APPLICATIONS OF PATH COMPRESSION ON BALANCED TREES

by

Robert E. Tarjan

STAN-CS-75-512
AUGUST 1975

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Applications of Path Compression on Balanced Trees

Robert Endre Tarjan [*]
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

We devise a method for computing functions defined on paths in trees. The method is based on tree manipulation techniques first used for efficiently representing equivalence relations. It has an almost-linear running time. We apply the method to give $O(m\,\alpha(m,n))$ algorithms for two problems.

A* Verifying a minimum spanning tree in an undirected graph (best previous bound: $O(m \log \log n)$).

B. Finding dominators in a directed graph (best previous bound: $O(n \log n + m)$).

Here n is the number of vertices and m the number of edges in the problem graph, and $\alpha(m,n)$ is a very slowly growing function which is related to a functional inverse of Ackermann's function.

The method is also useful for solving, in $O(m\,\alpha(m,n))$ time, certain kinds of pathfinding problems on reducible graphs. Such problems occur in global flow analysis of computer programs and in other contexts. A companion paper will discuss this application.

Keywords: balanced tree, dominators, equivalence relation, global flow analysis, graph algorithm, minimum spanning tree, path compression, pathfinding problem, tree.

[*] Research supported by a Miller Research Fellowship at University of California, Berkeley; and by NSF Grant DCR72-03752 A02 at Stanford University. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Introduction.

There is a small collection of basic techniques which are useful for building efficient algorithms for a wide variety of graph problems. Here we study one such technique, path compression on balanced trees. The technique is a combination of the ideas of several people. It was first used for efficiently representing equivalence relations, and was subsequently applied to a variety of problems. See [2,3,13,21,36] for extensive discussions and applications.

We significantly extend the range of application of the technique by using it to compute functions defined on paths in trees. We apply this function evaluation method to give $O(m\ \alpha(m,n))$ algorithms for two seemingly diverse problems:

A. Verifying a minimum spanning tree in an undirected graph (previous best bound: $O(m \log \log n)$ [10,33,40]).

B. Finding dominators in a directed graph (previous best bound: $O(n \log n + m)$ [34,38]).

Here n is the number of vertices and m the number of edges in the problem graph, and $\alpha(m,n)$ is a very slowly growing function which is related to a functional inverse of Ackermann's function.

The method is also useful for solving, in $O(m\ \alpha(m,n))$ time, certain kinds of pathfinding problems on reducible graphs. Reducible graphs are a special class of directed graphs which arise naturally when considering global properties of computer programs [7,12,18,19]. Solvable types of pathfinding problems include computing path sets using regular expressions [9,32], solving linear equations [15], and doing global flow analysis of computer programs [14,17,23]. These applications will be discussed in a companion paper. The best previous bound for these problem's is $O(m \log n)$ [5,14,17,23,39].

The paper contains ten sections. Section 2 gives definitions and various preliminary results. Section 3 solves the function evaluation problem using an algorithm which works in general but is highly efficient only for balanced trees. Section 4 discusses two previous applications of path compression on balanced trees. Section 5 presents a method of decomposing the function evaluation problem into a problem on a balanced tree and a problem on paths. Section 6 presents a simple, efficient algorithm for paths when the function of interest is max. Section 7 presents an efficient algorithm for paths which works for any function. Section 8 applies the algorithm to the problem of verifying a minimum spanning tree and to two similar problems. Section 9 applies the algorithm to the problem of finding dominators in a directed graph. Section 10 discusses lower bounds for various forms of the function evaluation problem.

2. Definitions and Preliminary Results.

This section contains the basic notions needed to discuss the function evaluation algorithm. We will introduce more advanced notions as needed.

A *graph* $G = (V,E)$ consists of a finite set; V of $n = |V|$ elements called *vertices* and a set E of $m = |E|$ elements called *edges*. Either the edges are ordered pairs (v,w) of distinct vertices (the graph is *directed*) or the edges are unordered pairs of distinct vertices, also represented as (v,w) (the graph is *undirected*). A directed edge (v,w)

is said to leave v and enter 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 of length k from v to w in G is a sequence of edges $(v_1,v_2),(v_2,v_3),\ldots,(v_k,v_{k+1})$ with $v_1 = v$ and $v_{k+1} = w$. The path contains vertices $v_1, v_2, \ldots, v_{k+1}$ and edges (v_1,v_2). Se (v_k, v_{k+1}) and avoids all other vertices and edges. The path is simple if $v_1, \ldots, v_{k+1}$ are distinct (except possibly $v_1 = v_{k+1}$) and the path is a cycle if $v_1 = v_{k+1}$. By convention there is a path of no edges from every vertex to itself but a cycle must contain at least two edges. An undirected graph is connected if there is a path joining every pair of vertices.

A tree T = (V,E) is an undirected graph such that T is connected and contains no cycles. If a tree T is a subgraph of a graph G with the same vertex set as T , then T is a spanning tree of G . In a tree T there is a unique simple path between any two vertices v and w ; we denote this path by T(v,w).

A rooted tree (T,r) is a tree with a distinguished vertex r , called the root. If v and w are vertices in a rooted tree (T,r) , we say v is an ancestor of w and w is a descendant of v (denoted by $v \overset{*}{\rightarrow} w$) if v is on the path from r to w . By convention $v \overset{*}{\rightarrow} v$ for all vertices v . If $v \overset{*}{\rightarrow} w$ and {v,w} is an edge of T (denoted by $v \rightarrow w$), we say v is the parent of w and w is a child of v . In a rooted tree each vertex has a unique parent (except the root, which has no parent). Any two vertices v and w in a rooted tree have a unique vertex x , called the least common ancestor of v and w (denoted by x = LCA(v,w)), such that x is on T(v,w) , $x \overset{*}{\rightarrow} v$, and

$x \overset{*}{\rightarrow} w$. The path T(v,w) consists of two parts, a path joining v and x containing descendants of x and ancestors of v , and a path joining x and w containing descendants of x and ancestors of w .

A directed, rooted tree $T = (V,E)$ is an acyclic directed graph with a distinguished vertex r , called the root, such that r has no entering edges and every other vertex has a unique entering edge. Every directed rooted tree may be converted into a rooted tree by ignoring the direction of all edges; every rooted tree may be converted into a directed, rooted tree by directing all edges from parent to child. Thus all the concepts of rooted trees apply to directed, rooted trees. We shall use either rooted trees or directed, rooted trees as appropriate.

In some contexts it is useful to have a numbering of rooted tree vertices such that each vertex has a number larger than its parent. In other contexts it is useful to have a numbering such that each vertex has a number smaller than its parent. The following algorithm generates numberings of these types.

```
procedure ORDER(T,r);
    begin
        procedure SEARCH(v);
            begin
                PRENUMBER(v) := i := i+1;
                for w such that v → w do SEARCH(w);
                POSTNUMBER(v) := j := j+1;
            end SEARCH;
        i := j := 0;
        SEARCH(r);
    end ORDER;
```

Any numbering PRENUMBER(v) generable by ORDER is called a preorder numbering of (T,r) [24] and satisfies the condition that every vertex have a higher number than its parent. Any numbering POSTORDER(v) generable by ORDER is called a postorder numbering of (T,r) [24] and satisfies the condition that every vertex have a lower number than its parent. Procedure ORDER requires O(n) time if implemented properly [24,35]. Note that PRENUMBER(r) = 1 , and POSTNUMBER(r) = n .

Let $\oplus$ be any associative (not necessarily commutative) binary operation, having an identity element $\underline{0}$ such that $\underline{0} \oplus x = x \oplus \underline{0} = x$ for all x . (If $\oplus$ has no identity element, we can create such an element by augmenting the domain of $\oplus$.) Let c(v,w) be an arbitrary function defined on the edges of a rooted tree (T,r) , such that the range of c(v,w) is contained in the domain of $\oplus$. If v and w are any vertices satisfying $v \overset{*}{\rightarrow} w$ and $(v = v_1, v_2), (v_2, v_3) \ldots (v_k, v_{k+1} = w)$ is the path T(v,w) , we define

$$\oplus (v,w) = c(v_1,v_2) \oplus c(v_2,v_3) \oplus \ldots \oplus c(v_k,v_{k+1}) \qquad \text{if } v \neq w ,$$

$$\oplus v = \underline{0} \qquad \text{if } v = w .$$

We are interested in carrying out an intermixed sequence of two types of instructions on a set of rooted trees. Initially the set contains n trees, each tree having only a single vertex. The two types of instructions are:

EVAL(v): return the value of $\oplus (r,v)$, where r is the root of the tree currently containing the vertex v ;

LINK(v,w,x): combine the trees with roots v and w into a single tree with root v by making w a child of v , and let the new edge (v,w) have value c(v,w) = x .

