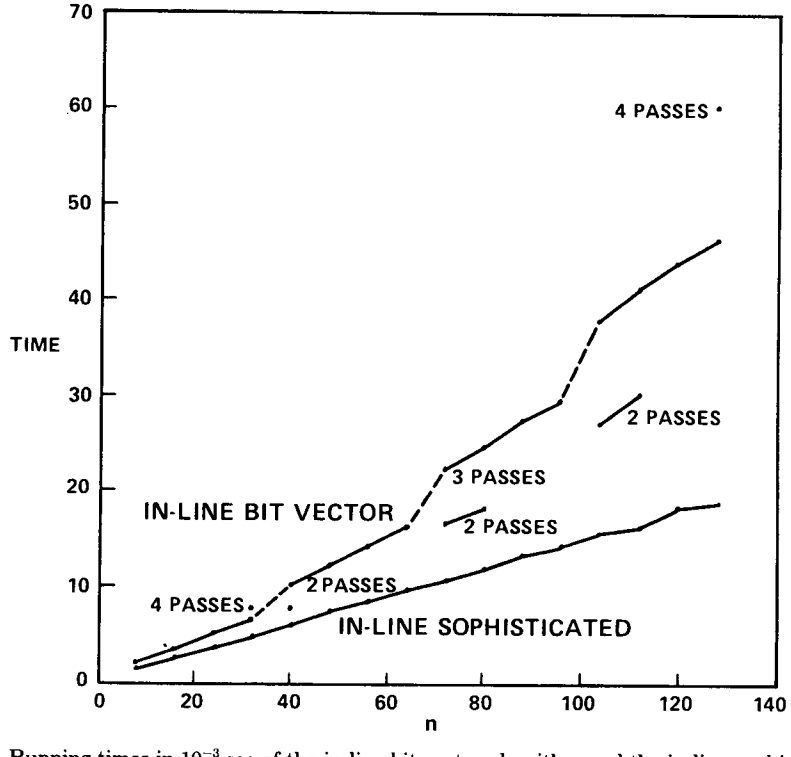

Fig. 9. Running times in 10^{-3} sec of the in-line bit vector algorithm and the in-line sophisticated version of the fast algorithm

values of n. Second, the dominator tree, not the dominator relation, is required for many kinds of global flow analysis [8, 13], but the bit vector algorithm computes only the dominator relation. Computing the relation from the tree is easy, requiring constant time per element of the relation or $O(n)$ bit vector operations total. However, computing the tree from bit vectors encoding the relation requires $O(n^2)$ time in the worst case.

We can summarize the good and bad points of the three algorithms as follows: The Purdom-Moore algorithm is easy to explain and easy to program but slow on all but small graphs. The bit vector algorithm is equally easy to explain and program, faster than the Purdom-Moore algorithm, but not competitive in speed with the fast algorithm unless it is run on small graphs which are reducible or almost reducible. The fast algorithm is much harder to prove correct but almost as easy to program as the other two algorithms, is competitive in speed on small graphs, and is much faster on large graphs. We favor some version of the fast algorithm for practical applications.

We conclude with a few comments on ways to improve the efficiency of the fast algorithm. One can speed up the algorithm by rewriting DFS and COMPRESS as nonrecursive procedures which use explicit stacks. One can avoid using an auxiliary stack for COMPRESS by instead using a trick of reversing *ancestor* pointers; see [12]. A similar trick allows one to avoid the use of an auxiliary stack for DFS. One can save some additional storage by combining certain arrays, such as *parent* and *ancestor*. These modifications save running time and storage space, but only at the expense of program clarity.

APPENDIX A. GRAPH-THEORETIC TERMINOLOGY

A *directed graph* $G = (V, E)$ consists of a finite set V of *vertices* and a set E of ordered pairs (v, w) of distinct vertices, called *edges*. If (v, w) is an edge, w is a *successor* of v and v is a *predecessor* of w. A graph $G_1 = (V_1, E_1)$ is a *subgraph* of G if $V_1 \subseteq V$ and $E_1 \subseteq E$. A *path* p of *length* k from v to w in G is a sequence of vertices $p = (v = v_0, v_1, \ldots, v_k = w)$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. The path is *simple* if $v_0, \ldots, v_k$ are distinct (except possibly $v_0 = v_k$), and the path is a *cycle* if $v_0 = v_k$. By convention there is a path of no edges from every vertex to itself, but a cycle must contain at least two edges. A graph is *acyclic* if it contains no cycles. If $p_1 = (u = u_0, u_1, \ldots, u_k = v)$ is a path from u to v and $p = (v = v_0, v_1, \ldots, v_l = w)$ is a path from v to w, the path p_1 *followed by* p_2 is $p = (u = u_0, u_1, \ldots, u_k = v = v_0, v_1, \ldots, v_l = w)$.

A *flowgraph* $G = (V, E, r)$ is a directed graph (V, E) with a distinguished *start vertex* r such that for any vertex $v \in V$ there is a path from r to v. A *program flowgraph* is a flowgraph such that each vertex has exactly two successors. A *(directed, rooted) tree* $T = (V, E, r)$ is a flowgraph such that $|E| = |V| - 1$. The start vertex r is the *root* of the tree. Any tree is acyclic, and if v is any vertex in a tree T, there is a unique path from r to v. If v and w are vertices in a tree T and there is a path from v to w, then v is an *ancestor* of w and w is a *descendant* of v (denoted by $v \overset{*}{\rightarrow} w$). If in addition $v \neq w$, then v is a *proper ancestor* of w and w is a *proper descendant* of v (denoted by $v \overset{+}{\rightarrow} w$). If $v \overset{*}{\rightarrow} w$ and (v, w) is an edge of T (denoted by $v \rightarrow w$), then v is the *parent* of w and w is a *child* of v. In a tree

each vertex has a unique parent (except the root, which has no parent). If $G = (V, E)$ is a graph and $T = (V', E', r)$ is a tree such that (V', E') is a subgraph of G and $V = V'$, then T is a *spanning tree* of G.

