FINDING DOMINATORS IN DIRECTED GRAPHS*

ROBERT TARJAN†

Abstract. This paper describes an algorithm for finding dominators in an arbitrary directed graph. The algorithm uses depth-first search and efficient algorithms for computing disjoint set unions and manipulating priority queues to achieve a time bound of \( O(V \log V + E) \) if \( V \) is the number of vertices and \( E \) is the number of edges in the graph. This bound compares favorably with the \( O(V(V + E)) \) time bound of previously known algorithms for finding dominators in arbitrary directed graphs, and with the \( O(V + E \log E) \) time bound of a known algorithm for finding dominators in reducible graphs. If \( E \geq V \log V \), the new algorithm requires \( O(E) \) time and is optimal to within a constant factor.

Key words. algorithm, binary tree, complexity, connectivity, depth-first search, directed graph, dominator, equivalence algorithm, graph, immediate dominator, priority queue, search, set union, stack, topological sorting, tree

1. Introduction. The following graph-theoretic problem arises when one is attempting to optimize computer code [1], [2]: suppose \( G \) is a directed graph with a start vertex \( s \). (\( G \) might represent the flow between blocks of a computer program; vertex \( s \) then represents the initial block of the program.) If vertex \( d \) lies on every path from vertex \( s \) to vertex \( i \), then \( d \) is called a dominator of \( i \). If \( d \) is a dominator of \( i \) and every other dominator \( d' \) of \( i \) also dominates \( d \), then \( d \) is called an immediate dominator of \( i \). It is easy to prove that each vertex has a unique immediate dominator if it has any dominators [1], [2]. We wish to find the immediate dominator of each vertex in the graph.

The dominators problem is relatively new and has not been studied extensively. Aho and Ullman’s algorithm [1] for finding dominators deletes each vertex \( v \) in turn from \( G \) and tests which vertices are reachable from \( s \). Any reachable vertex is not dominated by \( v \). This algorithm requires \( O(V(V + E)) \) time if the problem graph has \( V \) vertices and \( E \) edges. Purdom and Moore’s algorithm [3] has the same time bound; no previously published algorithm is faster in general. See [2], [4], [5] for other algorithms. Aho, Hopcroft and Ullman [6] have constructed an \( O(V + E \log E) \) algorithm for finding dominators in a restricted class of graphs called reducible graphs [7], [8], [9]. Their algorithm is based on an efficient method for finding least common ancestors in trees.

This paper describes the use of depth-first search [10] to reveal the structure of directed graphs. Using efficient algorithms for computing disjoint set unions [11], [12], [13] and for manipulating priority queues [14], [15], we may calculate dominators from the search information. The resultant dominators algorithm

---

* Received by the editors April 26, 1973. This research was supported in part by the National Science Foundation under Grant GJ-33170X while the author was at Stanford University.
† Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, California 94720.

The graph-theoretic definitions used in this paper are more or less standard. For those unfamiliar with graph theory, these definitions appear in Appendix A, along with a definition of the big "O" notation.
has an $O(V + E)$ space bound and an $O(V \log V + E)$ time bound. The method is optimal to within a constant factor if $E \geq V \log V$.

The paper is divided into several sections. Section 2 describes depth-first search and its application to directed graphs. Section 3 describes four dominator-preserving graph transformations which use search information and which form the heart of the dominators algorithm. Section 4 outlines the algorithm. Sections 5 and 6 give the details of some of the necessary calculations, and § 7 presents the complete algorithm. Section 8 gives an even faster algorithm for finding dominators in certain special graphs, suggesting that a faster algorithm may exist in general. Section 9 gives conclusions.

2. Depth-first search. We wish to calculate $\text{IDOM}(v)$, the immediate dominator of $v$, for each vertex $v$ in a directed graph $G$ with $V$ vertices and $E$ edges. Figure 1 shows a graph for which we might wish to solve this problem. We begin by exploring $G$ starting at vertex $s$ and marking all vertices reached. Vertex $s$ and vertices remaining unmarked have no dominators, while all other vertices have dominators. The problem is then reduced to finding dominators in the subgraph $G_1$ whose vertices are all those reachable from $s$. In $G_1$ each vertex has a dominator. Furthermore, we have the following lemma.

**Lemma 1** [1], [2]. We may construct a tree (called the dominator tree of $G_1$) whose vertices are those of $G_1$ and such that $w$ is a son of $v$ in the tree if and only if $v$ is the immediate dominator of $w$. The ancestors of $w$ in the tree are precisely the dominators of $w$. (Figure 2 shows the dominator tree of the graph in Fig. 1.)
To mark the vertices reachable from $s$, we carry out a depth-first search of $G$ [10]. That is, we start at vertex $s$ and choose an edge leading from $s$ to explore. Traversing the edge leads to a vertex, either new or already reached. In general, we continue the search by selecting and traversing an unexplored edge from the most recently reached vertex which still has unexplored edges. Eventually each edge will be traversed exactly once. To implement such a search, we use a set of adjacency lists $A(v)$, one for each vertex $v$ in the graph; if $(v, w)$ is an edge of $G$, then $w$ appears in the adjacency list $A(v)$. Each edge is represented exactly once. Here is a recursive procedure for carrying out a depth-first search:

```plaintext
procedure DFS(v);
begin comment v is the most recently reached vertex;
    MARK(v) := true;
    for w e A(v) do
        if not MARK(v) then DFS(w);
end;
```

The following statements will then mark every vertex reachable from $s$, by applying DFS:

```plaintext
comment mark all vertices reachable from s;
for each vertex $v$ do MARK(v) := false;
DFS(s);
```

A depth-first search yields much more information than just which vertices are reachable from the start vertex of the search. In particular, it gives enough information about the connectivity structure of the graph to efficiently determine dominators. Let us add a few more calculations to the search. (Henceforth for convenience we shall assume that all vertices in $G$ are reachable from $s$.)

DFS is a recursive procedure; the successively reached new vertices are input parameters to DFS and thus are stored on a stack (in any implementation of DFS).
This stack contains all vertices reached which may still have unexplored edges, and the vertices as they appear in order on the stack determine a path in $G$ from $s$ to the current vertex being examined during the search. Suppose we keep track of which vertices are stacked at any given time, and that we number the vertices from 1 to $V$ in the order they are reached during the search. Then a depth-first search of a directed graph partitions the edges traversed into four classes:

1. Edges $(v, w)$ with $w$ unmarked when $(v, w)$ is explored, called tree arcs.
2. Edges $(v, w)$ with $w$ stacked when $(v, w)$ is explored, called fronds.
3. Edges $(v, w)$ with $\text{NUMBER}(v) < \text{NUMBER}(w)$ and $w$ unstacked when $(v, w)$ is explored, called reverse fronds.
4. Edges $(v, w)$ with $\text{NUMBER}(v) > \text{NUMBER}(w)$ and $w$ unstacked when $(v, w)$ is explored, called cross-links.

Lemma 2 below gives the properties of edges in these four classes. In particular, the tree arcs determine a spanning tree $T$ of $G$ which has root $s$. One more numbering scheme based on depth-first search is useful. Let the vertices of $G$ be numbered from $V$ to 1 as they are unstacked during the search. We shall denote this numbering by $\text{SNUMBER}(v)$. Lemma 3 below gives properties of $\text{SNUMBER}$'s. Here is an elaborated version of the depth-first search procedure which computes both types of number for each vertex, divides edges into their classes, and also counts the number of descendants $\text{ND}(v)$ of each vertex $v$ in the spanning tree $T$.

```
procedure CLASSIFY(G, s);
begin comment the edge-classifying procedure uses the following elaborated version of DFS. Variable $m$ denotes the last NUMBER assigned to any vertex. Variable $n$ denotes the last SNUMBER assigned to any vertex. The procedure assumes that $G$ is represented as a set of adjacency lists $A(v)$. NUMBER$(v) = 0$ if and only if $v$ has not been reached. SNUMBER$(v) = 0$ if and only if $v$ has not yet been unstacked;
procedure DFSEARCH(v);
begin
  $m := \text{NUMBER}(v) := m + 1$;
  $\text{ND}(v) := 1$;
  for $w \in A(v)$ do
    if $\text{NUMBER}(w) = 0$ then
      begin
        label $(v, w)$ a tree arc;
        DFSEARCH(w);
        $\text{ND}(v) := \text{ND}(v) + \text{ND}(w)$;
      end
    else if $\text{SNUMBER}(w) = 0$ then
      begin comment vertex $w$ is stacked;
        label $(v, w)$ a frond;
      end
    else if $\text{NUMBER}(v) < \text{NUMBER}(w)$ then
      label $(v, w)$ a reverse frond
    else label $(v, w)$ a cross-link;
end
```
Figure 3 illustrates what CLASSIFY does to the graph in Fig. 1. It should be clear that this elaborated version of depth-first search correctly numbers the vertices and classifies the edges. Reference [10] contains a proof that these calculations require $O(V + E)$ time and space. Lemmas 2–5, given without proof, state basic properties of the numbers calculated by CLASSIFY.

**Lemma 2.** Suppose that all vertices of a directed graph $G$ are reachable from vertex $s$, and that the edges of $G$ are divided into classes using CLASSIFY($G, s$). Then:

(i) The tree arcs form a directed tree $T$ with root $s$ which contains all vertices in $G$. We shall denote the existence of a tree arc $(v, w)$ by $v \rightarrow w$, and the existence of a path from $v$ to $w$ in $T$ by $v \Rightarrow w$.

(ii) If $(v, w)$ is a frond, then $\text{NUMBER}(w) < \text{NUMBER}(v)$, and $w \Rightarrow v$ in $T$.

(iii) If $(v, w)$ is a reverse frond, then $v \Rightarrow w$ in $T$.

(iv) If $(v, w)$ is a cross-link, then neither $v \Rightarrow w$ nor $w \Rightarrow v$ in $T$. 