In the succeeding sections we develop an algorithm for carrying cut an intermixed sequence of m EVAL instructions and n-l LINK instructions. Then we apply this algorithm to a variety of problems.'

3. A Basic Algorithm Efficient for Balanced Trees.

In this section we present three algorithms for the function evaluation problem. The first algorithm is extremely simple but has only an O(mn) running time. The second algorithm improves on the first by adding a powerful technique called path compression. The resultant algorithm has an O(m log n) running time and an even faster $O(m\ \alpha(m,n))$ running time for a special class of trees, called balanced trees. The third algorithm achieves an $O(m\ \alpha(m,n))$ bound for all trees but only works for $\oplus$ operations having a suitable kind of inverse.

It is useful to consider a static version of the function evaluation problem. Consider any sequence of m EVAL instructions and n-l intermixed LINK instructions. Let T be the tree defined by the LINK instructions (i.e., (v,w) is an edge of T with value $c(v,w) = x$ if and only if there is a LINK(v,w,x) instruction in the sequence). For each EVAL(v) instruction, let r(v) be the root of the tree containing v at-the time the EVAL(v) instruction is to be executed. Then executing the sequence of instructions is equivalent to computing the value of $\oplus(r(v),v)$ in the tree T for each pair (r(v),v) . (However, the values on the edges of T , and even the shape of T , may depend on the results of the EVAL(v) instructions. Thus it may not be possible to construct T without simultaneously carrying out the evaluations.)

Conversely, let T be any tree of n vertices, with values $c(v,w)$ defined on the edges, and let $\{(v_i,w_i)\}$ be any set of m vertex pairs such that $v_i \overset{*}{\rightarrow} w_i$ in T . We can use the following method to evaluate $\oplus (v_i,w_i)$ for each vertex pair.

Step 1; Number the vertices of T in postorder. Identify each vertex by its number.

Step 2: Sort the pairs (v_i,w_i) in increasing order on v_i .

Step 3:
```
for v := 1 until n do begin
    for w such that $v \rightarrow w$ do LINK(v,w,c(v,w));
    for $(v_i,w_i)$ such that $v_i = v$ do EVAL($w_i$);
end;
```

Step 2 requires O(m) time and O(m) space using a radix sort [27], so the time required to solve this static function evaluation problem is within a constant factor of the time required to solve the dynamic problem defined by Step 3, and the storage space required is O(m) plus the space necessary to execute Step 3.

To solve the dynamic function evaluation problem we use two arrays, f(v) and cc(v) . The value of f(v) is the parent of vertex v in the set of trees so far constructed; f(v) = 0 if v has no parent. The value of cc(v) is c(f(v),v) if v has a parent and 0 otherwise. The following programs implement the LINK and EVAL instructions.

```
INITIALIZE: for v := 1 until n do begin
                f(v) := 0;
                cc(v) := 0;
            end INITIALIZE;

procedure LINK(v,w,x); begin
    f(w) := v;
    cc(w) := x;
end LINK;

procedure EVAL(v); begin
    a := 0;
    w := v;
    while f(w) ≠ 0 do begin
        a := cc(w) ⊕ a;
        w := f(w);
    end;
    EVAL := a;
end EVAL;
```

This method of implementing EVAL and LINK is simple but not very efficient. Consider a sequence of instructions which constructs a non-branching tree of n vertices, and then carries out m evaluations on the vertex farthest from the root. Such an instruction sequence requires $O(mn)$ computing time [13].

To avoid this inefficiency, we use the associativity of $\oplus$. We modify the EVAL instruction so that it not only computes $\oplus(r(v),v)$, but it modifies the tree containing v . Each vertex on the path from $r(v)$ to v is made a child of $r(v)$, and values on the edges are modified to preserve $\oplus(r(v),w)$ values for all vertices w in the same tree as v . Here is a program for this purpose.

```
procedure EVAL(v); begin
    if f(v) = 0 then begin r := v; a := 0 end;
    else if f(f(v)) = 0 then begin r := f(v); a := cc(v) end;
    else begin
        comment first loop reverses f pointers along path from
            v to root;
        x := 0; y := v; r := f(v);
        while f(r) ≠ 0 do begin
            f(y) := x; x := y; y := r; r := f(r);
        end;
        comment first loop ends with r = r(v);
        a := cc(y);
        comment second loop computes ⊕(r(v),v) and modifies
            pointers and values;
        while x ≠ 0 do begin
            y := f(x);
            a := a ⊕ cc(x);
            cc(x) := a; f(x) := r; x := y;
    end end;
    EVAL := a;
end EVAL;
```

We call this method of carrying out an EVAL instruction <u>path compression</u> [3]. As a side effect, this procedure sets r equal to the root of the tree currently containing v . It is easy to prove that this implementation returns the correct value of EVAL(v) for each EVAL instruction. Knuth [11] attributes the path compression idea to Tritter; independently, McIlroy and Morris [20] used it in an algorithm for finding minimum spanning trees. We call each tree defined by the f array an <u>f-tree</u>.

Theorem 1. For any intermixed sequence of $m \geq n$ EV. instructions and n-1 LINK instructions, the running time of the path compression algorithm is $O(m \cdot \max(1, \log_2(n^2/m) / \log_2(2m/n)))$.

Patterson [29] proved Theorem 1 for the case $m = n$; a proof for arbitrary $m > n$ appears in [36]. The bound in Theorem 1 is tight for values of m and n satisfying, for some positive constants c and ϵ , $m < cn$ or $m > cn^{1+\epsilon}$.

Let (T,r) be the rooted tree defined by the n-1 LINK instructions (with no path compression). For any vertex v in T , let d(v) be the number of descendants of v , including v itself. We say T is balanced if $v \to w$ in T implies $2d(w) \leq d(v)$. If T is balanced, the path compression algorithm is faster than indicated by the Theorem 1 bound.

Let the function $A(i,x)$ on integers be defined by $A(0,x) = 2x$ for $x \geq 0$; $A(i,0) = 0$ for $i > 1$; $A(i,1) = 2$ for $i > 1$; $A(i,x) = A(i-1, A(i,x-1))$ for $i \geq 1$, $x > 2$. $A(i,x)$ is a slight variant of Ackermann's function [1]. Let $\alpha(m, n) = \min\{z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log_2 n\}$ where $\lceil x \rceil$ denotes the smallest integer not less than x . For fixed n , the function $\alpha(m,n)$ decreases as m grows.

Theorem 2 [36]. The path compression algorithm runs in $O(m\, \alpha(m,n))$ time if the tree T defined by the LINK instructions is balanced.

Our goal is to devise a function evaluation algorithm which requires $O(m\, \alpha(m,n))$ time for all trees T . We will accomplish this by representing an arbitrary tree as a combination of a balanced tree and a set of paths, and constructing an efficient function evaluation algorithm for paths.

For $\oplus$ operations with a suitable kind of inverse, we can achieve the $O(m\,\alpha(m,n))$ bound for arbitrary trees with much less trouble than in the general case. Suppose that there is a Boolean function $Z(x)$ on the domain of $\oplus$ and another function $I(x)$ from the domain of $\oplus$ into the domain of $\oplus$ satisfying

(i) $Z(x)$ = true implies $y \oplus x = x$ for all y ;

(ii) $Z(x)$ = false implies $Z(I(x))$ = false and $y \oplus x + I(x) = y$ for all y ; and

(iii) $Z(x) = Z(y)$ = false implies $Z(x \oplus y)$ = false .

