FOR FURTHER TRAN [illegible]

(12)

(6)

A FAST ALGORITHM FOR FINDING DOMINATORS IN A FLOW GRAPH

by

(10)

Thomas Lengauer and Robert Endre Tarjan

ADA054144

(14) STAN-CS-78-658

(11) March 1978

(12) 42p.

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

(15) N00014-76-C-0688, NSF-MCS75-22870

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: CS-650	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Fast Algorithm for Finding Dominators in a Flow Graph.		5. TYPE OF REPORT & PERIOD COVERED: Analysis of Algorithms
		6. PERFORMING ORG. REPORT NUMBER: STAN-CS-78-650
7. AUTHOR(s): Thomas Lengauer & Robert Tarjan		8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0688
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University Comp Sci Stanford, CA [illegible]		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Thomas Lengauer Stanford, CA		12. REPORT DATE: March 1978
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without Limitations on dissemination

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

[illegible] FAST ALGORITHM FOR FINDING DOMINATORS IN A FLOW GRAPH

Authors: Thomas Lengauer & Robert E. Tarjan

This paper presents a fast algorithm for finding dominators in a flow graph. 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 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 [illegible]

DD FORM 1473 [illegible]

A Fast Algorithm for Finding Dominators in a Flow Graph

Thomas Lengauer[*]
Robert Endre Tarjan[**]

Computer Science Department
Stanford University
Stanford, California 94305

Abstract.

This paper presents a fast algorithm for finding dominators in a flow graph. 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.

Keywords: depth-first search, dominators, global flow analysis, graph algorithm, path compression.

[*] Supported by the German Academic Exchange Service.

[**] This research was supported in part by National Science Foundation grant MCS75-22870 and by Office of Naval Research contract N00014-76-C-0688. Reproduction in whole or in part is permitted for any purpose of the United States government.

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 flow graph 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 $v = \underline{\text{idom}}(w)$, if v dominates w and every other dominator of w dominates v .

Theorem 1 [2,6]. Every vertex of a flow graph $G = (V,E,r)$ except r has a unique immediate dominator. The edges $\{(\underline{\text{idom}}(w),w) \mid w \in V-\{r\}\}$ 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.

[Figure 1]

[Figure 2]

We wish to construct the dominator tree of an arbitrary flow graph G . Aho and Ullman [2] and Purdom and Moore [7] describe a straightforward algorithm for solving this problem. 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.

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

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 $O(mn)$ 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 flow graphs [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 [13]. 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.

The algorithm is a refinement of earlier versions appearing in [10,11,12]. 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.

The 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 Section 2 of [10] 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 Sections 1, 2, and 5 of [13] is useful for understanding this section. Section 5 presents our experimental results.

2. Depth-First Search and Dominators.

Suppose we perform a depth-first search on a flow graph $G = (V,E,r)$ starting from vertex r, and that we number the vertices of G from 1 to n as they are reached during the search. The search generates a spanning tree T rooted at r, with vertices numbered in preorder [5]. See Figure 3.

[Figure 3]

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

<u>Lemma 1</u> [9]. If v and w are vertices of G such that $\underline{\text{number}}(v) \leq \underline{\text{number}}(w)$, then any path from v to w in G must contain a common ancestor of v and w in T.

As an intermediate step, the dominators algorithm computes a value for each vertex $w \neq r$ called its semi-dominator, denoted by $\underline{\text{sdom}}(w)$ and defined by

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

See Figure 3.

The following lemmas describe some basic properties of semi-dominators and immediate dominators.

<u>Lemma 2</u>. For any vertex $w \neq r$, let v be the vertex such that $\underline{\text{number}}(v) = \underline{\text{sdom}}(w)$. Then v is a proper ancestor of w in T.

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

Lemma 3. For any vertex $w \neq r$, let v be the vertex such that number(v) = sdom(w). Then idom(w) is an ancestor of v in T.

Proof. The tree path from r to w contains only ancestors of w in T. Thus idom(w) is an ancestor of w. The path consisting of the tree path from r to v followed by a path $v = v_0, v_1, \ldots, v_k = w$ such that number(v_i) > number(w) for $1 \leq i \leq k-1$ (which must exist by (1)) avoids all proper descendants of v which are also proper ancestors of w. It follows that idom(w) is an ancestor of v. □

Corollary 1. For any vertex $w \neq r$, idom(w) $\overset{*}{\rightarrow}$ w.

Lemma 4. Let vertices v, w satisfy $v \overset{*}{\rightarrow} w$ in T. 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 $\underline{\text{idom}}(w)$ must be either a descendant of v or an ancestor of $\underline{\text{idom}}(v)$. □

Using Lemmas 1 - 4, we obtain two results which provide a way to compute immediate dominators from semi-dominators.

Theorem 2. Let $w \neq r$ and let v be the vertex such that $\underline{\text{number}}(v) = \underline{\text{sdom}}(w)$. Suppose no vertex u satisfies $\underline{\text{number}}(u) > \underline{\text{number}}(v)$, $u \overset{*}{\rightarrow} w$, and $\underline{\text{sdom}}(u) < \underline{\text{sdom}}(w)$. Then $\underline{\text{idom}}(w) = v$.

Proof. By Lemma 3, it suffices to show that v dominates w . Consider any path from r to w . Let x be the last vertex on this path satisfying $\underline{\text{number}}(x) < \underline{\text{number}}(v)$. If there is no such x , then $v = r$ dominates w . Otherwise, let y be the first vertex following x on the path and satisfying $v \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$. All vertices z following x on the path but preceeding y must satisfy $z > y$ by Lemma 1 and the choice of x and y . Thus $\underline{\text{sdom}}(y) \leq \underline{\text{number}}(x) < \underline{\text{number}}(v) = \underline{\text{sdom}}(w)$. By the hypothesis of the theorem, y cannot be a proper descendant of v . Thus $y = v$ and v lies on the path. Since the path selected was arbitrary, v dominates w . □

Theorem 3. Let $w \neq r$ and let v be the vertex such that $\underline{\text{number}}(v) = \underline{\text{sdom}}(w)$. Let u be a vertex for which $\underline{\text{sdom}}(u)$ is minimum among vertices satisfying $\underline{\text{number}}(u) > \underline{\text{number}}(v)$ and $u \overset{*}{\rightarrow} w$. Then $\underline{\text{sdom}}(u) \leq \underline{\text{sdom}}(w)$ and $\underline{\text{idom}}(u) = \underline{\text{idom}}(w)$.