**Fig. 3.** Depth-first search applied to graph in Fig. 1. Tree arcs are labeled $T$, fronds $F$, reverse fronds $R$, and cross-links $C$. Numbering vertices from 1 to $V$ as vertices are reached during the search gives the first number at each vertex. Numbering the vertices from $V$ to 1 as vertices are unstacked during the search gives the second number at each vertex.
LEMMA 3. If \((v, w)\) is a tree arc, a reverse frond, or a cross-link, \(\text{SNUMBER}(v) < \text{SNUMBER}(w)\). If \((v, w)\) is a frond, \(\text{SNUMBER}(v) > \text{SNUMBER}(w)\).

LEMMA 4. Let \(v\) be a vertex in \(G\). Then the number of descendants of \(v\) in the spanning tree \(T\) is given by:

\[
\text{ND}(v) = 1 + \sum_{w = v} \text{ND}(w).
\]

LEMMA 5. Statements (i), (ii), (iii) and (iv) below are equivalent.

(i) \(v \Rightarrow w\) in \(T\).
(ii) \(\text{NUMBER}(v) \leq \text{NUMBER}(w) < \text{NUMBER}(v) + \text{ND}(v)\).
(iii) \(\text{SNUMBER}(v) \leq \text{SNUMBER}(w) < \text{SNUMBER}(v) + \text{ND}(v)\).
(iv) \(\text{NUMBER}(v) \leq \text{NUMBER}(w)\) and \(\text{SNUMBER}(v) \leq \text{SNUMBER}(w)\).

Lemma 5 gives us three methods for identifying the descendants of a vertex and allows us to dynamically identify fronds, reverse fronds, and cross-links if we so desire.

LEMMA 6. \(G\) is acyclic if and only if \(G\) has no fronds.

Proof. If \(G\) has a frond \((v, w)\), then the frond and the set of tree arcs joining \(v\) and \(w\) form a cycle. If \(G\) has no fronds, all edges \((v, w)\) satisfy \(\text{SNUMBER}(v) < \text{SNUMBER}(w)\). Since any cycle in \(G\) must have at least one arc \((v, w)\) with \(\text{SNUMBER}(v) > \text{SNUMBER}(w)\), \(G\) has no cycles.

COROLLARY 7. If \(G\) is an acyclic directed graph, \(\text{CLASSIFY}\) assigns \(\text{SNUMBER}'s\) to \(G\) so that if \((v, w)\) is an edge, \(\text{SNUMBER}(v) < \text{SNUMBER}(w)\). Thus \(\text{CLASSIFY}\) gives an \(O(V + E)\) algorithm for “topologically sorting” the edges of \(G\). (See Knuth \[16\] for further discussion of this problem.)

LEMMA 8. Let \(p\) be a path from \(v\) to \(w\) in \(G\). Let vertices be identified by their number. Suppose \(v < w\). Then \(p\) contains some common ancestor of \(v\) and \(w\) in \(T\).

Proof. Let \(T_u\) with root \(u\) be the smallest subtree of \(T\) containing all vertices on the path \(p\). We prove that \(p\) passes through \(u\). If \(u = v\) or \(u = w\), the result is immediate. Otherwise, let \(u_1 < u_2 < \cdots < u_n\) be the sons of \(u\) such that for each \(u_i\), some descendant of \(u_i\) is on \(p\). For any \(i\), let \(T_{u_i}\) be the subtree of \(T\) with root \(u_i\). If \(n = 1\), \(p\) must pass through \(u\) since \(p\) is minimal. If \(n > 1\), there must be some \(u_i, u_j, i < j\), such that \(p\) leads from \(T_{u_i}\) to \(T_{u_j}\). This is true since \(v < w\) and all the vertices in \(T_{u_i}\) are numbered smaller than all the vertices in \(T_{u_j}\) if \(i < j\). But \(p\) can only get from \(T_{u_i}\) to \(T_{u_j}\) by passing through \(u\), since the only edges leading from lower numbered vertices to higher numbered ones are tree arcs and reverse fronds. The lemma follows.

The properties of depth-first search presented above may be used to construct a good algorithm to solve the dominators problem. One way to find dominators is to convert \(G\) into an equivalent acyclic graph, by deleting each frond and replacing it by an equivalent set of reverse fronds and cross-links to preserve dominators. In the resultant acyclic graph, the dominators may be found for the vertices in \(\text{SNUMBER}\) order from 1 to \(V\), since any path leads through vertices with increasing \(\text{SNUMBER}\). This algorithm has an \(O(V^2)\) time bound \[17\]; the time bound is not linear in the number of edges because the number of added reverse fronds and cross-links may be large. To get a faster algorithm, we must be a little more clever.
The idea we use is to convert $G$ into a graph with no fronds and no cross-links by adding suitable reverse fronds. We use four simplifying transformations which preserve dominators to accomplish this. First we delete all fronds and replace them with a simpler set of fronds, at most one leaving each vertex. Then we convert cross-links to reverse fronds, we combine fronds and reverse fronds to give new reverse fronds, and we delete all but one reverse frond entering each vertex. The last three transformations must be carried out simultaneously with the dominator calculations. The computation proceeds through the vertices in NUMBER order, from $V$ to 1. Section 3 describes the four transformations and proves that they preserve dominators.

3. Dominator-preserving transformations. Suppose that a depth-first search of a directed graph $G$ is carried out using CLASSIFY($G, s$), and that all vertices of $G$ are reachable from the start vertex $s$. For any vertex $v$, let $F(v) = \{w | w \neq v$ and $\exists u(w \rightarrow v \rightarrow u$ in $T$ and $(u, w)$ is a frond of $G)\}$. Let $\text{HIGHPT}(v)$ be the highest numbered vertex in $F(v)$ if $F(v)$ is nonempty. (Here and henceforth we shall only use one numbering of vertices, NUMBER as calculated by CLASSIFY.) Since each element in $F(v)$ is an ancestor of $v$ in $T$, it is clear that $w \in F(v)$ implies $w \rightarrow \text{HIGHPT}(v)$. Let $G'$ be the graph formed from $G$ by deleting all fronds and adding a new frond $(v, \text{HIGHPT}(v))$ for each vertex $v$ for which $\text{HIGHPT}(v)$ is defined. This is our first transformation, called frond replacement. Figure 4 shows the graph in Fig. 3 transformed in this way.

**Fig. 4.** Frond replacement applied to the graph in Fig. 3. Vertices are numbered in search order. Vertex K loses a frond; vertices I, F and C gain a frond.

We have the following results.

**Lemma 9.** Let $G'$ be formed from $G$ by frond replacement. If $w \rightarrow v \rightarrow u$ in $T$ (the spanning tree of $G$ and $G'$), and if $(u, w)$ is a frond in $G$, then there is a path from $v$ to $w$ in $G'$ which consists only of fronds.
Proof. The proof is by induction on the length of the tree path from $w$ to $v$. If $w = v$, then the lemma is true since there is a path containing no edges from any vertex to itself. Let the lemma be true whenever the tree path from $w$ to $v$ has length less than $k$, and suppose the tree path from $w$ to $v$ has length $k$. Since $G$ contains a frond $(u, w)$ with $w \rightarrow v \rightarrow u$, $w \in F(v)$ and \textsc{highpt}(v) is defined. Furthermore \textsc{highpt}(v) \neq v$ and $w \rightarrow \textsc{highpt}(v) \rightarrow v$. By the induction hypothesis, there is a path of fronds from \textsc{highpt}(v) to $w$ in $G'$. Adding $(v, \textsc{highpt}(v))$ to the front of this path gives a path in $G'$ from $v$ to $w$ which consists only of fronds. By induction the lemma is true.

Lemma 10. Vertex $d$ dominates vertex $v$ in $G$ if and only if $d$ dominates $v$ in $G'$.

Proof. Suppose $w$ does not dominate $v$ in $G$. Then in $G$ there is a path from $s$ to $v$ which does not contain $w$. Suppose this path contains a frond $(u, u')$ with $u' \rightarrow w$. Then we may replace the part of $p$ up to and including the last such frond by a path of tree arcs. This gives us a path in $G$ from $s$ to $v$ which contains neither $w$ nor any frond $(u, u')$ such that $u' \rightarrow w$. If we now replace each frond in the path by the corresponding path of fronds in $G'$ guaranteed by Lemma 9, we get a path $p'$ in $G'$ from $s$ to $v$ which doesn't contain $w$, and $w$ does not dominate $v$ in $G'$.

Conversely, suppose $w$ does not dominate $v$ in $G'$. Then there is a path $p'$ in $G'$ from $s$ to $v$ which doesn't contain $w$. Suppose this path contains a frond $(u, \textsc{highpt}(u))$ with $\textsc{highpt}(u) \rightarrow w$. Then we may replace the part of $p'$ up to and including the last such frond by a path of tree arcs. This gives us a path $p''$ in $G'$ from $s$ to $v$ which contains neither $w$ nor any frond $(u, \textsc{highpt}(u))$ with $\textsc{highpt}(u) \rightarrow w$. Corresponding to any remaining frond $(u, \textsc{highpt}(u))$ on path $p''$ there is a frond $(u', \textsc{highpt}(u))$ in $G$ with $u \rightarrow u'$. If we replace each frond $(u, \textsc{highpt}(u))$ in $p''$ by a path of tree arcs from $u$ to $u'$ followed by the frond $(u', \textsc{highpt}(u))$, we get a path $p$ in $G$ from $s$ to $v$ which doesn't contain $w$. It follows that $w$ does not dominate $v$ in $G$, and the lemma is true.

To calculate dominators in $G$, we apply frond replacement and calculate dominators in the transformed graph $G'$. Observe that if frond replacement is applied to $G'$, the result is $G'$ itself. Henceforth we shall assume that $G$ is a graph which has been explored using \textsc{classify} and whose fronds have been replaced as specified above. We shall identify vertices using the \textsc{number} assigned to them by \textsc{classify}.

Lemma 11. Let $(u, v)$ and $(u_1, v)$ be two reverse fronds in $G$, with $u_1 > u$. Let $G'$ be the graph formed from $G$ by deleting edge $(u_1, v)$. We call this transformation "reverse frond deletion". Then $d$ dominates $v$ in $G$ if and only if $d$ dominates $v$ in $G'$.