Then we can modify the implementation of LINK so that the EVAL instructions are performed on a balanced tree, regardless of the structure of T .
For this purpose we need a third array, $d(v)$, which records the number of descendants of each vertex v in the set of trees constructed by the modified LINK procedure. The new version of LINK appears below.

```
procedure LINK(v,w,x); begin
     EVAL(v);
     comment this EVAL instruction, as a side effect, sets r
           equal to the root of the f-tree currently containing v;
     $r_1$ := r;
     EVAL(w);
     $r_2$ := r;
     if Z(x) then cc($r_2$) := x ⊕ cc($r_2$)
     else if d($r_1$) ≥ d($r_2$) then begin
          comment make $r_2$ a child of $r_1$
          d($r_1$) := d($r_1$) + d($r_2$) ;
          f($r_2$) := $r_1$;
          cc($r_2$) := I(cc($r_1$)) ⊕ x ⊕ cc($r_2$);
     end else begin
```

```
        comment make $r_1$ a child of $r_2$;
        $d(r_2) := d(r_1) + d(r_2)$;
        $f(r_1) := r_2$;
        $\underline{cc}(r_2) := x \oplus \underline{cc}(r_2)$;
        $\underline{cc}(r_1) := I(\underline{cc}(r_2)) \oplus \underline{cc}(r_1)$;
end end LINK;
```

We must, in addition, modify EVAL to return the value $\underline{cc}(r) \oplus a$ instead of a .

We call the new implementation of LINK and EVAL path compression with balancing. Suppose this implementation is used and let T' be the tree such that $v \to w$ in T' if and only if v is the first non-zero value assigned to $f(w)$. T' and T differ in that certain parents and children are exchanged, and certain edges in T are missing from T' . It is easy to show that T' is balanced and that LINK adjusts the $\underline{cc}$ array in such a way that all EVAL instructions return correct values [2,13,21]. By Theorem 2, path compression with balancing requires $O(m\,\alpha(m,n))$ time for an arbitrary instruction sequence.

Morris [20] apparently originated the balancing idea. It also appears in [16]. Discussion, analysis, and applications of path compression with balancing appear in [2,3,13,21,36].

We can modify the LINK instruction to save n words of storage if storage is at a premium, The value of $d(v)$ is only of interest when $f(v) = 0$; thus we can store values of $d(v)$ in the f array if we add a Boolean array to indicate whether $f(v)$ represents a pointer or a count of descendants.

For sane applications it is useful to generalize the LINK instruction to allow w to be a vertex other than a tree root. Such an instruction GLINK(v,w,x) can be implemented as follows:

```
procedure GLINK(v,w,x); begin
     Y := EVAL(v);
     comment r is now the root of the f-tree containing v;
     LINK(r,w,y ⊕ x);
end GLINK;
```

4. Two Previous Applications.

This section presents two previous applications of path compression with balancing. The algorithms constructed for these applications will be used in succeeding sections.

The first algorithm computes unions of disjoint sets. We can use the algorithm to represent equivalence relations [25]. Suppse we are given n disjoint sets, each containing one element, and each having a distinguishing **name**. We wish to carry out two types of instructions on these sets. The instruction types are: FIND(x) : return the name of the set containing element x . UNION(A,B) : add the elements in set B to set A , destroying B .

To carry out these instructions, we use four arrays, cc(x) , d(x) , f(x) , and r(A) . We define $x \oplus y = x$ for all x, y , and $I(x) = x$, $Z(x) =$ *false* , for all x . We initialize cc(x) to be the name of the set initially containing x , d(x) to be one, f(x) to be zero, and r(A) to be the single element initially in set A . Then we use path compression with balancing to carry out UNION and FIND instructions as follows:

```
procedure FIND(x);
     W..(x) ;

procedure UNION(A,B);
     LINK(r(A) , r(B) , A) ;
```

The time required for $m > n$ FINDs and n-1 intermixed UNIONs is $O(m\ \alpha(m,n))$. The space required is O(n) . Since $\oplus$ is so simple, the procedures for EVAL and LINK can be shortened somewhat for this special case. This set union algorithm is useful for handling

EQUIVALENCE and COMMON statements in FORTRAN [8,16], finding minimum spanning trees [10,33], and checking flow graphs for reducibility [37].

The second algorithm, due to Aho, Hopcroft, and Ullman [2], computes least common ancestors in a rooted tree. Let (T,r) be a rooted tree and let $\{\{v_i,w_i\}\}$ be a set of m vertex pairs. We wish to compute $LCA(v_i,w_i)$ for each pair. The following method uses the set union algorithm to carry out the computation.

```
Step 1:  Number the vertices of T in postorder. Identify each
         vertex by its number.
Step 2:  Sort the pairs $\{v_i,w_i\}$ so that $v_i \le w_i$ for all i and
         $v_i \le v_j$ for all $i < j$ .
Step 3:  for v := 1 until n do
             initialize a set {v} named v; ,
Step 4:  for w := 1  -  n  %  -
             for $\{v_i,w_i\}$ such that $w_i = w$ do'
                 $LCA(v_i,w_i)$ := FIND($v_i$);
             let u be the vertex such that $u \rightarrow w$ in T;
             UNION(u,w);
         end;
```

We can prove that this algorithm works correctly by using properties of depth-first search; the postorder numbering corresponds to a depth-first search of the tree (T,r) . See [2,34,37,38]. If there are $m \ge n$ vertex pairs, the method requires $O(m\,\alpha(m,n))$ time and $O(m)$ space to compute least common ancestors.

5. Representation of an Unbalanced Tree.

Let (T,r) be a rooted tree. For each vertex v let $d(v)$ be the number of descendants of v in T, and let $f(v)$ be the parent of v in T $(f(r) = 0)$. If $v \rightarrow w$ in T, we say the edge (v,w) is good if $2d(w) \leq d(v)$ and bad if $2d(w) > d(v)$. For each vertex v there is at most one bad edge (v,w). Let $b(r) = 0$ and for $v \neq r$ let $b(v)$ be the unique vertex such that $b(v) \overset{*}{\rightarrow} f(v)$ in T, the path $T(b(v),f(v))$ contains only bad edges, and $f(b(v)) \neq 0$ implies $(f(b(v)),b(v))$ is a good edge. Let TB be the tree with edges $\{(b(v),v) \mid v \neq r\}$. (See Figure 1.)

Theorem 3. TB is balanced.

Proof. For each vertex v, let $d'(v)$ be the number of descendants of v in TB. If $(f(v),v)$ is a bad edge in T, $d'(v) = 1$. Thus $2d'(v) = 2 \leq d'(b(v))$. If $(f(v),v)$ is a good edge in T, then $d'(v) = d(v)$. Thus $2d'(v) = 2d(v) < d(f(v)) < d(b(v)) = d'(b(v))$. In either case $2d'(v) < d'(b(v))$, and TB is balanced. □

For the purposes of the function evaluation problem, we can represent any tree T by the corresponding balanced tree TB and the set of paths defined by the bad edges. Each edge $(b(v),v)$ in TB has an associated value $cb(b(v),v) = \oplus_T(b(v),v)$. Given any vertex pair $(r(v),v)$, we can represent $\oplus_T(r(v),v)$ as

$$\oplus_T(r(v),v) = c(r(v),x) \oplus [\ \oplus_T(x,y)\ ] \oplus c(y,z) \oplus [\ \oplus_{TB}(z,v)\ ]$$

where $(r(v),x)$ is an edge of T, $x \overset{*}{\rightarrow} y$ by a path of bad edges in T, (y, z) is an edge of T, and $z \overset{*}{\rightarrow} v$ in TB.

We can modify LINK to update the tree TB and the set of bad edges, and modify EVAL to compute $\oplus_T(r(v),v)$ using the decomposition above. LINK requires six arrays:.. $\underline{cb}(v)$, $\underline{cc}(v)$, $b(v)$, $f(v)$, $s(v)$, and $d(v)$. For each vertex v, $f(v)$ is the parent of v in T, $cc(v)$ is the value of edge $(f(v),v)$ in T, $s(v)$ is a list of the children of v in T, and $d(v)$ is the number of descendants of v in T. The pointers $b(v)$ represent the tree TB, and $\underline{cb}(v)$ is the value of $\oplus_{TB}(b(v),v) = \oplus_T(b(v),v)$. Initially $\underline{cb}(v) = \underline{cc}(v) = \underset{\sim}{0}$, $b(v) = f(v) = 0$, $s(v) = \emptyset$, and $d(v) = 1$ for each v.