Proof. Let x be the vertex such that $v \rightarrow x \overset{*}{\rightarrow} w$. Then $\underline{\text{sdom}}(u) \leq \underline{\text{sdom}}(x) \leq \underline{\text{number}}(v) = \underline{\text{sdom}}(w)$.

By Lemma 3, $\underline{\text{idom}}(w)$ is an ancestor of v and thus a proper ancestor of u . Thus by Lemma 4 $\underline{\text{idom}}(w) \overset{*}{\rightarrow} \underline{\text{idom}}(u)$. To prove $\underline{\text{idom}}(u) = \underline{\text{idom}}(w)$, it suffices to prove that $\underline{\text{idom}}(u)$ dominates w .

Consider any path from r to w . Let x be the last vertex on this path satisfying $\underline{\text{number}}(x) < \underline{\text{number}}(\underline{\text{idom}}(u))$. If there is no such x , then $\underline{\text{idom}}(u) = r$ dominates w . Otherwise, let y be the first vertex following x on the path and satisfying $\underline{\text{idom}}(u) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} w$. All vertices z following x on the path but preceding y satisfy $\underline{\text{number}}(z) > \underline{\text{number}}(y)$ by Lemma 1 and the choice of x and y. Thus $\underline{\text{sdom}}(y) \leq \underline{\text{number}}(x)$. Since $\underline{\text{number}}(\underline{\text{idom}}(u)) \leq \underline{\text{sdom}}(u)$ by Lemma 3, we have $\underline{\text{sdom}}(y) \leq \underline{\text{number}}(x) < \underline{\text{number}}(\underline{\text{idom}}(u)) \leq \underline{\text{sdom}}(u)$.

By the definition of u , y cannot be a proper descendant of v . Furthermore y cannot be both a proper descendant of $\underline{\text{idom}}(u)$ and an ancestor of u , for if this were the case the path consisting of the tree path from r to $\underline{\text{sdom}}(y)$ followed by a path $\underline{\text{sdom}}(y) = v_0, v_1, \ldots, v_k = y$ such that $\underline{\text{number}}(v_i) > \underline{\text{number}}(y)$ for $1 \leq i \leq k-1$ followed by the tree path from y to u would avoid $\underline{\text{idom}}(u)$; but no path from r to u avoids $\underline{\text{idom}}(u)$.

The only remaining possibility is that $\underline{\text{idom}}(u) = y$. Thus $\underline{\text{idom}}(u)$ lies on the path from r to w . Since the path selected was arbitrary, $\underline{\text{idom}}(u)$ dominates w . $\square$

Corollary 2. Let $w \neq r$ and let v be the vertex such that $\underline{\text{number}}(v) = \underline{\text{sdom}}(w)$. Let u be a vertex for which $\underline{\text{sdom}}(u)$ is minimum among vertices satisfying $\underline{\text{number}}(u) > \underline{\text{number}}(v)$ and $u \overset{*}{\rightarrow} w$. Then

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

Proof. Immediate from Theorems 2 and 3. □

The following theorem provides a way to compute semi-dominators.

Theorem 4. For any vertex $w \neq r$,

$$(3) \quad \underline{\text{sdom}}(w) = \min(\{\underline{\text{number}}(v) \mid (v,w) \in E \text{ and } \underline{\text{number}}(v) < \underline{\text{number}}(w)\} \cup \{\underline{\text{sdom}}(u) \mid \underline{\text{number}}(u) > \underline{\text{number}}(w) \text{ and there is edge } (v,w) \text{ such that } u \overset{*}{\rightarrow} v \text{ in } T\}) \ .$$

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

It remains for us to prove that $\underline{\text{sdom}}(w) \geq \ell$. Let x be the vertex such that $\underline{\text{number}}(x) = \underline{\text{sdom}}(w)$, and let $x = v_0, v_1, \ldots, v_k = w$ be a simple path such that $\underline{\text{number}}(v_i) > \underline{\text{number}}(w)$ for $1 \leq i \leq k-1$. If $k = 1$, $(x,w) \in E$, and $\underline{\text{number}}(x) < \underline{\text{number}}(w)$ by Lemma 2. Thus $\underline{\text{sdom}}(w) = \underline{\text{number}}(x) \geq \ell$. 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 $\underline{\text{number}}(v_i) > \underline{\text{number}}(v_j)$ for $1 \leq i \leq j-1$. Suppose to the contrary that $\underline{\text{number}}(v_i) < \underline{\text{number}}(v_j)$ for some i in the range $1 \leq i \leq j-1$. Choose the i such that $1 \leq i \leq j-1$ and $\underline{\text{number}}(v_i)$ is minimum. By Lemma 1, $v_i \overset{*}{\rightarrow} v_j$, which contradicts the choice of j . This proves the claim.

The claim implies $\underline{\text{sdom}}(w) = \underline{\text{number}}(x) \geq \underline{\text{sdom}}(v_j) \geq \ell$. Thus whether $k = 1$ or $k > 1$ we have $\underline{\text{sdom}}(w) \geq \ell$, 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 [10,11,12]; 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. For each vertex w, determine the set pred(w) of vertices v such that (v,w) is an edge and the vertex parent(w) which is the parent of w in the spanning tree generated by the search. Initialize the variables used in succeeding steps.

Step 2. Compute the semi-dominators 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 2.

Step 4. Explicitly define the immediate dominator of each vertex, carrying out the computation vertex-by-vertex in increasing order by number.