Proof. Since $G'$ is a subgraph of $G$, every path in $G'$ is a path in $G$. Thus if $d$ dominates $v$ in $G$, $d$ dominates $v$ in $G'$. Conversely, suppose $w$ does not dominate $v$ in $G$. Then there is a path $p$ in $G$ from $s$ to $v$ which doesn't contain $w$. If $p$ doesn't contain $(u_1, v)$, then $p$ is a path in $G'$ and $w$ doesn't dominate $v$ in $G'$. Suppose $p$ contains $(u_1, v)$. If $w$ is a descendant of $u_1$, we may replace the part of $p$ up to and including edge $(u_1, v)$ by the path of tree arcs from $s$ to $u$ followed by the frond $(u, v)$ to get a path $p'$ in $G'$ from $s$ to $v$ which doesn't contain $w$. If $w$ is not a descendant of $u_1$, we may replace $(u_1, v)$ in $p'$ by the path of tree arcs from $u_1$ to $v$ and get a path $p''$ in $G'$ from $s$ to $v$ which doesn't contain $w$. In no case can $w$ dominate $v$ in $G'$, and the lemma is true.
Lemma 12. Let \( v \) be a vertex in \( G \) such that one frond \((v, \text{HIGHPT}(v))\) leaves \( v \), at most one reverse frond (say \((u, v)\)) enters \( v \), and no cross-links or fronds enter \( v \). Let \( G' \) be the graph formed from \( G \) by deleting frond \((v, \text{HIGHPT}(v))\) and adding \((u, \text{HIGHPT}(v))\) if \((u, v)\) is defined and \((u, \text{HIGHPT}(v))\) is a reverse frond (i.e., \( u < \text{HIGHPT}(v) \)). We call this transformation "frond deletion". Then \( d \) dominates \( w \) in \( G \) if and only if \( d \) dominates \( w \) in \( G' \), assuming no frond deletions have been applied to vertices \( x < v \).

Proof. Suppose \( x \) does not dominate \( w \) in \( G \). Then there is a path \( p \) in \( G \) from \( s \) to \( w \) which doesn't contain \( x \). If \( p \) doesn't contain \((v, \text{HIGHPT}(v))\), then \( p \) is a path in \( G' \). If \( p \) contains \((v, \text{HIGHPT}(v))\), then \( p \) must contain either \((u, v)\) or the tree arc entering \( v \). If \( x \) is not a proper ancestor of \( \text{HIGHPT}(v) \), then we can replace the part of \( p \) up to and including \((v, \text{HIGHPT}(v))\) by the path of tree arcs from \( s \) to \( \text{HIGHPT}(v) \) and have a path in \( G' \) which doesn't contain \( x \). Suppose \( x \) is a proper ancestor of \( \text{HIGHPT}(v) \). If the edge before \((v, \text{HIGHPT}(v))\) in \( p \) is \((u, v)\) and \( u < \text{HIGHPT}(v) \), then we may replace \((u, v)\) and \((v, \text{HIGHPT}(v))\) by \((u, \text{HIGHPT}(v))\) and have a path in \( G' \) which doesn't contain \( x \). If the edge before \((v, \text{HIGHPT}(v))\) is a tree arc, or if it is \((u, v)\) and \( u \geq \text{HIGHPT}(v) \), then by Lemma 9 we may replace \((v, \text{HIGHPT}(v))\) and the edge before it by a path of fronds and have a path in \( G' \) which doesn't contain \( x \). In any case \( x \) doesn't dominate \( w \) in \( G' \).

Conversely, suppose \( x \) does not dominate \( w \) in \( G' \). Then there is a path \( p' \) in \( G' \) from \( s \) to \( w \) which doesn't contain \( x \). If \( p' \) doesn't contain \((u, \text{HIGHPT}(v))\), then \( p' \) is a path in \( G \). Suppose \( p' \) contains \((u, \text{HIGHPT}(v))\). If \( x \neq v \), we may replace \((u, \text{HIGHPT}(v))\) in \( p' \) by \((u, v)\) and \((v, \text{HIGHPT}(v))\) to give a path in \( G \) which doesn't contain \( x \). If \( x = v \), we may replace the part of \( p' \) up to and including \((u, \text{HIGHPT}(v))\) by the path of tree arcs from \( s \) to \( \text{HIGHPT}(v) \) and have a path in \( G \) which doesn't contain \( w \). In no case does \( x \) dominate \( w \) in \( G \), and the lemma is true.

The dominators algorithm works in the following way: first we apply frond replacement to \( G \). Next, we process the vertices from \( V \) to 1. To process a vertex \( v \), we convert all incoming cross-links to reverse fronds (by a transformation yet to be described), we eliminate all but one reverse frond entering \( v \) by applying reverse frond deletion, and we eliminate the frond (if any) leaving \( v \) by applying frond deletion. We are left with at most one tree arc and one reverse frond entering \( v \). We then update the partially calculated dominators and proceed to the next vertex.

In order to understand the transformation of cross-links into reverse fronds, we must examine in detail the way dominators are calculated. Let \( G \) be a graph which has been explored using CLASSIFY and whose fronds have been replaced. Let \( G(i) \) be the subgraph of \( G \) which contains all the tree arcs in \( G \) plus all edges leading to vertices \( v \) such that \( \text{NUMBER}(v) > i \). Let the \( i \)-th semidominator of vertex \( v \) ( abbreviated \( \text{SDOM}(i, v) \)) be the immediate dominator of \( v \) in \( G(i) \). In all that follows we shall assume that \( v \neq 1 \) (that is, \( v \) is not the start vertex) so all semidominators are defined. The dominators algorithm calculates \( \text{SDOM}(i - 1, v) \) for all vertices when vertex \( i \) is processed. The \( \text{SDOM} \) values tell us how to convert cross-links into reverse fronds as well as giving dominators \( \text{SDOM}(0, v) = \text{IDOM}(v) \). The lemmas below describe semidominators. By combining these results with the lemmas above, we can build a dominators algorithm.
LEMMA 13. If \( v \neq 1 \) and \( i \geq v \), \( SDOM(i, v) \) is the father of vertex \( v \) in the tree generated by CLASSIFY.

Proof. \( G(i) \) contains only one edge leading to vertex \( v \); namely, a tree arc. Every path from \( s \) to \( v \) must pass through this edge. The lemma follows.

LEMMA 14. For all \( i \), if \( v \neq 1 \) then \( SDOM(i - 1, v) \Rightarrow SDOM(i, v) \).

Proof. For all \( i \), if \( v \neq 1 \), the only dominators of \( v \) in \( G(i) \) lie on the tree path from \( s \) to \( v \). Furthermore, \( G(i) \) is a subgraph of \( G(i - 1) \), so if \( w \) dominates \( v \) in \( G(i - 1) \), \( w \) must dominate \( v \) in \( G(i) \). The lemma follows.

LEMMA 15. If \( v \neq 1 \) and \( SDOM(i, v) = i \), then \( IDOM(v) = i \).

Proof. By Lemma 14, \( IDOM(v) = SDOM(0, v) \Rightarrow SDOM(i, v) \). Thus if \( SDOM(i, v) \) dominates \( v \) in \( G \), \( SDOM(i, v) = IDOM(v) \). Now we show by contradiction that \( SDOM(i, v) \) dominates \( v \) in \( G \) if \( SDOM(i, v) = i \).

Suppose to the contrary that there is a path \( p \) in \( G \) from \( s \) to \( v \) which does not contain \( i = SDOM(i, v) \). We must have \( i \Rightarrow v \). Some edge in \( p \) must begin at a nondescendant of vertex \( i \) and lead to a descendant of \( i \). Let \( (u, w) \) be the last such edge in \( p \). Then \( w > i \), and all edges following \( (u, w) \) in \( p \) have both endpoints among the descendants of \( i \). Since all the descendants of \( i \) have numbers larger than \( i \), and since \( u \) is not a descendant of \( i \), we may replace the part of \( p \) up to edge \( (u, w) \) by a path of tree arcs from \( s \) to \( u \) and have a path in \( G(i) \) which doesn’t contain \( i \). But this is a contradiction by the definition of \( SDOM(i, v) \), and \( SDOM(i, v) \) must dominate \( v \) in \( G \). The lemma follows.

LEMMA 16. If \( v \neq 1 \) and \( i \leq v \), then either \( SDOM(i, v) = IDOM(v) \) or \( SDOM(i, v) \Rightarrow i \).

Proof. The proof is by induction on \( i \). If \( i = v \), \( SDOM(i, v) \Rightarrow i \) by Lemma 13. Let the lemma be true if \( i_0 < i \leq v \). Suppose \( i = i_0 \). Vertex \( i \) is a descendant of the father of vertex \( i + 1 \). By the induction hypothesis, either \( SDOM(i + 1, v) = IDOM(v) \) or \( SDOM(i + 1, v) \Rightarrow i + 1 \). If \( SDOM(i + 1, v) = IDOM(v) \), then \( SDOM(i, v) = IDOM(v) \) by Lemma 14. Otherwise \( SDOM(i + 1, v) \Rightarrow i + 1 \) and \( SDOM(i + 1, v) \neq i + 1 \) by Lemma 15. Then by Lemma 14 and the comment above, \( SDOM(i, v) \Rightarrow i \), and the lemma follows by induction on \( i \).

LEMMA 17. If \( u \) does not dominate \( v \) in \( G \), and edge \( (u, v) \) in \( G \) is replaced by edge \( (IDOM(u), v) \) to form graph \( G' \), then \( d \) dominates \( w \) in \( G \) if and only if \( d \) dominates \( w \) in \( G' \).

Proof. Suppose \( x \) does not dominate \( w \) in \( G \). Then there is a path \( p \) in \( G \) from \( s \) to \( w \) which does not contain \( x \). If \( p \) does not contain edge \( (u, v) \) then \( p \) is a path in \( G' \). Suppose \( p \) does contain \( (u, v) \). Then \( p \) contains \( IDOM(u) \), and we may replace the part of \( p \) leading from \( IDOM(u) \) to \( v \) by the edge \( (IDOM(u), v) \) and have a path in \( G' \) from \( s \) to \( w \) which doesn’t contain \( x \). Thus \( x \) does not dominate \( w \) in \( G' \).