As soon as a LINK(v,w,x) instruction occurs, we can compute the value of $d(w)$. Thus, for each child u of w in T, we can decide whether (w,u) is a good edge or a bad edge. If (w,u) is a bad edge, we use a procedure LINKP to add the edge (w,u) with value $cc(u)$ to the set of bad paths. If (w,u) is a good edge, we find all vertices y such that (u,y) is an edge of TB, and for each such y, we add (u,y) with value $\oplus_T(u,y)$ to TB. The program below implements this computation. The program uses a recursive procedure DFS to find, for each good edge (w,u), the vertices y such that (u,y) is an edge of TB. The program assumes the existence of a procedure LINKP for adding edges to bad paths.

```
procedure LINK(v,w,x); begin
    procedure DFS(y,a);
        for z ∈ s(y) do begin
            b(z) := u;
            cb(z) := a⊕cc(y);
            if 2*d(z) > d(y) then DFS(z, cb(z));
        end DFS;
    d(v) := d(v)+ d(w);
    cc(w) := x;
    add w to s(v);
    for u ∈ s(w) do if 2*d(u) >d(w) then
        LINKP(w, u, cc(u));
    else begin
        c := 0;
        DFS(u,c);
end end LINK;
```

Consider 'this program. The time required for n-1 calls on LINK is O(n) plus the time for all calls on DFS and LINKP . Each recursively nested call on DFS causes b(z) to become non-zero for a new value of z . Thus the total number of calls on DFS is O(n) . The time required for all calls on DFS is proportionalto the total number of calls, so this time is O(n) , and the total time for n-l LINK instructions is O(n) plus the time required for the LINKP instructions.

The following program implements the EVAL instruction. The program assumes the existence of a procedure EVALB which uses path compression on TB to compute path values in' TB . EVALB is identical to the path compression algorithm in Section 3 except for the use of arrays b(v) , cb(v) in place of f(v) , cc(v) . The program also assumes the existence of a procedure EVALP which computes path values on the set of bad paths.

```
procedure EVAL(v); begin
    a := EVALB(v);
    comment as a side effect EVALB(v) sets r equal to the root
        of the tree containing v in the part of TB so far
        constructed;
    x := r;
    if f(x) ≠ 0 then a := EVALP(f(x)) ⊕ cc(x) ⊕ a;
    comment as a side effect EVALP(f(x)) sets r equal to the
        root of the tree containing f(x) in the set of bad
        paths so far constructed;
    a := cc(r) ⊕ a;
    EVAL := a;
end EVAL;
```

Suppose we execute a sequence of m EVAL instructions and $n-1$ intermixed LINK instructions. The EVAL instructions require $O(m)$ time plus the time required for m EVALB and m EVALP instructions. The EVALB instructions carry out path compression on the balanced tree TB and by Theorem 2 require $O(m\,\alpha(m,n))$ time. Thus the entire sequence of instructions requires $O(m\,\alpha(m,n))$ time plus the time for the LINKP and EVALP instructions.

To complete the algorithm we need a way to implement function evaluation on a set of paths; that is, to implement LINKP and EVALP. The next two sections present two ways of doing this so as to achieve an $O(m\,\alpha(m,n))$ time bound. The algorithm of Section 6 is quite simple but is only valid for the special case when $x \oplus y = \max\{x,y\}$. The algorithm of Section 7 works for all operations $\oplus$ but requires certain advance knowledge about the sequence of EVAL and LINK instructions.

6. An Algorithm for the Operation $\max\{x,y\}$.

In this section we assume that $x \oplus y = \max\{x,y\}$. The special properties of $\max\{x,y\}$ allow us to construct a reasonably simple function evaluation algorithm for the set of bad paths. The algorithm uses the disjoint set union algorithm of Section 4, in combination with the following theorem.

Theorem 4. Suppose $x \overset{*}{\rightarrow} y \overset{*}{\rightarrow} z$ in T . Then $\oplus(x,y) \leq \oplus(x,z)$. If $w \rightarrow x \overset{*}{\rightarrow} y \overset{*}{\rightarrow} z$ in T and $\oplus(x,y) = \oplus(x,z)$, then $\oplus(w,y) = \oplus(w,z)$.

Proof. Obvious. □

For any vertex v , consider the set of vertices w such that $v \overset{*}{\rightarrow} w$ by a path of bad edges in T .By Theorem 4 we can partition this set of vertices into a collection of sets S_i such that each S_i consists of the vertices on a path of T , all vertices $w \in S_i$ have the same value of $\oplus(v,w)$ (denoted by $\oplus S_i$), and if $w \in S_i$, $x \in S_j$, $i \neq j$, $w \overset{*}{\rightarrow} x$, then $\oplus S_i < \oplus S_j$.

Our function evaluation method for the bad paths uses the set union algorithm to keep track of the sets S_i and'their associated values $\oplus S_i$. The algorithm uses as the name of the set S_i the vertex $w \in S_i$ such that $x \in S_i$ implies $w \overset{*}{\rightarrow} x$ in T . The algorithm uses two arrays, $\underline{\max}(v)$ and $t(v)$. Initially $\underline{\max}(v) = -\infty$ $(= \underline{0})$ and $t(v) = 0$. As the algorithm proceeds, $\underline{\max}(v) = \oplus S_i$ if v is the name of set S_i , and $t(v) = w$ if v is the name of a set S_i and w is the name of a set S_j such that $v \overset{*}{\rightarrow} x \overset{*}{\rightarrow} w$ implies $x \in S_i \cup S_j$. Initially each vertex v is in a singleton set (v) named v.

The algorithm also needs a mechanism to keep track of the vertex r(v) which is the first vertex on the path containing v in the set of bad paths so far constructed. Two arrays, last(v) and root(v) are used for this purpose. Initially last(v) = root(v) = v for all vertices. As the algorithm proceeds, last(v) the last vertex on the path containing v in the set of bad paths so far constructed, and root(last(v)) is the first vertex on this path. The following programs implement LINKP and EVALP.

```
procedure LINKP(v,w,x); begin
     last(v) := last(w);
     root(last(v)) := v;
     max(w) := x;
     t(v) := w;
     while (t(w) ≠ 0) and (max(t(w)) ≤ x) do begin
          UNION(w,t(w));
          t(w) := t(t(w));
end end LINKP;

procedure EVALP(v); begin
     r := root(last(v));
     EVALP := max(FIND(v));
end EVALP;
```

Execution of n-1 LINKP and m intermixed EVALP instructions requires $O(m\ \alpha(m,n))$ time. Using this implementation in combination with the decomposition method of Section 5 gives an $O(m\ \alpha(m,n))$ time function evaluation method for the special case of $x \oplus y = \max\{x,y\}$. The method requires O(n) storage space.

7. A General Algorithm.

To achieve an $O(m\,\alpha(m,n))$ bound for an arbitrary operation $\oplus$, we must make an assumption about the sequence of EVAL and LINK instructions. We assume that the entire sequence of EVAL and LINK instructions, with the exception of the x parameters in the LINK instructions, is known in advance. Thus we can precompute the trees T and TB, and determine in advance the paths $v \overset{*}{\rightarrow} w$ over which we must compute $\oplus(v,w)$.

We represent the set of bad paths by two sets of balanced trees, TR and TL . Consider any bad path and suppose its vertices are numbered in postorder from 1 to k. Let v and w be vertices on this path for which we want the value of $\oplus(v,w)$. We compute $\oplus(v,w)$ as $\oplus(v,w) = [\oplus(v,u)] \oplus [\oplus(u,w)]$, where $u = (2j+1)2^i$ is the vertex with largest i in the range $w \leq u \leq v$.

To compute' $\oplus(u,w)$, we use a forest TR . TR is the set of trees with vertices 1 through k such that the father of vertex $(2j+1)2^i$ is $(j+1)2^{i+1}$. (See Figure 2.) The value of an edge (x,y) in TR is $\oplus_T(x,y)$. TR is a set of balanced trees numbered in postorder. We can use path compression in TR to compute $\oplus_T(u,w) = \oplus_{TR}(u,w)$.

To compute $\oplus(v,u)$, we use a forest TL . TL is the set of trees with vertices 1 through k such that the father of vertex $(2j+1)2^i$ is $j2^{i+1}$. (See Figure 3.) The value of an edge (x,y) in TL is $\oplus_T(y,x)$. TL is a set of balanced trees numbered in preorder. If we define $x \oplus' y = y \oplus x$, then $\oplus_T(v,u) = \oplus'_{TL}(u,v)$ for any pair of vertices (v,u) such that $u \overset{*}{\rightarrow} v$ in TL .

The idea we want to use is to compute $\oplus_T(v,u) = \oplus'_{TL}(u,v)$ for appropriate pairs (v,u) by using path compression in TL . This idea does not work directly, however, because compressing a path $u_1 \overset{*}{\rightarrow} v_1$ in TL may cause a later pair (u_2,v_2) to become unrelated in TL . (See Figure 4.)