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. The procedure assumes that succ(v) is the set of vertices w such that $(v,w) \in E$. When a vertex v receives a number i, the procedure assigns semi(v) := i and vertex(i) := v.

```
procedure DFS(vertex v);
    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 $v \neq r$ in decreasing order by number. When processing a vertex v, the algorithm computes sdom(v) by applying Theorem 4. Each edge (u,v) is examined. If number(u) < number(v), number(u) is a candidate for sdom(v). If number(u) > number(v), the algorithm finds a vertex x of minimum sdom(x) among vertices satisfying number(x) > number(v) and $x \overset{*}{\rightarrow} u$; sdom(x) is a candidate for sdom(v). The minimum of all the candidates is sdom(v). After computing sdom(v), the algorithm assigns semi(v) := sdom(v) and adds vertex v to the set bucket(u), where u is the vertex such that number(u) = sdom(v). This completes Step 2 for v. Note that before sdom(v) is found, semi(v) = number(v), and after sdom(v) is found, semi(v) = sdom(v).

After the semi-dominator of v is computed, the algorithm empties bucket(parent(v)) . For each vertex $w \in$ bucket(parent(v)) , the algorithm finds a vertex u of minimum sdom(u) among vertices satisfying number(u) > number(parent(v)) and $u \overset{*}{\rightarrow} w$. If sdom(u) = sdom(w) , then by Corollary 2 the immediate dominator of w is parent(v) , and the algorithm assigns dom(w) := parent(v) . If sdom(u) < sdom(w) , then by Corollary 2, u and w have the same immediate dominator, and the algorithm assigns dom(w) := u . The intent of this assignment is to implicitly define the immediate dominator of w to be the immediate dominator of a vertex with smaller semi-dominator than w . This completes Step 3 for vertices $w \in$ bucket(parent(v)) .

Both Step 2 and Step 3 require determining, for certain paths $v \overset{*}{\rightarrow} w$ in the spanning tree, a vertex u on the path $v \overset{*}{\rightarrow} w$ having minimum sdom(u) . To find such vertices the algorithm uses a method described in [10]. The algorithm maintains a data structure which represents a forest with vertex set V and edge set {(parent(v),v) | sdom(v) has been computed} . To manipulate this data structure, the algorithm uses two procedures:

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 a vertex $u \neq r$ of minimum sdom(u) on the path $r \overset{*}{\rightarrow} v$ in the forest.

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
         v := vertex(i);
 step2:  for each u ∈ pred(v) do
              x := EVAL(u); if semi(x) < semi(v) then semi(v) := semi(x) od;
         LINK(parent(v),v);
         add v to bucket(vertex(semi(v)));
 step3:  for each w ∈ bucket(parent(v)) do
              delete w from bucket(parent(v));
              u := EVAL(w);
              dom(w) := if semi(u) < semi(parent(v)) then u
                             else parent(v) 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
             v := vertex(i);
             if dom(v) ≠ vertex(semi(v)) then dom(v) := dom(dom(v)) od;
```

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

[Figure 4]

Appendix B contains a complete Algol-like version of the algorithm, including variable declarations and initialization. Using Theorem 4 and Corollary 2 it is not hard to prove that, after execution of the algorithm,

$\underline{dom}(v) = \underline{idom}(v)$ 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.

Reference [13] provides two ways to implement LINK and EVAL, one simple and one sophisticated. We shall not discuss the details of these methods here, but merely provide Algol-like implementations of LINK and EVAL which are adapted from [13].

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 sdom(x) is minimum among vertices $x \in \{$label$(v_i) \mid 1 \leq i \leq k\}$. Then

(*) x is a vertex such that sdom(x) is minimum among vertices x satisfying $r \overset{+}{\rightarrow} x \overset{*}{\rightarrow} v$ in the forest.

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 $s \leq i \leq k$, updating labels to maintain (*). 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 $O(m \log n)$ [13]. 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 [13]. 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 [13].

```
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 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 $\alpha(m,n) = \min\{i \geq 1 \mid A(i,\lfloor 2m/n \rfloor) > \log_2 n\}$. Thus the sophisticated version of the dominators algorithm requires $O(m\,\alpha(m,n))$ time.

5. Implementation and Experimental Results.

We translated both versions of the algorithm as contained in Appendix B into Algol W and ran the programs on a series of randomly generated program flow graphs. Table 1 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%, increasing with increasing n .

[Table 1]

[Figure 5]

[Figure 6]

We transcribed the Purdom - Moore algorithm into Algol W and ran it and the sophisticated version of our algorithm on another series of program flow graphs. Table 2 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 non-competitive as n increased. The trade-off point was about $n = 10$.

[Table 2]

[Figure 7]

We implemented the bit vector algorithm using a set of procedures for manipulating multi-precision bit vectors. (Algol W allows bit vectors only of length 32 or less.) Table 3 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 depending upon the number of passes required, but it was always slower than the fast algorithm.

[Table 3]

[Figure 8]

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% speed-up. The corresponding change in the fast algorithm, inserting LINK and EVAL in-line, produced a 20% speed-up. 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 Table 1, Table 4, and Figure 9.

[Table 4]

[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 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 don't 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 values of n. Second, the dominator tree, not the dominator relation, is required for many kinds of global flow analysis [8,14], 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, competitive in speed on small graphs, and 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 non-recursive procedures which use explicit stacks. One can avoid using an auxiliary stack for COMPRESS by instead using a trick of reversing <u>ancestor</u> 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 <u>parent</u> and <u>ancestor</u>. 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 $p = (v = v_0, v_1, \ldots, v_\ell = w)$ is a path from v to w, the path p_1 followed by p_2 is $p = (u = u_0, u_1, \ldots, u_k = v = v_0, v_1, \ldots, v_\ell = w)$.

A flow graph $G = (V,E,r)$ 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 flow graph is a flow graph such that each vertex has exactly two successors. A (directed, rooted) tree $T = (V,E,r)$ is a flow graph 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 <u>spanning tree</u> 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 $V = \{v \mid 1 \le v \le n\}$.

```
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
             v := vertex(i);
step2:  for each u ∈ pred(v) do
             x := EVAL(u); if semi(x) < semi(v) then semi(v) := semi(x) od;
        LINK(parent(v),v);
        add v to bucket(vertex(semi(v)));
step3:  for each w ∈ bucket(parent(v)) do
             delete w from bucket(parent(v));
             u := EVAL(w);
             dom(w) := if semi(u) < semi(parent(v)) then u
                             else parent(v) fi od od;
step4:  i := 2 until n do
             v := vertex(i);
             if dom(v) ≠ vertex(semi(v)) then dom(v) := dom(dom(v)) od

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] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," Math. Ann. 99 (1928), 118-133.

[2] A. V. Aho and J. D. Ullman, The Theory of Parsing, Translation, and Compiling: Volume II: Compiling, Prentice-Hall, Englewood Cliffs, N.J. (1972).

[3] A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass. (1977).

[4] M. S. Hecht and J. D. Ullman, "A simple algorithm for global data flow analysis problems," SIAM J. Comput. 4 (1973), 519-532.

[5] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

[6] E. S. Lorry and C. W. Medlock, "Object code optimization," Communications ACM 12 (1969), 13-22.

[7] P. W. Purdom and E. F. Moore, "Algorithm 430: immediate predominators in a directed graph," Communications ACM 15 (1972), 777-778.

[8] J. Reif, "Combinatorial aspects of symbolic program analysis," TR-11-77, Center for Research in Computing Technology, Harvard University (1977).