Conversely, suppose \( x \) does not dominate \( w \) in \( G' \). Then there is a path \( p' \) in \( G' \) from \( s \) to \( w \) which doesn’t contain \( x \). If (IDOM(u), v) is not on p', then p' is a path in G. Suppose (IDOM(u), v) is on p'. If x = u, then since u doesn’t dominate v, there is a path q in G from s to v which doesn’t contain x. By substituting q for the part of p' up to and including edge (IDOM(u), v), we get a path p in G from s to w which doesn’t contain x. Suppose x \( \neq u \). If every path from IDOM(u) to u in G contains x, then x dominates u. Also, every path from s to x must contain IDOM(u), since IDOM(u) dominates u. Then IDOM(u) dominates x or
IDOM(u) = x, and either is a contradiction. It follows that there is some path r
in G from IDOM(u) to u which doesn’t contain x. Substituting r and the edge
(u, v) for the edge (IDOM(u), v) in p’, we get a path in G from s to w which doesn’t
contain x. In no case can x dominate w in G, and the lemma is true.

**Lemma 18.** Let (u, v) be a cross-link in G. Suppose G is transformed into a new
graph G’ by deleting (u, v) and adding edge (SDOM(v, u), v). We call this trans-
formation “cross-link replacement”. Then d dominates w in G if and only if d
dominates w in G’.

**Proof.** Since (u, v) is a cross-link, v < u. If SDOM(v, u) = IDOM(u), then
the lemma is true by Lemma 17, since u cannot dominate v unless u ? v,
and u ? v is impossible by Lemma 2. Suppose SDOM(v, u) ? IDOM(u). Then
SDOM(v, u) ? v by Lemma 16. Now suppose x does not dominate w in G. Then
there is a path p in G from s to w which doesn’t contain x. If (u, v) is not an edge
of p, then p is a path in G’. Suppose p contains (u, v). If x is not an ancestor of
SDOM(v, u), we may replace the part of p up to and including edge (u, v) by the
path of tree arcs from s to SDOM(v, u) and the edge (SDOM(v, u), v) to get a
path in G’ from s to w which doesn’t contain x.

On the other hand, suppose x is an ancestor of SDOM(v, u). Consider the
part of p from s to u. Let (y, z) be the last edge on this part of p with y ? v. Then
y must be an ancestor of u, since by Lemma 8 any path from y to u must pass
through a common ancestor a of y and u, and this a satisfies a ? y ? v. Vertex y
must also be an ancestor of v, since y v < u, and any ancestor of u which is
not an ancestor of v will have a number greater than the number of v. The part
of p from y to u lies in G(v), so SDOM(v, u) ? y. Otherwise there would be a
path in G(v) from s to u which didn’t contain SDOM(v, u), an impossibility. Thus
we have a vertex y on p such that x ? SDOM(v, u) ? y ? v. We may replace the
part of p from y to v (including edge (u, v)) by the path of tree arcs from y to v.
This gives a path in G’ from s to w which doesn’t contain x, and x does not domi-
nate w in G’.

Conversely, suppose x doesn’t dominate w in G’. Let p’ be a path from s to
w in G’ which doesn’t contain x. If p’ doesn’t contain (SDOM(v, u), v), then p’
is a path in G. Suppose p’ does contain (SDOM(v, u), v). If x = u, we may re-
place (SDOM(v, u), v) in p’ by the path of tree arcs from SDOM(v, u) to v and get
a path from s to w in G which doesn’t contain x. If x ? u, then there must be a
path q in G(v) (hence in G) from SDOM(v, u) to u which doesn’t contain x. Other-
wise SDOM(v, u) dominates x and x dominates u, which is impossible. Replacing
(SDOM(v, u), v) in p’ by q plus edge (u, v) gives a path from s to w in G which
doesn’t contain x. In no case can x dominate w in G, and the lemma is true.

Lemma 18 tells us how to transform cross-links into reverse fronds. Suppose
that for a fixed v, we know SDOM(v, w) for all vertices w. To convert cross-link
(u, v) into a reverse frond, we apply cross-link replacement. If the resultant edge
is still a cross-link, we apply cross-link replacement to it. We continue until we
get an edge which is a reverse frond.

Now all we need is a method for calculating semidominators. Lemma 13
tells us how to initialize the calculation. Lemma 15 tells us when we can stop
calculating semidominators for a particular vertex. The next lemma indicates
how to update the semidominator values.
LEMMA 19. Let \( i \) be a vertex in \( G \) such that the only two edges entering \( i \) are a tree arc \((u, i)\) and possibly a reverse frond \((w, i)\). If no reverse frond enters \( i \), then \( \text{SOMD}(i-1, v) = \text{SDOM}(i, v) \) for all \( v \). If a reverse frond \((w, i)\) enters \( i \), then \( \text{SDOM}(i-1, v) = w \) for all \( v \) such that \( i \not\rightarrow v \) and \( w \not\rightarrow \text{SDOM}(i, v) \rightarrow u \), and \( \text{SDOM}(i, v) = \text{SDOM}(i, v) \) for all other vertices \( v \).

Proof. If \( i \) has no frond, cross-link or reverse frond entering it, then \( G(i) = G(i-1) \); thus the first part of the lemma is true. Suppose \( i \) has a reverse frond \((w, i)\) entering it. Since \( G(i-1) \) has no edges but tree arcs which lead to vertices numbered less than \( i \), any path in \( G(i-1) \) which contains \((w, i)\) must terminate at a descendant of \( i \). Thus the paths from \( s \) to nondescendants of \( i \) are the same in both \( G(i) \) and \( G(i-1) \). This means that if \( v \) is not a descendant of \( i \), then \( \text{SDOM}(i, v) \not\rightarrow \text{SDOM}(i-1, v) \).

Suppose that \( v \) is a descendant of \( i \). If \( \text{SDOM}(i, v) \not\rightarrow \text{SDOM}(i, v) \), then there must be a path \( p \) in \( G(i-1) \) from \( s \) to \( v \) which doesn’t contain \( \text{SDOM}(i, v) \), but no such path in \( G(i) \). Path \( p \) must contain reverse frond \((w, i)\). But if it is not true that \( w \not\rightarrow \text{SDOM}(i, v) \rightarrow u \), we may replace \((w, i)\) in \( p \) by the path of tree arcs from \( w \) to \( i \) and get a path in \( G(i) \) from \( s \) to \( v \) which doesn’t contain \( \text{SDOM}(i, v) \). This contradiction implies that if \( \text{SDOM}(i-1, v) \not\rightarrow \text{SDOM}(i, v) \), then \( w \not\rightarrow \text{SDOM}(i, v) \rightarrow u \).

Now suppose it is true that \( w \not\rightarrow \text{SDOM}(i, v) \rightarrow u \). If \( \text{SDOM}(i, v) \rightarrow u \), then if \( d \not\rightarrow \text{SDOM}(i, v) \), \( d \) dominates \( \text{SDOM}(i, v) \) in \( G(i) \) and \( d \) dominates \( v \) in \( G(i) \). (The only arcs entering any ancestor of \( \text{SDOM}(i, v) \) in \( G(i) \) are tree arcs since \( \text{SDOM}(i, v) \leq u < i \).) On the other hand, any vertex which is not an ancestor of \( \text{SDOM}(i, v) \) cannot dominate \( v \) in \( G(i) \). Thus in \( G(i) \) the dominators of \( v \) are exactly the ancestors of \( \text{SDOM}(i, v) \). Now in \( G(i-1) \), if \( d \not\rightarrow w \), then \( d \) dominates \( w \) and \( d \) dominates \( v \). If \( d \not\rightarrow \text{SDOM}(i, v) \) but \( d \) is not an ancestor of \( w \), then \( d \) does not dominate \( v \), since the path of tree arcs from \( s \) to \( w \) followed by the reverse frond \((w, i)\) followed by the path of tree arcs from \( i \) to \( v \) is a path in \( G(i-1) \) which doesn’t contain \( d \). Thus the dominators of \( v \) in \( G(i-1) \) are exactly the ancestors of \( w \), and \( \text{SDOM}(i-1, v) = w \). The lemma follows.

Now we have a method for calculating semidominators. We must not overlook one subtle point. To get dominators, we will calculate semidominators, applying the various dominator-preserving transformations to simplify the calculations. We must make sure that these transformations preserve not only dominators but also semidominators; otherwise the intermediate calculations may go haywire. The next lemma takes care of this worry.

LEMMA 20. Let \( G' \) be formed from \( G \) by applying either reverse frond deletion, frond deletion, or cross-link replacement. Then all semidominator values of \( G \) and \( G' \) agree.

Proof. We must compare dominators in \( G(i) \) and \( G'(i) \) to verify that the \( i \)th semidominators in \( G \) and \( G' \) are the same. Suppose \( G' \) is formed from \( G \) by reverse frond deletion. Let \((u, v)\) be the deleted reverse frond, where \((u, v)\) is another reverse frond in \( G \) and \( u_1 > u \). If \( i \geq v \), then \( G(i) = G'(i) \), and the lemma is true. If \( i < v \), then both \((u, v)\) and \((u_1, v)\) appear in \( G(i) \), \( G'(i) \) is formed from \( G(i) \) by reverse frond deletion, and the lemma is true by Lemma 11.

Suppose \( G' \) is formed from \( G \) by frond deletion. Let \((v, \text{HIGHPT}(v))\) be the deleted frond and \((u, \text{HIGHPT}(v))\) the added reverse frond, where only a tree arc
and a reverse frond \((u, v)\) enter \(v\), and only one frond leaves \(v\). If \(i \geq \text{HIGHPT}(v)\), then \(G(i) = G'(i)\), and the lemma is true. If \(i < \text{HIGHPT}(v)\), then \((v, \text{HIGHPT}(v))\) and \((u, v)\) are in \(G(i)\), \((u, \text{HIGHPT}(v))\) is in \(G'(i)\), and \(G'(i)\) is formed from \(G(i)\) by frond deletion. In this case the lemma follows by Lemma 12.