To solve this problem, we use the fact that we can precompute the trees T , TB , TL , and TR and the paths over which we wish to evaluate. We reorder the paths in TL so that path compression will work, and we symbolically compute values for each appropriate path in T , TB , TL , and TR. This symbolic computation works as follows. We construct a unique identifier e for each edge (v,w) of T , with $f(e) = v$, $g(e) = w$. For each path $T(x,z)$, the value of which we wish to compute as $\oplus_T(x,z) = [\oplus_T(x,y)] \oplus [\oplus_T(y,z)]$, we also construct a unique identifier e , with $f(e) = x$, $g(e) = z$, $p,(e) = e_1$, $p,(e) = e_2$, where e_1 identifies the path $T(x,y)$ and e_2 identifies the path $T(y,z)$.

After constructing identifiers to represent the entire computation, we reorder the identifiers in a way consistent with the order of the EVAL and LINK instructions. Then we read through the identifiers and the EVAL and LINK instructions, carrying out the computation. The algorithm, presented below, has six steps.

Step 1: Initialize all variables. Construct T . Compute $d(v)$ for each vertex of T .

Step : Construct TB , TR , TL . For each EVAL(v) instruction, find the vertex r such that $\text{EVAL}(v) = \oplus(r,v)$ and construct identifiers e_2 , e_3 , e_4 such that

$$\oplus(r,v) = [\oplus'_{TL}(f(e_2),r)] \oplus [\oplus_{TR}(f(e_2),g(e_2))]$$
$$\oplus\, c(f(e_3),g(e_3)) \oplus [\oplus_{TB}(f(e_4),g(e_4))] .$$

Use path compression to symbolically compute values for appropriate paths in TB and TR .

Step 3: Sort the pairs $(f(e_2),r)$ in decreasing order on $d(f(e_2))$.

Step 4: Use path compression to symbolically compute values for appropriate paths in TL. For each pair $(r,f(e_2))$, construct an identifier e_1,

Step 5: Sort the identifiers e in increasing order on d(f(e)) , breaking ties in decreasing order on d(g(e)) .

Step 6: Process the identifiers and the LINK and EVAL instructions in order, carrying out the actual evaluation.

This algorithm hinges upon the symbolic computation and the reordering of identifiers so that the actual computation proceeds in an order consistent with the order of the EVAL and LINK instructions; the x values occuring in the LINK instructions may depend on the results of previous EVAL instructions. Since $d(v) \geq d(w)$ implies $v = w$ or $\neg(w \overset{*}{\rightarrow} v)$ in T , the sorting in Step 3 guarantees that the path compression in Step 4 will work. Furthermore Step 5 sorts the identifiers e so that $p_1(e)$ and $p_2(e)$, if defined, precede e , and if e_1 precedes e_2 and e_1 and e_2 identify edges of T , then the LINK instruction corresponding to e_1 precedes the LINK instruction corresponding to e_2 .

If all the x values in LINK instructions are known ahead of time, as in the static evaluation problem mentioned in Section 3, we can dispense with Steps 5 and 6 and the symbolic computation and carry out all the evaluations directly. We must still reorder the evaluations on the forest TL using Step 3.

The algorithm uses thirteen arrays and one array of lists. For each vertex v , et(v) is the identifier of the edge from the parent of v to v in T . Arrays eb , er , el similarly represent TB , TR , TL . For each vertex v , d(v) is the number of descendants of v in T , and s(v) is a list, of children of v in T . Arrays root(v) and last(v) are used to find the first vertex on each bad path as described in Section 6. For each vertex v , h(v) is the number of vertices (including v) from v to the end of the bad path containing v . The array c(e) is used to store values computed for the identifiers in Step 6. For each vertex v , the algorithm constructs a dummy identifier e with f(e) = g(e) = v .

```
Step  :  for v := 1 until n do begin
              et(v) := eb(v) := er(v) := el(v) := 0;
              f(v) := g(v) := root(v) := last(v) := v;
              d(v) := h(v) := 1;
              s(v) := ∅;
              c(v) := 0;
         end;
         k := n;
         ident := list := ∅;
         for i := 1 until m+n-1 do
              if instruction i is LINK(v,w,x) then begin
                   d(v) := d(v)+ d(w);
                   k := k+1;
                   et(w) := k;
                   f(k) := v; g(k) := w;
                   s(v) := s(v) ∪ {w};
              end Step 1;
```

```
Step 2:   for i := 1 until m+n-1 do
              if instruction i is LINK(v,w,x) then begin
                  if $2^*d(v) > d(w)$ then LINKP(v,w)
                  else DFS(w);
              end else begin
                  let instruction i be EVAL(v);
                  EVAL(v, eb, $e_4$) ;
                  if et(f($e_4$(i))) = 0 then $e_3$(i) = v
                  else $e_3$(i) := et(f($e_4$(i)));
                  EVAL(f($e_3$(i)), er, $e_2$);
                  r := root(last(f($e_2$(i))));
                  list := list $\cup$ {(f($e_2$(i)), r, i)};
              end Step 2;

          procedure DFS(x);
              for $y \in s(x)$ do begin
                  if f(et(y)) = w then eb(y) := et(y)
                  else begin
                      k := k+1;
                      f(k) := f(eb(x)); g(k) := y; $p_1$(k) := eb(x);
                          $p_2$(k) := et(y);
                      eb(y) := k;
                      ident := ident $\cup$ {k};
                  end;
                  if $2^*d(y) > d(x)$ then DFS(y);
              end DFS;
```

An examination of Figures 2 and 3 verifies the following facts, which form the basis for procedure LINKP(v,w) below. Let $h(v) = (2j+1)2^i$ be the number of vertices (including v) from v to the end of the bad path containing v . Then $\{(g \circ \underline{el})^{\ell}(w) \mid 0 \leq \ell \leq i-1\}$ is the set of children of v in TR . If $i = 0$, w is the parent of v in TL ; if $i > 0$ and $j > 0$, $(g \circ \underline{el})^{i}(w)$ is the parent of v in TL ; and if $i>0$ and $j = 0$, v has no parent in TL .

```
procedure LINKP(v,w) ; begin
    h(v) := h(w)+1;
    j := h(v);
    if j is odd then el(v) := et(w)
    else begin
        er(w) := et(w);
        j := j/2;
        z := el(w);
        while j is even do begin
            k := k+1;
            f(k) := v; g(k) := g(z);
            $p_1$(k) := er(f(z)) ; $p_2$(k) := z;
            er(g(z)) := k;
            ident := ident $\cup$ {k};
            z := el(g(z));
        end
        if g(z) $\neq$ 0 then begin
            k := k+1;
            f(k) := v; g(k) := g(z);
            $p_1$(k) := er(f(z)); $p_2$(k) := z;
            ident := ident $\cup$ {k};
            el(v) := k;
end end end LINKP;
```

```
procedure EVAL(v,e,ei);
    if e(v) = 0 then ei(i) := v
    else if e(f(e(v))) = 0 then ei(i) := e(v)
    else begin
        X := 0; y := e(v);
        while e(f(y)) ≠ 0 do begin
            e(g(y) ) := x; x := y; y := e(f(y));
        end;
        while x ≠ 0 do begin
            k := k+1;
            f(k) := f(y); g(k) := g(x);
            $p_1$(k) := e(f(x)); $p_2$(k) := x;
            X := e(g(x));
            e($p_2$(k)) := k;
            ident := ident ∪ {k};
        end;
        ei(i) := k;
    end EVAL;
```

Step 3: Using a radix sort, order the triples (z,r,i) on list in decreasing order on d(z) .

```
Step 4: for (z,r,i) ∈ list do
            if z = r then e,(i) = r
            else if z = g(el(r)) then e,(i) := el(r)
            else begin
                X := 0; y := el(r);
                while el(g(y)) ≠ 0 do begin
                    el(f(y)) := x; x := y; y := el(g(y));
                end;
                while x ≠ 0 do begin
                    k := k+1
                    g(k) := g(y); f(k) := f(x);
                    $p_2$(k) := el(g(x)); $p_1$(k) := x;
                    X := el(f(x));
                    el($p_1$(k)) := k;
                    ident := ident ∪ {k};
                end;
                e,(i) := k;
        end Step 4;
```