[9] R. E. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Computing 1 (1972), 146-160.

[10] R. Tarjan, "Finding dominators in directed graphs," SIAM J. Computing 3 (1974), 62-89.

[11] R. E. Tarjan, "Edge-disjoint spanning trees, dominators, and depth-first search," Technical Report STAN-CS-74-455, Computer Science Department, Stanford University (1974).

[12] R. E. Tarjan, "Applications of path compression on balanced trees," Technical Report STAN-CS-75-512, Computer Science Department, Stanford University (1975).

[13] R. E. Tarjan, "Applications of path compression on balanced trees," Journal ACM, submitted.

[14] R. E. Tarjan, "Solving path problems on directed graphs," Technical Report STAN-CS-528, Computer Science Department, Stanford University (1975).

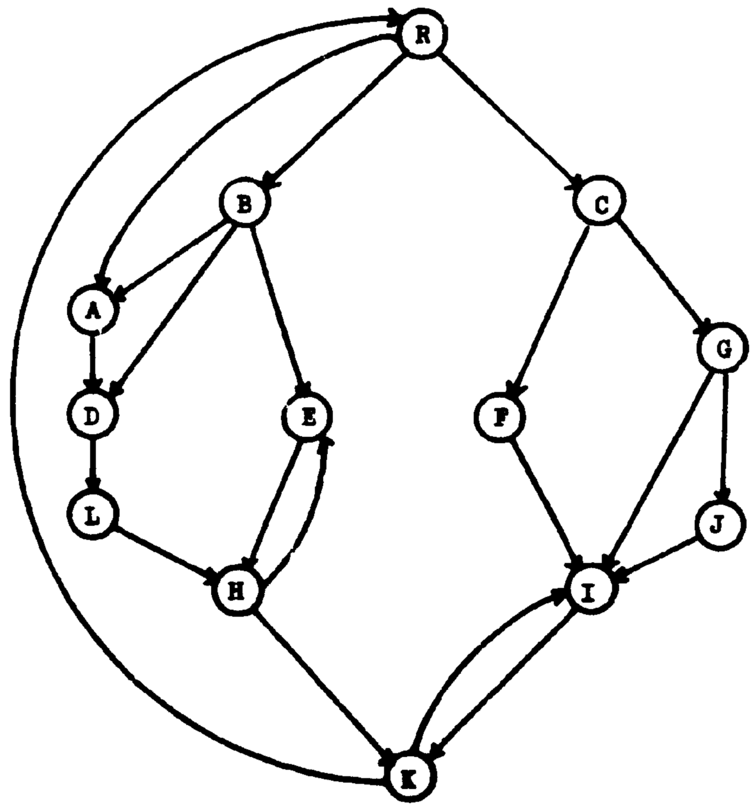

Figure 1. A flow graph.

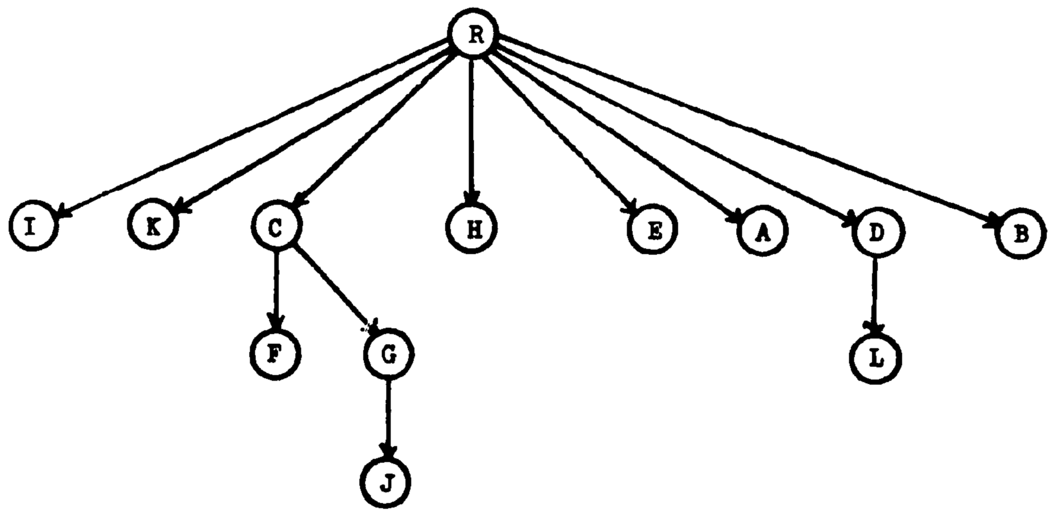

Figure 2. Dominator tree of flow graph in Figure 1.

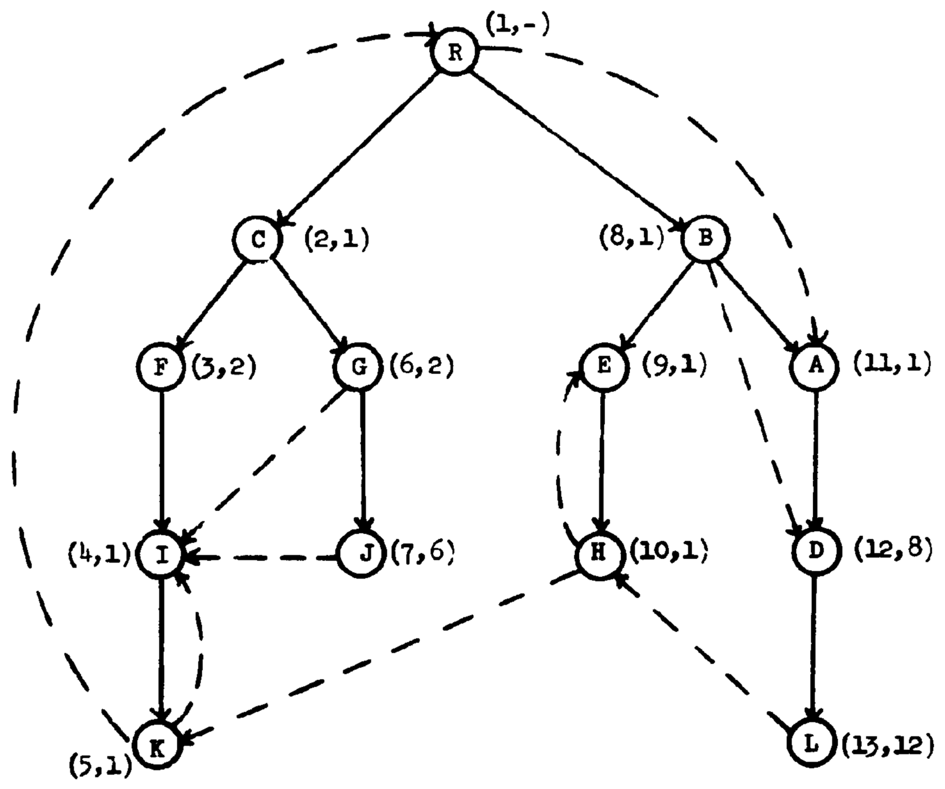

Figure 3. Depth-first search of flow graph in Figure 1.
Solid edges are spanning tree edges, dashed edges are non-tree edges.
First number in parentheses is vertex number, second is semi-dominator.