Suppose \(G'\) is formed from \(G\) by cross-link replacement. Let \((u, v)\) be the cross-link in \(G\) which is replaced by \((\text{SDOM}(v, u), v)\) to form \(G'\). If \(i \geq v\), then \(G'(i) = G(i)\), and the lemma is true. If \(i < v\), then \(G(v)\) is a subgraph of \(G(i)\), and the \(v\)th semidominator of \(u\) is the same in both \(G(i)\) and in \(G\). Edge \((u, v)\) is in \(G(i)\) and edge \((\text{SDOM}(v, u), v)\) is in \(G'(i)\). It follows that \(G'(i)\) is formed from \(G(i)\) by cross-link replacement, and the lemma follows by Lemma 18.

4. An outline of the dominators algorithm. Now we have all the results needed to build a dominators algorithm. Below is an outline of the algorithm in ALGOL-like notation.

**procedure DOMINATORS(G, s);**

```
begin
a: apply CLASSIFY(G, s) to classify the edges and number the vertices of G reachable from s;
   let \(G_1\) be the subgraph of \(G\) containing all vertices reachable from \(s\);
   let \(G_1\) have \(V_1\) vertices;
   for each vertex \(v\) of \(G\) not in \(G_1\) do \(\text{IDOM}(v) = 0\);

b: for each vertex \(v\) of \(G_1\) do calculate \(\text{HIGHPT}(v)\); apply frond replacement to \(G_1\) to form graph \(G_2\);

c: for \(i := V_1 - 1\) until 0 do
   for \(v := 1\) until \(V_1\) do
      calculate \(\text{SDOM}(i, v)\);
   for \(i := 1\) until \(V_1\) do \(\text{IDOM}(i) := \text{SDOM}(0, i)\);
end;
```

This algorithm is straightforward and works correctly by the results in §3. (Frond replacement preserves dominators by Lemma 12.) Step a requires \(O(V + E)\) time by the discussion in §2, and the total time required by all steps except b and c is \(O(V + E)\). Using results in §3 we can give some details of Step c:

```
c: comment calculate semidominators;
begin
for \(v := 1\) until \(V_1\) do
begin
   using reverse frond deletion, delete all reverse fronds but one entering vertex \(v\);
   let this reverse frond be \((\text{LOWPT}(v), v)\);
end;
for \(i := 1\) until \(V_1\) do calculate \(\text{SDOM}(V_1, i)\) using Lemma 13;
for \(i := V_1 - 1\) until 1 do
begin
   for each cross-link \((u, i)\) do
      begin
         convert \((u, i)\) into a reverse frond \((w, i)\) by repeated
```
cross-link replacement;
   if two reverse fronds now enter \( i \) then delete one by reverse frond deletion;
end;
if a frond \((i, x)\) leaves \( i \) then delete it by frond deletion;
if two reverse fronds now enter \( x \) then delete one by reverse frond deletion;

\[ \text{h: for } v := 1 \text{ until } V_1 \text{ do } \text{apply Lemma 19 to calculate } \text{SDOM}(i - 1, v); \]

Consider this implementation of the semidominators calculation. For a fixed value of \( i \), Step f deletes any frond leaving \( i \). Thus before Step f is executed for any fixed \( j \), all fronds entering \( j \) have been deleted. It follows that the transformations in Step c preserve semidominators, by Lemma 20. Thus Step c as implemented above works correctly. The total time required by Step c, not including Steps g and h, is obviously \( O(V + E) \). The dominators algorithm thus has a linear time bound not including Steps b, g and h. These steps require some good data structures, which are presented in the next two sections.

5. Calculating \text{HIGHPT}(v). In this section we implement Step b, the calculation of \text{HIGHPT} values. A straightforward algorithm for calculating \text{HIGHPT} values requires \( O(V^2) \) time. With a good scheme for implementing priority queues, such as Crane’s using binary trees [14] or Hopcroft’s using 3-2 trees [15], we may achieve an \( O(E \log E) \) time bound. However, if we are a little more clever, then we can use a good algorithm for computing disjoint set unions and construct an almost-linear algorithm. First we sort the fronds \((u, w)\) of \( G \) by the \text{NUMBER} of \( w \). Then we calculate \text{HIGHPT}’s by processing the fronds \((u, w)\) in order from largest \( w \) to smallest \( w \). We will label each vertex exactly once with a \text{HIGHPT} value. If \((u, w)\) is the next frond to be processed, then each currently unlabeled vertex except \( w \) on the tree path from \( w \) to \( u \) has \text{HIGHPT} = \( w \) and may be so labeled. Step b is:

\[ \text{b: comment calculate } \text{HIGHPT}(v) \text{ for every vertex } v \text{ in } G_1; \]
begin
for \( i := 1 \) until \( V_1 \) do
begin
   \text{HIGHPT}(i) := 0;
   set \text{BUCKET}(i) equal to the empty list;
end;

1: for each frond \((u, w)\) in \( G_1 \) do add \((u, w)\) to \text{BUCKET}(w);

m: for \( w := V_1 \) step \(-1\) until 1 do
while \text{BUCKET}(w) is not empty do
   begin
      let \((u, w)\) \in \text{BUCKET}(W);
      delete \((u, w)\) from \text{BUCKET}(W);
      n: for each vertex \( v \neq w \) on the tree path from \( w \) to \( u \) satisfying
         \text{HIGHPT}(v) = 0 \text{ do } \text{HIGHPT}(v) = w;
   end;
Consider this calculation. Step \( m \) is a radix sort which orders the fronds on the \textit{NUMBER} of their second vertex. For any vertex \( v \), if there is a frond \((u, w)\) with \( w \neq v \) and \( w \rightarrow v \rightarrow u \), then \( \text{HIGHPT}(v) 
eq 0 \) when Step \( m \) is finished; otherwise \( \text{HIGHPT}(v) = 0 \) when Step \( m \) is finished. If \( \text{HIGHPT}(v) 
eq 0 \) when Step \( m \) is finished, then \( \text{HIGHPT}(v) \) is equal to the highest numbered vertex \( w \neq v \) such that there is a frond \((u, w)\) with \( w \rightarrow v \rightarrow u \). It follows that Step \( m \) calculates \( \text{HIGHPT} \) values correctly.

For the algorithm to work efficiently, Step \( m \) must not reexamine vertices whose \( \text{HIGHPT} \) values have already been calculated. To take care of this problem, we use a fast method for computing unions of disjoint sets \([11], [12], [13]\). We shall have sets numbered \( 1 \) to \( V_1 \). If \( v \neq 1 \) is a vertex, then \( v \) will appear in the set whose number is the highest numbered unlabeled proper ancestor of \( v \). Since vertex \( 1 \) never gets labeled, each vertex except \( 1 \) always appears in a set. Initially, if \((v, w)\) is a tree arc, then \( w \) appears in the set named \( v \).

To process frond \((u, w)\), we find the set \( u_1 \) containing \( u \), the set \( u_2 \) containing \( u_1 \), and so on, until we reach a set \( u_n \) such that \( u_n \rightarrow w \). The vertices \( u_1, u_2, \ldots, u_{n-1} \), and possibly \( u \), are the unlabeled vertices on the tree path from \( w \) to \( u \). We label them with \( \text{HIGHPT} \) value \( w \), and then we compute the union of sets \( u_1, u_2, \ldots, u_{n-1}, u_n \) (and possibly \( u \)) and number the union \( u_n \). Step \( m \) becomes:

\[
\text{begin} \quad \text{for } i := 1 \text{ until } V_1 \text{ do SET}(i) := \text{the empty set}; \\
\text{for each tree arc } (v, w) \text{ do SET}(v) := \text{SET}(v) \cup \{v\}; \\
\text{for } w := V_1 \text{ step } -1 \text{ until } 1 \text{ do} \\
\quad \text{while BUCKET}(w) \text{ is not empty do begin} \\
\quad \quad \text{let } (u, w) \in \text{BUCKET}(v); \\
\quad \quad \text{delete } (u, w) \text{ from BUCKET}(w); \\
\quad \quad n: \text{while } \neg (u \rightarrow w) \text{ do begin} \\
\quad \quad \quad x := \text{FIND}(u); \\
\quad \quad \quad \text{if } \text{HIGHPT}(u) = 0 \text{ then begin} \\
\quad \quad \quad \quad \text{SET}(x) := \text{SET}(x) \cup \text{SET}(u); \\
\quad \quad \quad \quad \text{HIGHPT}(u) := w; \\
\quad \quad \quad \text{end}; \\
\quad \quad u := x; \\
\quad \text{end}; \\
\text{end}; \\
\text{end};
\]

All the set unions in Step \( m \) are unions of disjoint sets. The operation \( \text{FIND}(x) \) computes the number of the set containing \( x \) as an element. Implementation and timing of the union and find operations are discussed in Appendix B. It is not hard to prove by induction on the number of vertices labeled that at all times
during execution of Step m, vertex \( w \) appears in set \( v \) if and only if \( v \) is the highest numbered unlabeled proper ancestor of \( w \). It follows that Step m calculates HIGHPT values correctly.

**Lemma 21.** Step b (calculating HIGHPT values) requires \( O(V \log V + E) \) time.

**Proof.** Initialization requires \( O(V) \) time. Step 1 requires \( O(E) \) time. Step m requires \( O(V + E) \) time for removing fronds from buckets. Step n requires \( O(V + E) \) time exclusive of set unions and finds. There is one find for each frond plus at most one find for each vertex, giving \( O(V + E) \) finds. There is one set union for each labeled vertex and one set union for each tree arc when the sets are initialized, giving in all \( O(V) \) set unions. Using the implementation for finds and unions discussed in Appendix B, the set operations require \( O(V \log V + E) \) time. Combining these facts gives the lemma.

The set unions and finds done in Step b actually require less time than the bound in Lemma 21 indicates, but Step b is not the slowest part of the dominators algorithm, and the bound in Lemma 21 is good enough.

6. Calculating semidominators. In this section, we implement Steps g and h, the conversion of cross-links to reverse fronds and the calculation of semidominators. A straightforward algorithm for these steps requires \( O(V^2) \) time, but we can do better by using good data structures. First, consider the conversion of a cross-link to a reverse frond. Suppose we are processing vertex \( v \), and we want to convert cross-link \((u, v)\) to a reverse frond. For any \( w > v \), either \( \text{SDOM}(v, w) \rightarrow v \) or \( \text{SDOM}(v, w) = \text{IDOM}(w) \), by Lemma 16. We can apply Lemma 15 to discover whether \( \text{SDOM}(v, w) = \text{IDOM}(w) \); if \( \text{SDOM}(v, w) \) is not a proper ancestor of \( v \), we will know the value of \( \text{IDOM}(w) \).