Step 5: Using a two-pass radix sort, order the identifiers e on ident in increasing order on d(f(e)) , breaking ties in decreasing order on d(g(e)) .

```
Step 6: for i := 1 until m+n-1 do
            if instruction i is LINK(v,w,x) then begin
                c(et(w)) := x;
                for j ∈ ident such that f(j) = v do
                    c(j) := c(p_1(j)) ⊕ c(p_2(j));
            end else begin
                let instruction i be EVAL(v);
                return c(e_1(i)) ⊕ c(e_2(i)) ⊕ c(e_3(i)) ⊕ c(e_4(i))
                    as the result of instruction i;
            end Step 6;
```

Initialization and construction of T , TB , TR , TL require O(n) time. The path compressions and symbolic computations in Steps 2 and 4 require O(m $\alpha(m,n)$) time. Step 3 requires O(m) time and space, and Step 5 requires O($\alpha(m,n)$) time and space, since O(m $\alpha(m,n)$) identifiers are constructed. Step 6 requires O(m $\alpha(m,n)$) time. Thus the entire algorithm requires O(m $\alpha(m,n)$) time and space. The corresponding algorithm for the static function evaluation problem (omitting Steps 5 and 6 and the symbolic computations) requires O(m $\alpha(m,n)$) time and O(m) space. It is possible to save storage space in the algorithm for the dynamic function evaluation problem by delaying evaluation on TB and TR until Step 6 when the values are actually known and using symbolic computation only on TL . However, this saves at most a constant factor in running time and storage space.

8. Verifying a Minimum Spanning Tree.

This section presents a simple, direct application of the function evaluation algorithm. Let T be an arbitrary tree and let $\oplus$ be a commutative, associative operation. Let each edge (x,y) of T have an associated value $c(x,y)$ which is in- the domain of $\oplus$. For any two vertices v and w in T, let $\oplus(v,w) = c(v_1, v_2) \oplus c(v_2,v_3) \oplus \ldots \oplus (v_k,v_{k+1})$, where $T(v,w) = (v_1,v_2), (v_2,v_3), \ldots, (v_k,v_{k+1})$. The problem we solve is this: given a set of m pairs of vertices $\{\{v_i,w_i\}\}$, compute $\oplus(v_i,w_i)$ for each pair.

Our algorithm, an application of the least common ancestors algorithm of Section 4 and of the function evaluation algorithm, appears below.

Step 1: Pick an arbitrary vertex r of T and convert T into a rooted tree (T,r).

Step 2: For each pair $\{v_i,w_i\}$, compute $x_i = LCA(v_i,w_i)$ using the algorithm of Section 4.

Step 3: Compute $\oplus_T(x_i,v_i)$, $\oplus_T(x_i,w_i)$ for each pair $\{v_i,w_i\}$ using the static version of the function evaluation algorithm and combine the answers to give $\oplus(v_i,w_i)$ for each pair.

This algorithm requires $O(m\,\alpha(m,n))$ time and $O(m)$ storage space.

The algorithm has several interesting applications. Suppose $c(v,w)$ is a real value representing the cost of edge (v,w), and let $x \oplus y = x+y$. Then the algorithm computes the total cost of each of a set of m paths $T(v_i,w_i)$. In this case $\oplus$ has an inverse and we can use path compression with balancing, as described in Section 3, to carry out Step 3. See [2] for a similar solution to a problem requiring computation of depths in rooted trees.

Suppose $c(v,w)$ is a real value, and let $x \oplus y = \min\{x,y\}$. Then the algorithm computes the minimum value along each path $T(v_i,w_i)$. In this case we can use the algorithm of Section 6 to carry out Step 3. This problem arises when determining the minimum cut (or maximum flow) between given pairs of vertices in an undirected graph with edge weights. Gomory and Hu [22] have given a method for constructing, for any undirected graph G with edge weights, a tree T such that

(i) T has the same vertices as G , and

(ii) the value of the minimum cut between any pair of vertices v and w in G is equal to the minimum edge value on the path $T(v,w)$.

Thus, we can use the algorithm above to compute minimum cut values for a set of vertex pairs, assuming that the cut tree T is given.

Suppose $G = (V,E)$ is a graph with real values $c(v,w)$ on its edges and $T = (V,E')$ is a spanning tree of G . We say T is a <u>minimum spanning tree</u> if $\sum_{(v,w)\in E'} c(v,w)$ is aminimumamong all s-panning trees of G . We wish to test whether T is a minimum spanning tree. The following well-known theorem allows us to apply the algorithm above.

<u>Theorem 5</u>. T is minimum if and only if, for each edge $(v,w) \in E-E'$, $c(v,w) \geq \max\{c(x,y) \mid (x,y) \text{ is on } T(v,w)\}$.

Thus, if G has m edges, we can test whether 'I' is minimum in $O(m\ \alpha(m,n))$ time by computing $\oplus_T(v,w)$ for each non-tree edge (v,w) using the algorithm above with $x \oplus y = \max\{x,y\}$ and applying the test of Theorem 5. This result is interesting because the best known algorithms for actually finding a minimum spanning tree [10,33,40] require $O(m \log \log n)$ time.

9. Finding Dominators.

Several interesting graph-theoretic problems arise in the study of global flow analysis and optimization of computer code. This section discusses a problem of this type. A flow graph (G,r) is a directed graph with a distinguished start vertex r such that there is a path from r to each node in G . Vertex v dominates vertex w in flow graph (G,r) if $v \neq w$ and every path from r to w contains v . Vertex v is the immediate dominator of w , denoted $v = \underline{idom}(w)$, if v dominates w and every other dominator of w also dominates v . By convention $\underline{idom}(r) = 0$.

Theorem 6. Every vertex of a flow graph (G,r) except r has a unique immediate dominator. The edges $\{(\underline{idom}(w),w) \mid w \in V-\{r\}\}$ form a-directed tree rooted at r , called the dominator tree of (G,r) , such that v dominates w if and only if $v \overset{*}{\rightarrow} w$ in the dominator tree.

Proof. See [6]. □

We wish to construct the dominator tree of an arbitrary flow graph (G,r) . Reference [6] describes uses of the dominator tree in global code optimization. Aho and Ullman [6] and Purdom and Moore [30] have given simple O(nm) time algorithms. Reference [34] gives a more complicated O(n log n + m) time algorithm and [38] gives a simplified version of this algorithm. Here we use extensions of the ideas in [34,38] to develop a new algorithm which uses path compression to achieve an $O(m\ \alpha(m,n))$ time bound.

We need ne new concept, that of a depth-first spanning tree. Let (G,r) be a flow graph with G = (V,E) , and let (T,r) be a directed spanning tree of G rooted at r , with T = (V,E') . Let T have a postorder numbering and assume that vertices of T are identified by number. (T,r) with the given numbering is a *depth-first spanning tree* (DFS tree) of (G,r) if the edges of E-E' can be partitioned into three sets:

(i) a set of edges (v,w) with $v \xrightarrow{*} w$ in T , called *forward edges;*

(ii) a set of edges (v,w) with $w \xrightarrow{*} v$ in T , called *cycle edges;*

(iii) a set of edges (v,w) with neither $v \xrightarrow{*} w$ nor $w \xrightarrow{*} v$, but with $w > v$, called *cross edges*.

A DFS tree is so named because it can be generated by starting at r and carrying out a depth-first search of G . A properly implemented algorithm requires O(m) time to carry out such a search [35], using a set of adjacency lists [4,26] to represent G . The search generates T , numbers the vertices in postorder, and partitions the edges into tree edges, forward edges, cycle edges, and cross edges. Henceforth we assume that (T,r) is a DFS tree of G and that vertices are identified by number.

Theorem 7. If $v > w$, any path from v to w in G must contain a common ancestor of v and w in T .

Proof. See [34,35]. □

We will calculate idom(w) for each vertex w by processing the vertices in order, from smallest to largest. For $0 \le k \le n$, let $G_k = (V, \{(v,w) \mid (v,w) \in E \text{ and } w \le k\})$. $G_0 = (V,\emptyset)$; $G_n = G$. For $0 \le k \le n$ and $1 \le w \le n$ let $\underline{dom}(k,w) = \max\{v \mid$ there is a path from v to w in $G_k\}$. It is clear by examining T that $\underline{dom}(k,w) \ge \max\{k,w\}$ for all k and w , and $\underline{dom}(k,w) > k$ if $k \ge w$ and $w < n$. Furthermore, it follows from Theorem 7 that $\underline{dom}(k,w) \overset{*}{\rightarrow} w$ for all k and w . We prove some more facts about $\underline{dom}(k,w)$ which enable us to calculate it.