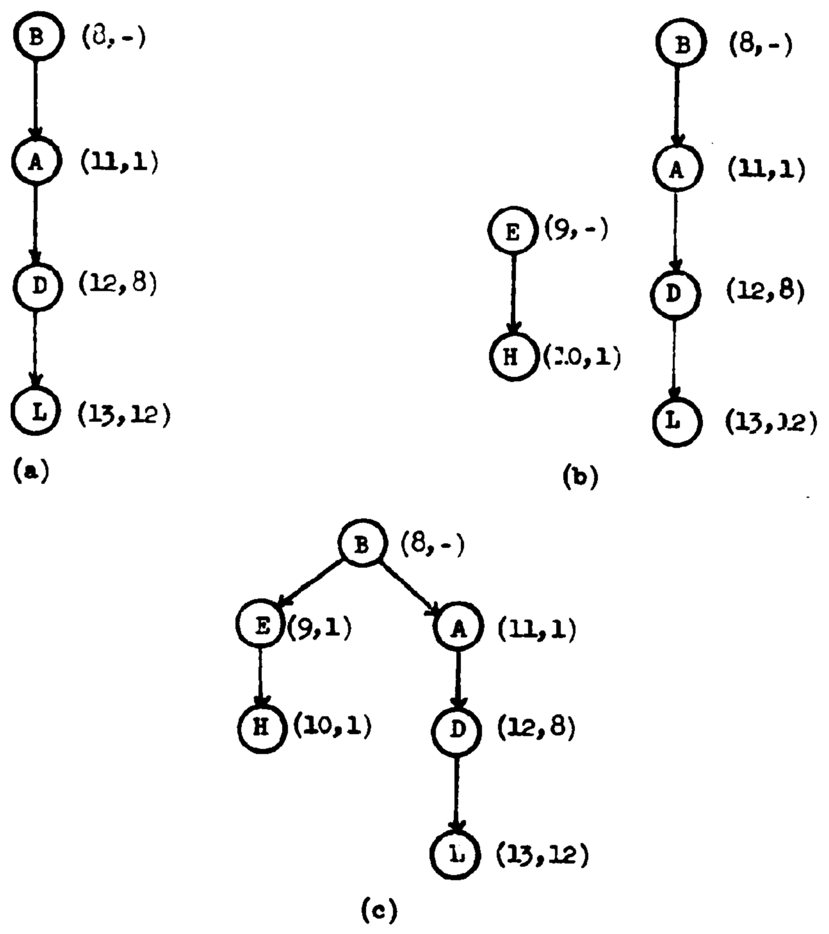

Figure 4. Forest maintained by LINK and EVAL during steps 2 and 3 of the dominators algorithm. (Trees in the forest consisting of single vertices are not shown.)

(a) Before vertex H is processed. Candidates for $\underline{\text{sdom}}(H)$ are $9 = \underline{\text{number}}(E)$ and $1 = \min\{\underline{\text{sdom}}(v) \mid B \overset{+}{\rightarrow} v \overset{*}{\rightarrow} L\}$.

(b) Before vertex E is processed. Candidates for $\underline{\text{sdom}}(E)$ are $8 = \underline{\text{number}}(B)$ and $1 = \min\{\underline{\text{sdom}}(v) \mid E \overset{+}{\rightarrow} v \overset{*}{\rightarrow} H\}$. After $\underline{\text{sdom}}(E) = 1$ is computed, $\underline{\text{bucket}}(B)$ is unloaded. At this time D is the only element of $\underline{\text{bucket}}(B)$. A is the vertex such that $\underline{\text{sdom}}(A) = \min\{\underline{\text{sdom}}(v) \mid B \overset{+}{\rightarrow} A \overset{*}{\rightarrow} D\}$. Since $\underline{\text{sdom}}(A) = 1 < \underline{\text{sdom}}(D) = 8$, $\underline{\text{dom}}(D)$ is assigned $\underline{\text{dom}}(D) := A$. Note that $\underline{\text{idom}}(D) = \underline{\text{idom}}(A) = R$.

(c) After E is processed.