The semidominator calculations build the dominator tree from the leaves downward; at any given time, the part of the dominator tree which we know will consist of several vertices and all their descendants in the dominator tree. Let this set of subtrees be \( F \). We shall use sets numbered 1 through \( V_1 \) (called ISET’s) to contain information about \( F \). If \( v \) is a vertex, \( v \) will be in the ISET whose number is the root of the subtree in \( F \) which contains \( v \). If \( v \) is in no subtree in \( F \), then \( v \) will be in ISET(\( v \)). Initially each ISET(\( v \)) contains exactly one element, \( v \) itself. To update the ISET’s, each time we calculate \( \text{IDOM}(v) \) for a new vertex, we let \( \text{ISET}(\text{IDOM}(v)) = \text{ISET}(v) \cup \text{ISET}(\text{IDOM}(v)) \). We can use the set union algorithm in Appendix B to keep track of the sets.

To convert cross-link \((u, v)\) to a reverse frond, we find the set \( x \) which contains \( u \). Then \((u, v)\) may be converted to \((x, v)\) by repeated cross-link replacement. (None of the elements of the set \( x = \text{ISET}(u) \) can be an ancestor of \( v \), since if \( y \rightarrow v \), then \( y < v \) and \( y \in \text{ISET}(v) \) when \( v \) is being processed.) Either \((x, v)\) is a reverse frond or \( \text{SDOM}(v, x) \rightarrow v \), and applying one more cross-link replacement gives a reverse frond. This is the crux of our implementation of Step g; now we must see how to keep track of semidominator values.

For fixed \( i \), we do not want to calculate \( \text{SDOM}(i - 1, v) \) for all vertices \( v \), since for most vertices \( \text{SDOM}(i - 1, v) = \text{SDOM}(i, v) \). We only want to calculate semidominators which change when \( i \) changes. We use a set of priority queues to keep track of the semidominators. A priority queue contains a set of items, each with an attached numeric priority. We need to be able to add an item with any
priority to a queue and to remove the item with highest priority from a queue. (If
two or more items have the same highest priority, we do not care which is re-
moved first.) We also need to be able to combine two priority queues to give a
large queue containing all items from both old queues.

Several good methods for implementing priority queues are known [14],
[15]. They all use some sort of tree representation, and have a time bound of
$O(n \log n)$ to perform $n$ operations of the three types discussed above, starting
with initially empty queues. We shall not discuss here how to implement priority
queues; let us assume that we have some good implementation on hand.

To implement the semidominator calculations using priority queues, we set
up a queue for each vertex $v$. Queue $v$ will contain items, each of which is a set of
descendants of $v$. (These sets we call QSET’s.) All vertices $w$ in a QSET will have
the same value of $\text{SDOM}(v, w)$, and this value will be the priority of the QSET
in the queue. Only vertices whose $\text{IDOM}$ values are not known are included in
QSET’s; thus if $\text{SDOM}(v, w)$ is the priority of some QSET on the queue for $v$,
$\text{SDOM}(v, w) \rightarrow v$.

To update the semidominators when processing vertex $v$, let $(\text{LOWPT}(v), v)$
be the reverse frond (if any) entering vertex $v$. All sons of $v$ have already been
processed. First we construct a priority queue for $v$ by combining the queues of
the sons of $v$. Each QSET in the new queue has a priority corresponding to some
ancestor of $v$. We remove each QSET having priority $v$ (the highest possible
priority). Each vertex in a removed QSET has $\text{IDOM}$ value equal to $v$, by Lemma
15. We label these vertices with $\text{IDOM}$ values and update the ISET’s as described
above. If $v$ has no reverse frond entering it, we add to the queue a new QSET $\{v\}$
with priority equal to the father of $v$. If $v$ has a reverse frond entering it, we remove
each QSET with priority equal to or greater than $\text{LOWPT}(v)$, we compute the
union of all these QSET’s, we add $v$ as an element to this QSET, and we add the
new QSET to the queue with priority $\text{LOWPT}(v)$. This implements Lemma 19
for updating the semidominator calculations.

We handle the QSET unions using the set union algorithm described in
Appendix B. Each vertex appears in at most one QSET which is on the priority
queue of some vertex. For convenience, we assign each QSET a name consisting of
its priority and some number distinguishing QSET’s with the same priority. We
now can finish the implementation of repeated cross-link replacement. Once a
cross-link $(u, v)$ has been converted into an edge $(x, v)$ with $\text{IDOM}(x)$ undefined,
if $(x, v)$ is not a reverse frond, then $x$ must be in some QSET. Let $y$ be the priority
of this QSET in its queue. Then $(y, v)$ is a reverse frond and may be substituted
for $(x, v)$.

Steps g and h are given below in ALGOL-like notation.

\begin{verbatim}
comment we need some initialization to set up the ISET’s; for $i := 1$ until $V_1$ do
ISET(i) := \{i\};
g: begin comment convert $(u, i)$ into a reverse frond by repeated cross-link re-
placement;
  $x := \text{IFIND}(u)$;
  if $x \rightarrow i$ then replace $(u, i)$ by $(x, i)$;
  else
\end{verbatim}
begin
  (y, v) := QFIND(x);
  replace (u, i) by (y, i);
end;

We shall assume for convenience in Step h that the priority queue operations are implemented so that if we try to remove a set from an empty queue, we get an empty set called QSET(0, 0), and if we try to add QSET(0, 0) to a queue, nothing gets added to the queue.

\[ h \begin{array}{l}
  \text{begin comment calculate SDOM}(i - 1, v) \text{ for all } v \text{ such that } SDOM(i - 1, v) \\
  \neq SDOM(i, v). \text{ The semidominator of a vertex } v \text{ is the priority of the} \\
  \text{QSET containing it, if } v \geq i \text{ and IDOM}(v) \text{ is yet unknown}; \\
  \text{QUEUE}(i) := \text{the empty queue}; \\
  \text{for } w \text{ a son of } i \text{ do} \\
  \text{QUEUE}(i) = \text{QUEUE}(i) \cup \text{QUEUE}(w); \\
  \text{remove QSET}(z, j) \text{ with highest priority } z \text{ from QUEUE}(i); \\
  \text{while } z = i \text{ do} \\
  \text{begin} \\
  \text{for each element } e \in \text{QSET}(z, j) \text{ do} \\
  \text{begin} \\
  \text{IDOM}(e) := i; \\
  \text{ISET}(i) := \text{ISET}(e) \cup \text{ISET}(i); \\
  \text{end}; \\
  \text{remove QSET}(z, j) \text{ with highest priority } z \text{ from QUEUE}(i); \\
  \text{end}; \\
  \text{add QSET}(z, j) \text{ to QUEUE}(i); \\
  \text{if } \text{LOWPT}(j) \geq i \text{ then} \\
  \text{begin comment } (\text{LOWPT}(j), i) \text{ is the reverse frond (if any) entering} \\
  \text{vertex } i; \\
  \text{QSET}(\text{FATHER}(i), i) := \{i\}; \\
  \text{add QSET}(\text{FATHER}(i), i) \text{ to QUEUE}(i); \\
  \text{end} \\
  \text{else} \\
  \text{begin} \\
  \text{QSET}(\text{LOWPT}(i), i) := \{i\}; \\
  \text{remove QSET}(z, j) \text{ with highest priority } z \text{ from QUEUE}(i); \\
  \text{while } \text{LOWPT}(i) \leq z \text{ do} \\
  \text{begin} \\
  \text{QSET}(\text{LOWPT}(i), i) := \text{QSET}(z, j) \cup \text{QSET}(\text{LOWPT}(i), i); \\
  \text{remove QSET}(z, j) \text{ with highest priority } z \text{ from QUEUE}(i); \\
  \text{end}; \\
  \text{add QSET}(z, j) \text{ to QUEUE}(i); \\
  \text{add QSET}(\text{LOWPT}(i), i) \text{ to QUEUE}(i); \\
  \text{end}; \\
  \end{array} \]

Step h is a straightforward implementation using the preceding ideas, and it is easy to prove the following hypothesis by induction on the number of times
that Step h is executed: if \( v \in \text{ISET}(x) \), \( x \) is the highest numbered ancestor of \( v \) in the dominator tree such that IDOM(i) has not yet been calculated; if \( v \) is in QSET(z, j) and QSET(z, j) is in QUEUE(i), then SDOM(i, v) = z; if IDOM(v) \( \neq 0 \), then IDOM(v) has the correct value. It follows that Steps g and h correctly calculate dominators.

**Lemma 22.** If the dominators algorithm uses Steps g and h as implemented above, then the total running time of Steps g and h is \( O(V \log V + E) \).

**Proof.** Ignoring set and queue operations, the total running time of Steps g and h is \( O(V + E) \). \( O(V) \) unions of ISET’s and \( O(E) \) finds on ISET’s will be carried out in Steps g and h. \( O(V) \) unions of QSET’s and \( O(V) \) finds on QSET’s will be carried out. The total cost of all the set operations is thus \( O(V \log V + E) \) by the timing result in Appendix B. \( O(V) \) additions to QUEUE’s, \( O(V) \) additions to QUEUE’s, and \( O(V) \) deletions from QUEUE’s are carried out. By Crane’s results [14], the priority queue operations may be carried out in \( O(V \log V) \) time. Combining these results gives the lemma.