APPENDIX B. THE COMPLETE DOMINATORS ALGORITHM

This appendix contains a complete listing of both versions of the dominators algorithm. The algorithm assumes that the vertex set of the problem graph is $V = \{v \mid 1 \leq v \leq n\}$.

```
procedure DOMINATORS(integer set array succ(1 :: n); integer r, n; integer array
    dom(1 :: n));
  begin
    integer array parent, ancestor, [child, ] vertex(1 :: n);
    integer array label, semi [, size](0 :: n);
    integer set array pred, bucket(1 :: n);
    integer u, v, x;

    procedure DFS(integer v);
      begin
        semi(v) := n := n + 1;
        vertex(n) := label(v) := v;
        ancestor(v) := [child(v) :=] 0;
        [size(v) := 1;]
        for each w ∈ succ(v) do
          if semi(w) = 0 then parent(w) := v; DFS(w) fi;
          add v to pred(w) od
      end DFS;

    procedure COMPRESS(integer v);
      if ancestor(ancestor(v)) ≠ 0 then
        COMPRESS(ancestor(v));
        if semi(label(ancestor(v))) < semi(label(v)) then
          label(v) := label(ancestor(v)) fi;
        ancestor(v) := ancestor(ancestor(v)) fi;

    integer procedure EVAL(integer v);
      if ancestor(v) = 0 then EVAL := v
        else COMPRESS(v); EVAL := label(v) fi;

    procedure LINK(integer v, w);
      ancestor(w) := v;

    step1:  for v := 1 until n do
              pred(v) := bucket(v) := ∅; semi(v) := 0 od;
            n := 0;
            DFS(r);
            [size(0) := label(0) := semi(0) := 0;]
            for i := n by −1 until 2 do
              w := vertex(i);
    step2:  for each v ∈ pred(w) do
              u := EVAL(v);
                if semi(u) < semi(w) then semi(w) := semi(u) fi od;
              add w to bucket(vertex(semi(w)));
              LINK(parent(w), w);
    step3:  for each v ∈ bucket(parent(w)) do
              delete v from bucket(parent(w));
              u := EVAL(v);
```

```
            dom(v) := if semi(u) < semi(v) then u
                      else parent(w) fi od od;
step4:  i := 2 until n do
            w := vertex(i);
            if dom(w) ≠ vertex(semi(w))
              then dom(w) := dom(dom(w)) fi od;
        dom(r) := 0
end DOMINATORS;
```

The simple version of the algorithm consists of the procedure above, with everything in brackets deleted. The sophisticated version of the algorithm consists of the procedure above, with everything in brackets included, and the following procedures substituted for EVAL and LINK.

```
integer procedure EVAL(integer v);
  if ancestor(v) = 0 then EVAL := label(v)
    else COMPRESS(v);
      EVAL := if semi(label(ancestor(v))) ≥ semi(label(v))
                then label(v) else label(ancestor(v)) fi fi;

procedure LINK(integer v, w);
  begin integer s;
    s := w;
    while semi(label(w)) < semi(label(child(s))) do
      if size(s) + size(child(child(s))) ≥ 2·size(child(s))
        then ancestor(child(s)) := s;
          child(s) := child(child(s))
      else size(child(s)) := size(s);
        s := ancestor(s) := child(s) fi od;
    label(s) := label(w);
    size(v) := size(v) + size(w);
    if size(v) < 2·size(w) then s, child(v) := child(v), s fi;
    while s ≠ 0 do ancestor(s) := v; s := child(s) od
  end LINK;
```

REFERENCES

1. Ackermann, W. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann. 99* (1928), 118–133.
2. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
3. Aho, A.V., and Ullman, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
4. Hecht, M.S., and Ullman, J.D. A simple algorithm for global data flow analysis problems. *SIAM J. Comput. 4* (1973), 519–532.
5. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
6. Lorry, E.S., and Medlock, C.W. Object code optimization. *Comm. ACM 12,* 1 (Jan. 1969), 13–22.
7. Purdom, P.W., and Moore, E.F. Algorithm 430: Immediate predominators in a directed graph. *Comm. ACM 15,* 8 (Aug. 1972), 777–778.
8. Reif, J. Combinatorial aspects of symbolic program analysis. Tech. Rep. TR-11-77, Center for Research in Computing Technology, Harvard U., Cambridge, Mass., 1977.
9. Tarjan, R.E. Depth-first search and linear graph algorithms. *SIAM J. Comptng. 1* (1972), 146–160.
10. Tarjan, R. Finding dominators in directed graphs *SIAM J. Comptng. 3* (1974), 62–89.

11. TARJAN, R.E. Edge-disjoint spanning trees, dominators, and depth-first search. Tech. Rep. STAN-CS-74-455, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1974.
12. TARJAN, R.E. Applications of path compression on balanced trees. Tech. Rep. STAN-CS-75-512, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1975.
13. TARJAN, R.E. Solving path problems on directed graphs. Tech. Rep. STAN-CS-528, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1975.
14. TARJAN, R.E. Applications of path compression on balanced trees. To appear in *J. ACM*.

Received December 1977; revised March 1979