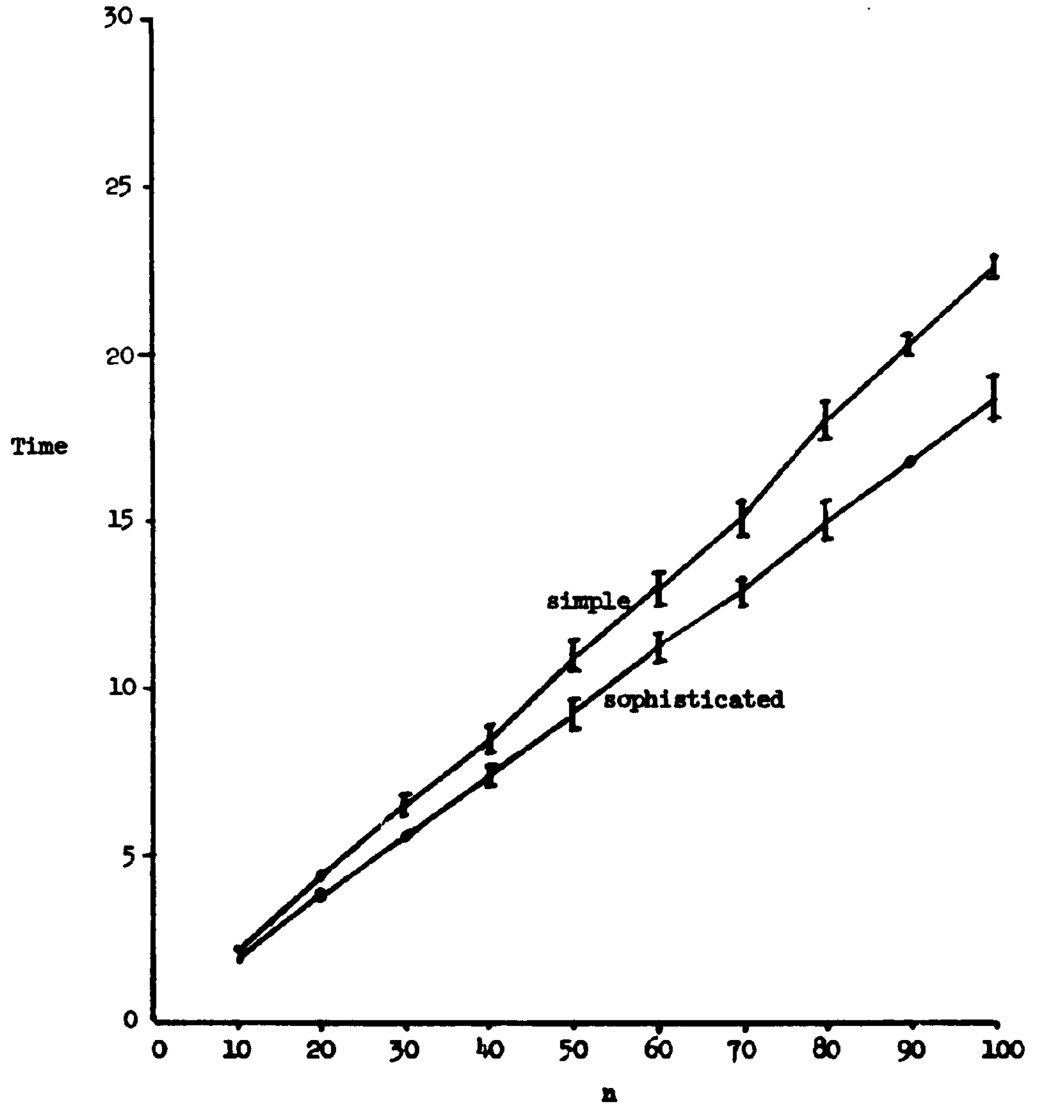

Figure 5. Running times in 10^{-3} seconds of the simple and sophisticated versions of the fast algorithm.

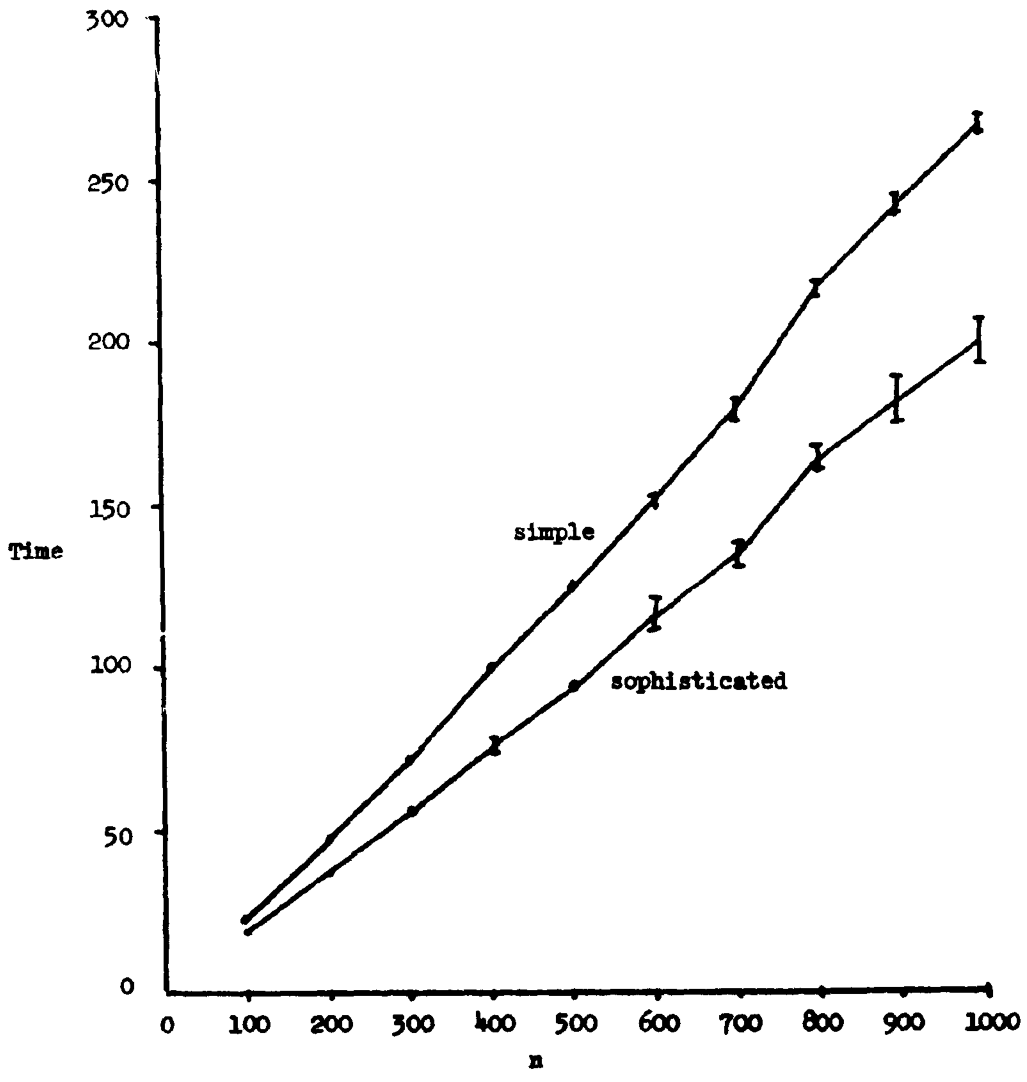

Figure 6. Running times in 10^{-3} seconds of the simple and sophisticated versions of the fast algorithm.