7. **The complete dominators algorithm.** This section contains the entire dominators algorithm in ALGOL-like notation. Several of the steps which have been discussed separately are combined; for instance, the initial depth-first search can be used to sort the fronds by the value of their second vertex and to begin the process of reverse frond deletion. The search can also be used to calculate the father of each vertex in the generated tree; this information is needed to initialize the semidominator calculations. The set and priority queue operations are not implemented here, but are assumed to be primitive operations. (The time required by these operations is included in the time bound for the entire algorithm, however.) Here is the dominators algorithm:

```
procedure DOMINATORS(G, s);
begin comment we assume that the graph G is represented as a set of adjacency lists A(v);
procedure SEARCH(v);
begin comment this is the modified version of DFSEARCH used to initially explore the graph. It numbers the vertices, classifies the edges, deletes all but one reverse frond (LOWPT(v), v) entering each vertex v, sorts the fronds using a radix sort, and computes the number of descendants and the father of each vertex in the generated tree. Vertices not reached during the search have no dominators. Variable m denotes the last NUMBER assigned to any vertex. Variable n denotes the last SNUMBER assigned to any vertex;
    m := NUMBER(v) := m + 1;
    ND(v) := 1;
    for w A(v) do
    if NUMBER(w) = 0 then
    begin
        FATHER(w) := v;
        SEARCH(w);
        ND(v) := ND(v) + ND(w);
    end
```
else if SNUMBER(w) = 0 then
    begin comment vertex w is stacked and (v, w) is a frond;
        i: add (v, w) to BUCKET(w);
    end
else if NUMBER(v) < NUMBER(w) then
    begin (v, w) is a reverse frond;
        d: if NUMBER(v) < LOWPT(w) then LOWPT(w) := NUMBER(v);
    end
else add (v, w) to list of cross-links entering w;
end;

integer m, n;
a: comment to classify the edges we initialize and call SEARCH;
m := 0;
n := V + 1;
for each vertex v do
    begin
        NUMBER(v) := SNUMBER(v) := 0;
        BUCKET(v) := the empty list;
        IDOM(v) := 0;
        LOWPT(v) := V;
    end;
SEARCH(s);
Vl := m;

comment henceforth for convenience we assume that the program refers
to each vertex by its number;

modify all data structures so that vertices are named by their number;

B: comment calculate HIGHPT(v) for every reachable vertex v;

for i := 1 until Vl do
    begin
        HIGHPT(i) := 0;
        SET(i) := the empty set;
    end;
for i := 2 until Vl do
    SET(FATHER(i)) := SET(FATHER(i)) U SET(i);
m: for w := Vl step - 1 until 1 do
    while BUCKET(w) is not empty do
        begin
            let (u, w) ∈ BUCKET(w);
            delete (u, w) from BUCKET(w);
            n: while ¬(u → w) do begin
                x := FIND(u);
                if HIGHPT(u) = 0 then
begin
  SET(x) := SET(x) U SET(u);
  HIGHPT(u) := w;
end;
  u := x;
end;
end;
c: comment calculate semidominators;
for i := 1 until V_1 do ISET(i) = \{i\};
for i := V_1 step - 1 until 1 do
  f: begin
    for each cross-link (u, i) do
      g: begin comment convert (u, i) to a reverse frond by repeated
        cross-link replacement;
        x := IFIND(u);
        if (x \rightarrow i) then replace (u, i) by (x, i)
        else begin
          (x, v) := QFIND(x);
          replace (u, i) by (x, i);
        end;
        comment if two reverse fronds now enter i then delete one
        if x \rightarrow LOWPT(i) then LOWPT(i) := x;
    end;
    comment if a frond leaves i then delete it and add a reverse frond if
    necessary;
    if HIGHPT(v) < v and (LOWPT(v) < v) and
    (LOWPT(LOWPT(v)) > HIGHPT(v)) then
      LOWPT(LOWPT(v)) := HIGHPT(v);
    h: comment calculate SDOM(i, v) for all v such that SDOM(i, v) \neq SDOM(i, v). The semidominator of a vertex v is the priority
    of the QSET containing it, if v \geq i and IDOM(v) is not yet known;
    QUEUE(i) := the empty queue;
    for w a son of i do
      QUEUE(i) := QUEUE(i) \cup QUEUE(w);
    remove QSET(z, j) with highest priority z from QUEUE(i);
    while z = i do
      begin
        for each element v \in QSET(z, j) do
          begin
            IDOM(v) := i;
            ISET(i) := ISET(i) \cup ISET(v);
          end;
        remove QSET(z, j) with highest priority z from QUEUE(i);
      end;
    add QSET(z, j) to QUEUE(i);
    if LOWPT(i) \geq i then
begin
  QSET(FATHER(i), i) := \{i\};
  add QSET(FATHER(i), i) to QUEUE(i);
end
else
  begin
    QSET(LOWPT(i), i) := \{i\};
    remove QSET(z,j) with highest priority z from QUEUE(i);
    while LOWPT(i) \leq z do
      begin
        QSET(LOWPT(i), i) := QSET(LOWPT(i), i) \cup QSET(z,j);
        remove QSET(z,j) with highest priority z from QUEUE(i);
      end;
    add QSET(z,j) to QUEUE(i);
    add QSET(LOWPT(i), i) to QUEUE(i);
  end;
end;

This gives a complete algorithm for calculating dominators. Figure 5 shows the graph which results when all the dominator-preserving transformations are applied to the graph in Fig. 4. It is easy to verify that the algorithm has an \(O(V + E)\) space bound. Combining the timing results in §§ 2, 4, 5 and 6, we see that the dominators algorithm has an \(O(V \log V + E)\) time bound if the set and priority queue operations are implemented efficiently. The slowest parts of the algorithm are those which require priority queues; the set union operations run faster than the queue operations. If we could somehow handle the semidominator calculations using set unions (as we handled the HIGHPT calculations), then we could construct an even faster algorithm. The next section outlines how this may be done for a special case.

8. Toward a faster algorithm. The slow part of the dominators algorithm is the use of priority queues. If we could somehow use sets in place of priority queues (as we could for the HIGHPT calculations), then we could construct a faster dominators algorithm. This section outlines the construction of such an algorithm for the case when \(G\) has no cross-links.

Suppose \(G\) is a graph which has no cross-links when explored from vertex \(s\) using depth-first search. To find dominators in \(G\), we classify the edges and number the vertices of \(G\) as before. Then we calculate HIGHPT values using the method described in § 4. Next, we apply reverse frond deletion and frond deletion repeatedly, until we have converted \(G\) into a graph with no fronds, no cross-links, and at most one reverse frond entering each vertex. Now we don’t have to bother with calculating semidominators; we can calculate the dominators directly.

To calculate the dominators, we sort the remaining reverse fronds \((u, v)\) of \(G\) so that if \((u_2, v_2)\) follows \((u_1, v_1)\), then \(u_2 > u_1\) or \(u_2 = u_1\) and \(v_2 < v_1\). This may
be done using a two-pass radix sort with $V$ buckets, similar to the sorting for the calculation of HIGHPT values. We then process the reverse fronds in sorted order, using the following lemma.

**Lemma 23.** Let $G$ be a graph such that each vertex is reachable from $s$, each vertex has at most one reverse frond entering it, and $G$ contains no cross-links or fronds. Suppose $v \neq 1$. If $v$ has no reverse frond entering it, IDOM($v$) is the father of $v$ in the spanning tree of $G$. If $v$ has a reverse frond $(u, v)$ entering it and no reverse frond $(x, w)$ satisfies $x \leq u \leq w$ and $x < u < w < v$, then IDOM($v$) = $u$. Otherwise let $(x, w)$ be the reverse frond with smallest $x$ (largest $w$) satisfying $x \leq u \leq w$ and $x < u < w < v$. Then IDOM($v$) = IDOM($w$).

**Proof.** If $v$ has no reverse frond entering it, every path to $v$ must pass through the father of $v$, and IDOM($v$) is the father of $v$. Suppose $(u, v)$ enters $v$ but no reverse frond $(x, w)$ satisfies $x \preceq u \preceq w \preceq v$ and $x < u < w < v$. Suppose path $p$ leads from $s$ to $v$ but doesn’t contain $u$. Let $(x, w)$ be the first edge on $p$ with $w > u$. $(x, w)$ must be a reverse frond with $x \preceq u \preceq w \preceq v$ and $x < u < w < v$. But this is a contradiction, so $u$ dominates $v$. Since $(u, v)$ is an edge, IDOM($v$) = $u$.

If some reverse frond $(x, w)$ satisfies $x \preceq u \preceq w \preceq v$ and $x < u < w < v$, let $(x, w)$ be one with smallest $x$ (largest $w$). It is clear that any vertex which does not dominate $w$ cannot dominate $v$. Thus every vertex which dominates $v$ must also dominate $w$. Now we need only show that IDOM($w$) dominates $v$.

Suppose, to the contrary, that $p$ is a path from $s$ to $v$ which doesn’t contain IDOM($w$). Let $(i, j)$ be the first edge on this path satisfying $j \geq x(j \geq u)$. Then IDOM($w$) cannot dominate $w$ because $j \preceq u$, $x \preceq j \preceq u(j \preceq w, u \preceq j \preceq w)$, and
we may form a path from \( s \) to \( w \) which doesn’t contain \( \text{IDOM}(w) \). This contradiction gives the rest of the lemma.

To calculate dominators, we start with one set numbered 0 containing all the vertices. Then we process the reverse fronds in the order described above. At any given time, a vertex \( v \) will be in a set labeled \( x \) if \( x \) is the largest vertex such that \( x \rightarrow v \) and a reverse frond \( (u, x) \) has been processed. To process a reverse frond \( (u, v) \), we locate \( u \) and \( v \) in sets. If they are in the same set, \( \text{IDOM}(v) = u \) by Lemma 23. If they are in different sets, \( \text{IDOM}(v) = \text{IDOM}((\text{FIND}(v)) \) by Lemma 23. It happens that in this case \( \text{IDOM}(\text{FIND}(v)) \) will already have been computed, but even if this were not true we could fill in the value of \( \text{IDOM}(v) \) later, once we knew the value of \( \text{IDOM}(\text{FIND}(v)) \). In any case, we split the set containing \( v \) into two parts: a set containing descendants of \( v \), having label \( v \), and a set containing nondescendants of \( v \), having the same label as the old set containing \( v \). This algorithm computes dominators, if we can implement the set-splitting operation.