Theorem 8. $\underline{dom}(k,k) = \max\{\underline{dom}(k-1,v) \mid (v,k)$ is an edge$\}$ if $k < n$.

Proof. Obvious. Cl

For $0 \le k \le n$, $1 \le w \le n$, $k \ge w$, let a(k,w) be the smallest ancestor of w larger than k . Define $c(v,w) = \underline{dom}(w,w)$ for all edges $(v,w) \in T$, and $x \oplus y = \max\{x,y\}$.

Theorem 9. If $k>w$, $\underline{dom}(k,w) = \oplus(a(k,w),w)$.

Proof. Clearly there is a path from $\oplus(a(k,w),w)$ to w in G_k , so $\underline{dom}(k,w) \ge \oplus(a(k,w),w)$. We prove by induction on k that $k \ge w$ implies $\underline{dom}(k,w) \le \oplus(a(k,w),w)$. The hypothesis is clearly true for $k = w$. Suppose the hypothesis is true for some k and consider the path in G_{k+1} from $\underline{dom}(k+1,w)$ to w . If this path does not contain k+1 , then $\underline{dom}(k+1,w) = \underline{dom}(k,w) \le \oplus(a(k,w),w) \le \oplus(a(k+1,w),w)$ by the induction hypothesis. If this path does contain k+1 , then $k+1 > w$

implies the path from k+1 to w in G_{k+1} contains a common ancestor of k+1 and w , which must k+1 . Then $\underline{\text{dom}}(k+1,w) = \underline{\text{dom}}(k+1, k+1) \leq \oplus(a(k+1,w),w)$. ☐

Theorems 8 and 9 allow us to compute $\underline{\text{dom}}(w,w)$ for each vertex $w < n$ by using path compression. We simply execute the following loop.

```
for w := 1 until n-1 do
     begin
          dom(w,w) := [max{v | (v,w) ∈ E and v > w)]
                              ⊕ [max{EVAL(v) | (v,w) ∈ E and v < w}];
        --. let v → w in T;
          LINK(v,w,dom(w,w));
     end;
```

The next theorem shows how to use the values $\underline{\text{dom}}(w,w)$ to compute immediate dominators.

Theorem 10. Let $v \neq n$. If no vertex u satisfies $u \overset{*}{\rightarrow} v$, $\underline{\text{dom}}(u,u) > \underline{\text{dom}}(v,v) > u$, then $\underline{\text{idom}}(v) = \underline{\text{dom}}(v,v)$. Otherwise, let u be the smallest vertex such that $u \overset{*}{\rightarrow} v$ and $\underline{\text{dom}}(u,u) > \underline{\text{dom}}(v,v) > u$. Then $\underline{\text{idom}}(v) = \underline{\text{idom}}(u)$.

Proof. Clearly no vertex except $\underline{\text{dom}}(v,v)$ on the tree path from $\underline{\text{dom}}(v,v)$ to v can dominate v . Suppose no vertex u satisfies $u \overset{*}{\rightarrow} v$, $\underline{\text{dom}}(u,u) > \underline{\text{dom}}(v,v) > u$. Consider any path from n to v . Let x be the last vertex on the path with $x > \underline{\text{dom}}(v,v)$. If there is no such vertex then $\underline{\text{dom}}(v,v) = n$ and $\underline{\text{dom}}(v,v)$ dominates v . Otherwise, let y be the first vertex following x on the path with $\underline{\text{dom}}(v,v) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} v$. All vertices z between x and y on the path must satisfy $z < y$ by Theorem 7 and the choice of x and y. Thus $\underline{\text{dom}}(y,y) \geq x > \underline{\text{dom}}(v,v)$. By the hypothesis this means $y = \underline{\text{dom}}(v,v)$ ($y = v$ is impossible since then there is a path from $x \geq \underline{\text{dom}}(v,v)$ to v in G_v). Thus $\underline{\text{dom}}(v,v)$ lies on the path from n to v . Hence $\underline{\text{dom}}(v,v)$ dominates v , and $\underline{\text{idom}}(v) = \underline{\text{dom}}(v,v)$.

Conversely, suppose some vertex u satisfies $u \overset{*}{\rightarrow} v$, $\underline{\text{dom}}(u,u) > \underline{\text{dom}}(v,v) > u$. Pick the minimum such vertex u . Clearly no vertex which does not dominate u can dominate v . Thus every vertex which dominates v dominates u . Now we need only show that $\underline{\text{idom}}(u)$ dominates v . Consider any path from n to v . Let x be the last vertex on this path satisfying $x \geq \underline{\text{idom}}(u)$. If there is no such x , then $\underline{\text{idom}}(u) = n$ dominates v . Otherwise, let y be the first vertex following x on the path and satisfying $\underline{\text{idom}}(u) \overset{*}{\rightarrow} y \overset{*}{\rightarrow} v$. All vertices z between x and y on the path must satisfy $z < y$ by Theorem 7 and the choice of x and y . Thus $\underline{\text{dom}}(y,y) \geq x > \underline{\text{idom}}(u) \geq \underline{\text{dom}}(u,u)$. Hence y cannot lie between $\underline{\text{idom}}(u)$ and u , or equal u , since otherwise $\underline{\text{idom}}(u)$ would not dominate u . Also y cannot lie between u and v by the choice of u . Furthermore $y \neq v$ since $y = v$ implies there is a path from $x > \underline{\text{dom}}(u,u) > \underline{\text{dom}}(v,v)$ to v in G_v . The only remaining possibility is $y = \underline{\text{idom}}(u)$. Thus $\underline{\text{idom}}(u)$ lies on the path from nto v, and $\underline{\text{idom}}(u)$ dominates v . □

We use the set union algorithm and Theorem 10 to compute immediate dominators. First we sort the pairs $(\underline{\text{dom}}(v,v),v)$ so that (u_1,v_1) precedes (u_2,v_2) if and only if $u_1 < u_2$ or $u_1 = u_2$ and $v_1 > v_2$.

We use a two-pass radix sort, which requires $O(n)$ time. This ordering has the feature that if (u_1,v_1) precedes (u_2,v_2) and $v_1 < v_2$, then $u_1 < u_2$. Next we apply the set union algorithm. Initially each vertex v is in a singleton set containing only v and named v. As the algorithm examines the pairs in order, vertex v will be in the set named x if and only if x is the smallest vertex such that $x \overset{*}{\rightarrow} v$ and the pair (dom(x,x),x) has not yet been examined. Here is the computation.

```
Step 1:  for each pair (dom(x,x),x) in order do begin
             let u → x in T;
             UNION(FIND(u),x);
             if FIND(dom(x,x)) ≠ FIND(x) then
                 begin idom(x) := FIND(x); flag(x) := true end
             else idom(x) := dom(x,x); flag(x) := false end;
         end;

Step 2:  for i := n-1 step -1 until 1 do if flag(i) then
         idom(i) := idom(idom(i));
```

The first loop constructs a set of pointers in array idom(v) using Theorem 10. The second loop uses these pointers to compute dominators. The total time to compute dom(w,w) values and dominator values is $O(m\,\alpha(m,n))$ using the function evaluation algorithm of Section 6. The storage space necessary is $O(m)$.

10. Lower Bounds.

An interesting theoretic problem is to determine whether the $O(m\,\alpha(m,n))$ bound is tight, for either the general function evaluation problem or for interesting special cases. Perhaps surprisingly in light of the dearth of lower bound results, we can prove that the $O(m\,\alpha(m,n))$ bound is tight to within a constant factor, for various cases of the function evaluation problem.

To prove lower bounds, we use the following formal setting. Let (T,r) be a rooted tree on n vertices, with edge values selected from the domain of an associative binary operation $\oplus$. Given a set of m pairs (v_i,w_i) of related vertices, we desire a lower bound on the number of $\oplus$ operations required to compute $\oplus(v_i,w_i)$ for all m pairs.

A computation sequence for the pairs (v_i,w_i) is a list of assignments of the form $x := y \oplus z$, where y and z are either edges of T or are variables which have occurred on the left side of some previous assignment, and each variable x occurs on the left side of only one assignment. Corresponding to each pair (v_i,w_i) is a variable x_i such that, for all substitutions of values for the edges, the variable x_i is assigned value $\oplus(v_i,w_i)$ when the computation sequence is carried out. We prove that, in the worst case, any computation sequence for m pairs must be of length at least $k\,m\,\alpha(m,n)$, for some constant k . We prove this result for various interesting operations $\oplus$. In some cases the lower bound holds even if we allow a second operation to occur in the computation sequence.