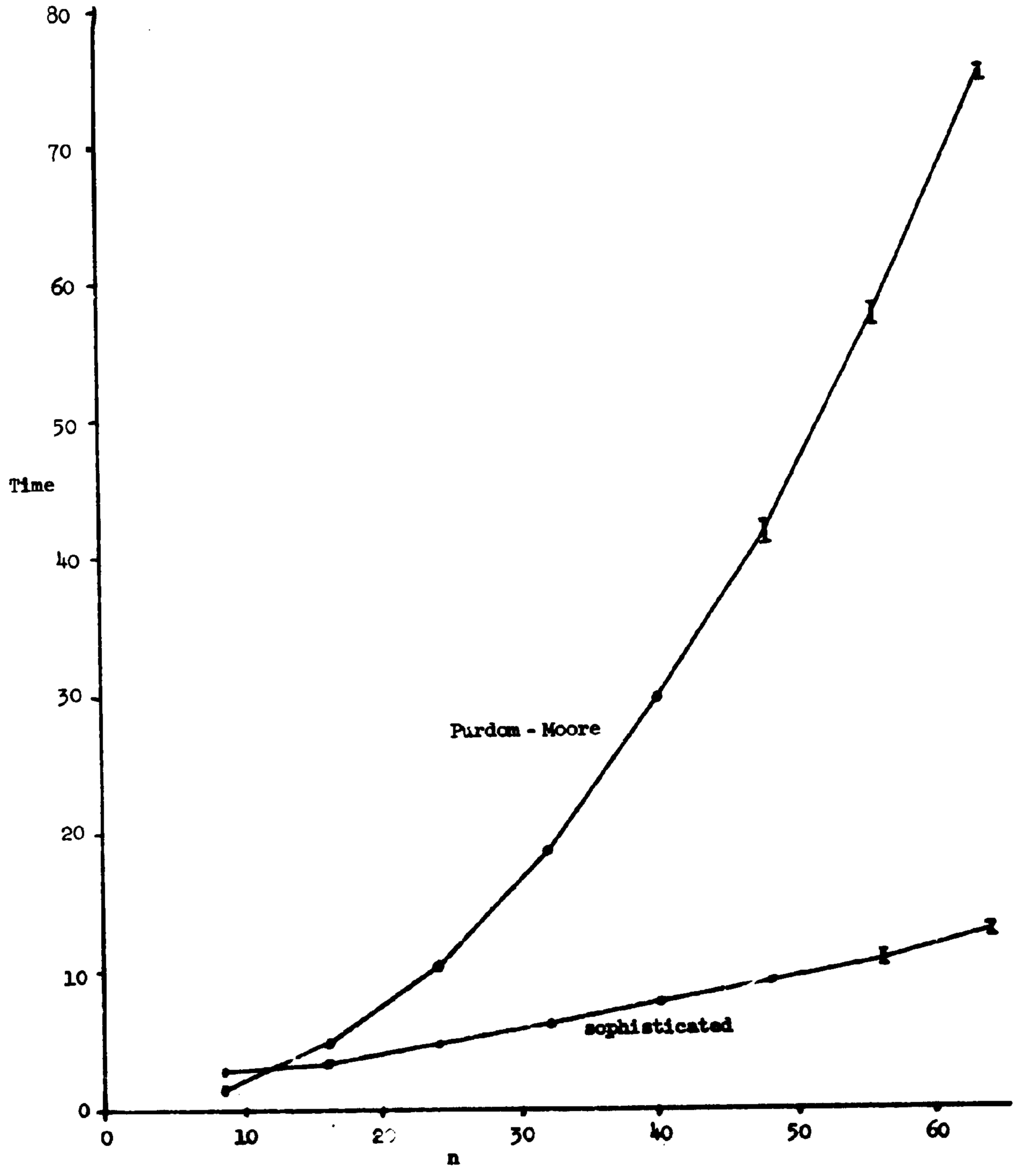

Figure 7. Running times in 10^{-3} seconds of the Purdom - Moore algorithm and the sophisticated version of the fast algorithm.