Actually, we don’t have to split sets; we can run this algorithm backwards, and turn the splits into union operations. Then we can use the algorithm described in Appendix B. The correct labels for all the resultant sets and the \( \text{IDOM} \) values must be filled in after the set union operations are carried out, but this is no great problem. The resultant dominators algorithm has a time bound of \( O(V + E) \) not counting set unions and finds. There are \( O(V) \) unions in the HIGHPT and dominator calculations and \( O(V + E) \) finds. If \( E \geq V \log V \), we get the same overall bound as the algorithm in §7: \( O(E) \). If \( E \) is substantially smaller than \( V \log V \), we get a better bound than that for the algorithm in §7, namely, \( O((V + E) \cdot \log^* (V + E)) \), where \( \log^* x = \min \{ i | i \log^i(x) \leq 1 \} \) (see [12], [13]). It seems possible that this faster algorithm may be generalized to handle arbitrary graphs.

9. Conclusions. This paper has presented an algorithm for finding dominators in directed graphs. The algorithm illustrates the use of depth-first search for revealing the connectivity structure of a graph and the use of sophisticated data structures in building efficient graph algorithms. The algorithm requires \( O(V + E) \) space and \( O(V \log V + E) \) time to find dominators in a graph with \( V \) vertices and \( E \) edges. The time bound compares favorably with the \( O(V(V + E)) \) time bound of previously known algorithms such as Aho and Ullman’s [1] and Purdom and Moore’s [3] for finding dominators in arbitrary graphs, and with the \( O(E \log E) \) time bound of Aho, Hopcroft and Ullman’s algorithm for finding dominators in reducible graphs [6]. If \( E \geq V \log V \), then the time bound is \( O(E) \), and the new algorithm is optimal to within a constant factor, since every edge must be examined to determine dominators. Although the algorithm is based on some delicate graphical transformations, it is easy to program.

Still open is the question of whether a faster algorithm exists if \( E < V \log V \). Section 8 gives a faster algorithm for graphs which have no cross-links when they are explored using depth-first search. Many, but not all program flow graphs have this special form; in particular, the \textbf{if} \ldots \textbf{then} \ldots \textbf{else} construction produces a cross-link in the resultant flow graph. By adding vertices, any program flow graph may be converted into a computationally equivalent program flow graph which has no cross-links. However, the number of vertices in the graph may
grow enormously. The slower but more general algorithm thus seems more useful than the faster algorithm. However, it may be that the algorithm in § 8 can be extended to reducible graphs or even to arbitrary graphs.

**Appendix A. Basic definitions.** A directed graph $G = (\mathcal{V}, \mathcal{E})$ is an ordered pair consisting of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. Each edge is an ordered pair $(v, w)$ of distinct vertices. We say edge $(v, w)$ leaves $v$ and enters $w$. A graph contains no loops (edges of the form $(v, v)$) and no multiple edges, although the algorithms presented in this paper may be easily modified to handle graphs with loops and multiple edges. A graph $G_1 = (\mathcal{V}_1, \mathcal{E}_1)$ is a subgraph of a graph $G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ if $\mathcal{V}_1 \subset \mathcal{V}_2$ and $\mathcal{E}_1 \subset \mathcal{E}_2$.

If $G$ is a graph, a path $p : v \Rightarrow w$ is a sequence of vertices and edges leading from vertex $v$ to vertex $w$. A vertex $w$ is reachable from vertex $v$ if there is a path from $v$ to $w$. A path is simple if all its edges are distinct. A path $p : v \Rightarrow v$ is called a closed path. A closed path may contain no edges. A closed path $p : v \Rightarrow v$ is a cycle if all its edges are distinct and the only vertex to occur twice is $v$, which occurs exactly twice. A cycle contains at least two edges. Two cycles which are cyclic permutations of each other are considered to be the same cycle. A directed graph is acyclic if it contains no cycles.

A (directed, rooted) tree $T$ is a graph with one distinguished vertex called the root $r$ such that every vertex in $T$ is reachable from $r$, no edges enter $r$, and exactly one edge enters every other vertex in $T$. A tree vertex with no exiting edges is called a leaf. The relation “$(v, w)$ is an edge in $T$” is denoted by $v \rightarrow w$. The relation “there is a path from $v$ to $w$ in $T$” is denoted by $v \Rightarrow w$. If $v \rightarrow w$, $v$ is the father of $w$ and $w$ is a son of $v$. If $v \Rightarrow w$, $v$ is an ancestor of $w$ and $w$ is a descendant of $v$. Every vertex is an ancestor and a descendant of itself. If $v \Rightarrow w$ and $v \neq w$, $v$ is a proper ancestor of $w$ and $w$ is a proper descendant of $v$. If $T_1$ is a tree and $T_2$ is a subgraph of $T_2$, then $T_1$ is a subtree of $T_2$. If $T$ is a tree which is a subgraph of a directed graph $G$ and $T$ contains all the vertices of $G$, then $T$ is a spanning tree of $G$. References on directed graphs include Busacker and Saaty [18], Harary, Norman and Cartwright [19], and Ore [20].

If $f$ and $g$ are functions of $x$, we say $g(x)$ is $O(f(x))$ if there are constants $k_1$ and $k_2$ such that $|g(x)| \leq k_1|f(x)| + k_2$ for all $x$.

**Appendix B. A good set union algorithm.** Suppose we are given a collection of disjoint sets. We want to carry out operations of two types on the sets: FIND($x$), which computes the name of the set containing $x$ as an element, and UNION($A, B, C$), which computes the union of sets $A$ and $B$ and names the new set $C$. Initially we have $n$ distinct elements, each in a singleton set. We then carry out $n - 1$ unions and $m$ intermixed finds. We desire a good method for implementing these operations.

A very simple algorithm will solve the problem. Each set is represented as a tree. Each tree vertex represents an element in the set, and the root of the tree represents the entire set as well as some element in the set. Each tree vertex is represented in a computer by a cell containing either two or three items. A cell representing a nonroot vertex contains the element corresponding to the vertex and a pointer to the cell for the father of the vertex in the tree. A cell corresponding
to a root contains the element corresponding to the root, the name of the set corresponding to the tree with that root and the number of vertices in the set.

To carry out FIND(x), we locate the cell containing element x and follow pointers to the cell for the root of the corresponding tree. This cell contains the name of the set. In addition we collapse the tree by changing the father of each vertex reached on the way to the root. The root itself becomes the father of each of these vertices. This collapsing process saves time in later finds. To carry out UNION(A, B, C), we choose the set with fewer elements, say A. Then we make the root of A a son of the root of B. The cell corresponding to the root of A becomes a nonroot cell pointing to the cell for the root of B. The root cell of B is changed to contain the name C and the sum of the number of elements in A and B.

Although this set union algorithm is very simple, it is very hard to analyze [11], [12], [13]. Hopcroft and Ullman [12] have studied the algorithm and shown that its running time is $O((n + m) \log^* (n + m))$, where $\log^* x = \min \{i | \log^i x \leq 1 \}$.

Tarjan [13] has derived the same upper bound on the running time using a different method and has also shown that the algorithm does not have a linear upper bound on its running time. The exact running time of the algorithm is still unknown. However, for our purposes, a loose upper bound is all that we need. The bound below is generally known but apparently unpublished.

It is useful to think about the set union algorithm in the following way: suppose we perform all $n-1$ unions first. Then we have a single tree with $n$ vertices. Each of the original finds is now a “partial” find in the new tree: to carry out FIND(x), we follow fathers from x to the closest ancestor of x corresponding to a union which appears before FIND(x) in the original sequence of operations. In this interpretation of the problem, we are interested in bounding the total length of $m$ partial finds performed on a tree generated by $n-1$ set unions. (The total time required for the set unions is $O(n)$; the time for a find is proportional to its length.)

Let $T$ be a tree containing $n$ vertices numbered 1 through $n$ which has been constructed using $n-1$ set unions. Let $d_i$ be the number of descendants of vertex $i$. Let $C(T)$, the cost of tree $T$, be defined by

$$C(T) = \sum_{i=1}^{n} d_i.$$ 

Let $\bar{C}(n)$ be the maximum cost of a tree with $n$ vertices constructed by applying set unions. Then we have the following.

**Lemma 28.** $\bar{C}(n) \leq n \log 2n$.

**Proof.** We prove the lemma by induction on $n$. $\bar{C}(1) = 1 = 1 \cdot \log 2$. Suppose the lemma is true for $n < k$. Let $n = k$. Let $T$ be a tree such that $C(T) = \bar{C}(n)$ and $T$ is formed by taking the union of trees $T_1$ with $a$ vertices and $T_2$ with $b$ vertices, $a \leq b$, $a + b = n$. Then:

$$\bar{C}(n) = C(T) = C(T_1) + C(T_2) + n - b$$

$$\leq a \log 2a + b \log 2b + a$$

$$\leq a(\log 2n - 1) + b(\log 2n) + a$$

$$\leq n \log 2n.$$
The lemma follows by induction on $n$.

Now suppose we apply a partial find of length $k$ to a tree $T$. Assume without loss of generality that the find starts at vertex 1 and causes vertices $1, 2, \ldots, k - 1$ to become sons of vertex $k$. Let $T'$ be the tree after this find is performed, and let $d_i'$ be the number of descendants of vertex $i$ in $T'$. Then $d'_1 = d_1$, $d'_i = d_i - d_{i-1}$ for $2 \leq i \leq k - 1$, and $d'_k = d_k$. Since $d'_i \geq d'_{i-1} + 1 \geq i \geq 1$, it follows that $C(T') \leq C(T) - k - 2$, and we have the following result.

**Lemma 29.** If $m$ partial finds are performed on a tree with $n$ vertices formed with $n - 1$ unions, the total length of all the finds is $O(n \log n + m)$.

**Proof.** Let $k_i, i = 1, \ldots, m$, be the length of the $i$th find. Since every tree has positive cost and any find of length $k$ decreases the cost of the corresponding tree by at least $k - 2$, we have

$$n \log 2n - \sum_{i=1}^{m} (k_i - 2) > 0.$$  

It follows that

$$\sum_{i=1}^{m} k_i < 2m + n \log 2n.$$  

Lemma 29 implies that $n - 1$ unions and $m$ finds require $O(n \log n + m)$ time.

REFERENCES