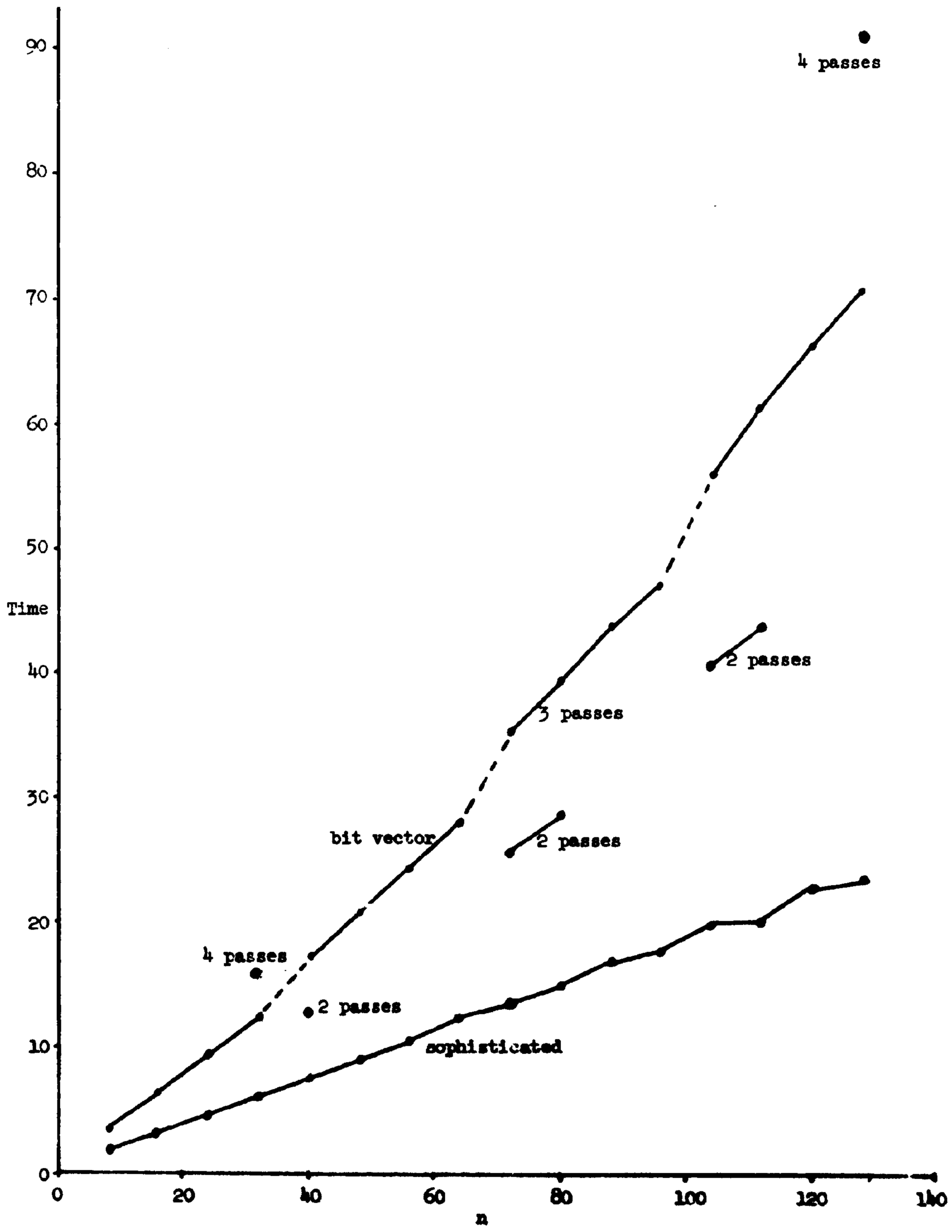

Figure 8. Running times in 10^{-3} seconds of the bit vector algorithm and the sophisticated version of the fast algorithm.

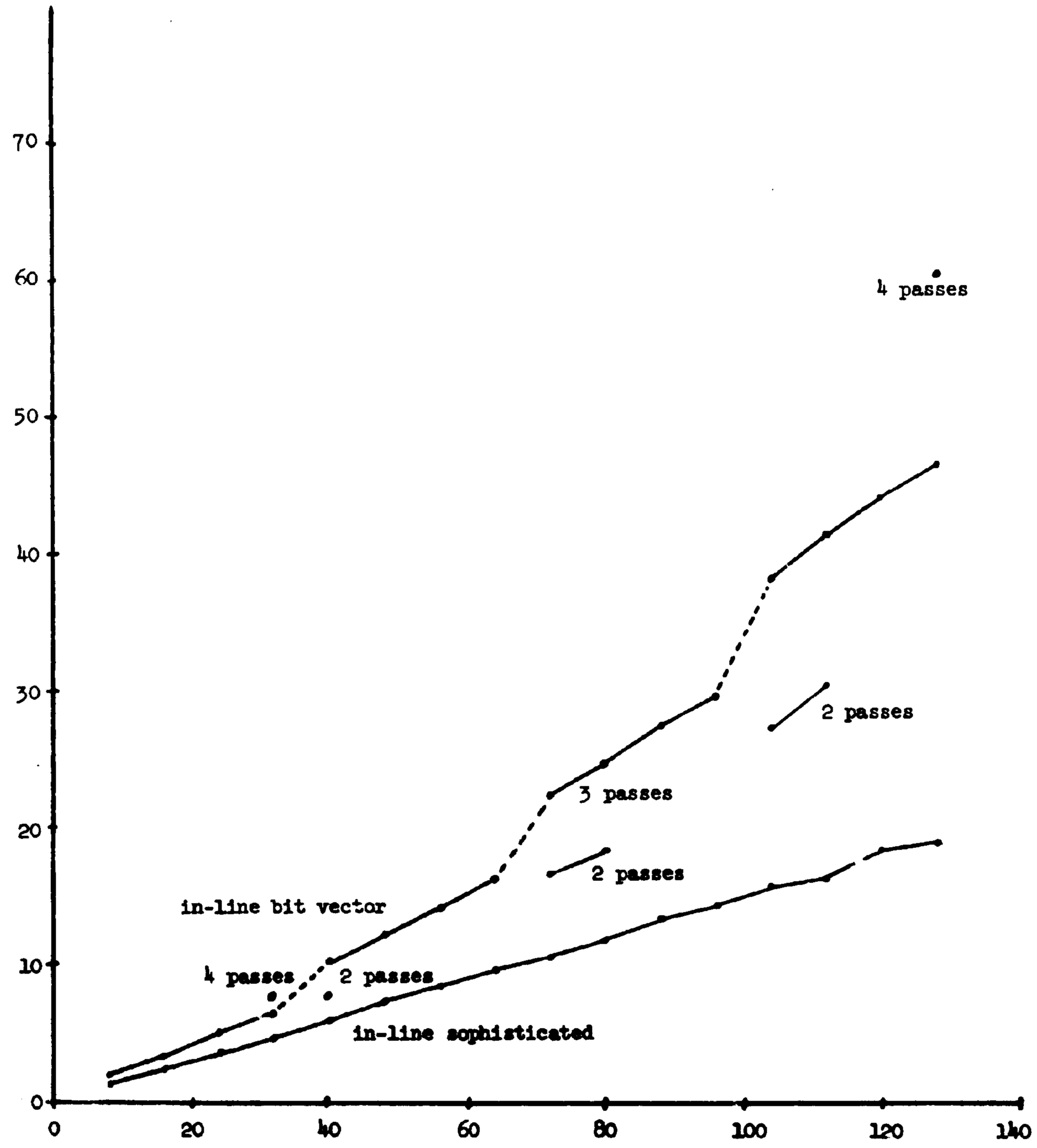

Figure 9. Running times in 10^{-3} seconds of the in-line bit vector algorithm and the in-line sophisticated version of the fast algorithm.

n	simple		sophisticated	
	min	max	min	max
10	2.0	2.1	1.9	2.0
20	4.3	4.4	3.7	3.9
30	6.2	6.8	5.5	5.8
40	8.0	8.8	7.1	7.6
50	10.5	11.4	8.9	9.6
60	12.4	13.4	10.9	11.6
70	14.6	15.4	12.6	13.1
80	17.4	18.6	14.5	15.6
90	20.0	20.2	16.7	16.8
100	22.4	22.7	18.0	19.3

n	simple		sophisticated	
	min	max	min	max
200	46.4	47.2	36.2	36.4
300	70.1	72.3	55.0	55.7
400	98.5	101	74.7	78.1
500	123	125	92.0	93.7
600	150	152	110	120
700	176	181	130	137
800	214	217	158	167
900	238	244	173	188
1000	263	268	192	206

Table 1. Running times in 10^{-3} seconds of the simple and sophisticated versions of the fast algorithm (three graphs for each value of n).

n	sophisticated		in-line sophisticated		Purdom - Moore	
	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		

Table 2. **Running times in 10^{-3} seconds of the Purdom - Moore algorithm and the sophisticated version of the fast 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

Table 3. Running times in 10^{-3} seconds and number of passes of the 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

Table 4. **Running times in 10^{-3} seconds and number of passes of the in-line bit vector algorithm (three graphs for each value of n).**