Notice that our computation model allows only straightline programs, with no branching. In certain cases the lower bound does not hold if we allow branching. In other cases, we conjecture the lower bound still holds but cannot prove it.

Consider any computation sequence, and let x be any variable which occurs in the sequence. Corresponding to x is an expression of the form $x = c(x_1,y_1) \oplus \ldots \oplus c(x_k,y_k)$ which gives the value computed for x as a function of the edge values. Suppose the computation sequence satisfies the following property.

(*) If $x = c(x_1,y_1) \oplus \cdot * \cdot \oplus (x_k,y_k)$ is the expression for any variable x , then $(x_1,y_1), \ldots, (x_k,y_k)$ all lie on $T(v_i,w_i)$ for some pair (v_i,w_i).

Order the pairs (v_i,w_i) so that if (v_i,w_i) precedes (v_j,w_j) in the ordering and vi $\neq v_j$, then $\neg(v_i \xrightarrow{*} v_j)$ in T . For each variable x in the computation sequence, assign the corresponding expression to the first pair (v_i,w_i) in the ordering such that every edge in the expression is on $T(v_i,w_i)$.

Now associate with T and with the pairs (v_i,w_i) a directed graph G^* and a cost C as follows. Initially $G^* = T$. Process the pairs (v_i,w_i) in the order defined above. To process a pair (v_i, w_i) , let $v_i = x_1 \to x_2 \to \ldots \to x_{k+1} = w_i$ be the path in T from v to w . Add to G^* each edge (x_{j_1}, x_{j_2}) with $j_1 < j_2$ which is not already present in G^* . Let the cost of the pair (v_i,w_i) be $\ell_i - 1$, where ℓ_i is the length of the shortest path from v_i to w_i in G^* (before the new edges for (v_i,w_i) are added). Let the cost C be the total cost of all pairs (v_i,w_i) .

Theorem 11. The cost C is a lower bound on the length of any computation sequence satisfying (*).

proof. Consider a computation sequence satisfying (*). Assign the expressions computed by the computation sequence to pairs (v_i,w_i) as described above. Process the pairs (v_i,w_i) in the order defined above, a6 follows. Initialize $G^* = T$. For each pair (v_i,w_i), add edges to G^* a6 described above, and compute the value of all expression6 assigned to the pair (v_i,w_i).

For each pair (v_i,w_i), the number of $\oplus$ operations required to compute all expressions assigned to the pair (v_i,w_i) is at least as great a6 the cost of (v_i,w_i). To prove this, suppose the expression for $\oplus(v_i,w_i)$ is computed as

$$\{\oplus\{c(x_{j1},y_{j1}) \oplus c(x_{j2},y_{j2}) \oplus \cdots \oplus c(x_{jk_j},y_{jk_j})\} \mid 1 \le j \le p\} ,$$

where each expression inside the outer sum is assigned to a pair previous to (v_i,w_i). We can order the expressions so that for some $r \le p$ and for some $q_2,q_3,\ldots,q_r$,

$$v_i = x_{11} \overset{*}{\rightarrow} y_{1k_1} = x_{2q_2} \overset{*}{\rightarrow} y_{2k_2} = x_{3q_3} \overset{*}{\rightarrow} y_{3k_3} \overset{*}{\rightarrow} \cdots \overset{*}{\rightarrow} y_{rk_r} = w_i .$$

Then (x_{11},y_{1k_1}), (x_{2q_2},y_{2k_2}), $\ldots$, (x_{rq_r},y_{rk_r}) are edges of G^* before pair (v_i,w_i) is processed, and the number of expressions combined to compute $\oplus(v_i,w_i)$ is no fewer than ℓ_i-1, where ℓ_i is the length of the shortest path from v_i to w_i in G^* before (v_i,w_i) is processed. Thus $C = \sum_{i=1}^{m} \ell_i-1$ is a lower bound on the total length of the computation sequence. $\square$

Now we apply the very general lower bound result of [36], which states:

Theorem 3.2 [36]. There is a constant k such that, for all m and n with $m \geq n$, there is a tree T of n vertices and a sequence of m pair6 (v_i, w_i) for which the cost of G^* is at least $k m \alpha(m,n)$.

We have immediately; ,

Theorem 13. For any $m \geq n$, there is a static function evaluation problem for m pairs on a tree with n vertices such that any computation sequence satisfying (*) has length at least $k m \alpha(m,n)$.

The power of Theorem 13 lies in the fact that for many interesting operation6 $\oplus$, any expression which does not satisfy (*) is useless in any computation sequence; thus any minimum-length computation sequence must satisfy (*). Such operations include the following:

(1) Function composition over a suitably general function space.

(2) String concatenation.

(3) Set union. The lower bound holds even if set intersection is also allowed as an operation.

(4) Maximum over real numbers. The lower bound holds even if minimum is also allowed.

(5) Boolean "and" over the domain [true, false] . The lower bound holds even if Boolean "or" is also allowed.

We prove the lower bound for (5). Consider any computation sequence which uses $\wedge$ (and) and $\vee$ (or) to compute $\wedge (v_i, w_i)$ for

for a sequence of **m** pair6 (v_i, w_i) . Such a computation sequence corresponds to a <u>monotone Boolean circuit</u> for computing $\wedge (v_i, w_i)$ for all pair6 (v_i, w_i) . See [28,31] for lower bounds on the sizes of restricted kind6 of Boolean circuits for other functions.

Let E be any expression involving $\wedge$ and $\vee$. Let $\equiv$ denote truth value equivalence. Convert E into disjunctive normal form

$$E \equiv E_D = (x_{11} \wedge x_{12} \wedge \ldots \wedge x_{1i_1}) \vee \ldots \vee (x_{k1} \wedge \ldots \wedge x_{ki_k})$$

with $i_1 \leq i_j$ for $1 \leq j \leq k$. Then E is equivalent to a conjunction, namely $E \equiv (x_{11} \wedge x_{12} \wedge$ ● ⊠⊠✌☞●)($_1$) , if and only if each variable $x_{1\ell}$ in the first clause occur6 in all the clauses. It follows that if $E_1 \vee E_2 \equiv (x_1 \wedge x_2 \wedge x_3 \wedge \ldots \wedge x_i)$, then either $E_1 \equiv (x_1 \wedge x_2 \wedge \ldots \wedge x_i)$ or $E_2 \equiv (x_1 \wedge x_2 \wedge \ldots \wedge x_i)$.

Similarly, let E be any expression and convert E into conjunctive normal form

$$E \equiv E_C = (y_{11} \vee y_{12} \vee \ldots \vee y_{1i_1}) \wedge \ldots \text{ A } (y_{k1} \vee \ldots \vee y_{ki_k})$$

with $i_1 \leq i_j$ for $1 < j < k$. Then E is equivalent to a conjunction, **namely** $E \equiv (y_{11} \wedge y_{21} \wedge \ldots \wedge y_{\ell 1})$, if and only if $i_j = 1$ for $1 \leq j < \ell$ and each clause $(y_{j1} \vee \ldots \vee y_{ji_j})$ for $1 \leq j \leq k$ contains 6ome variable y_{P1} with $1 \leq p \leq \ell$. Thus if $E_1 \wedge E_2 \equiv (y_1 \wedge y_2 \wedge \ldots \wedge y_i)$, then $E_1 \equiv (y_1 \wedge y_2 \wedge \ldots \wedge y_k)$ **and** $E_2 \equiv (y \text{ A} \cdot_j \cdot \wedge y_i)$ for some j , k satisfying $1 \leq j \leq k+1 \leq i$. (Achieving this **representation** may require renumbering the variables.)

Now consider any computation sequence which uses $\wedge$ and $\vee$ to compute $\wedge(v_i,w_i)$ for a set of m pairs (v_i,w_i) . Let E_i be the expression computed for $\wedge(v_i,w_i)$. By the remarks above a subsequence of the computation sequence must compute a sequence of expressions $E_{i1}, E_{i2}, \ldots, E_{ik} = E_i$ such that each E_{ij} is either an edge of T or is equivalent to $E_{ip} \wedge E_{iq}$ for some $p,q < j$. Delete all assignments from the computation sequence except those corresponding to expressions E_{ij} . The resultant sequence still computes $\wedge(v_i,w_i)$ for all pairs (v_i,w_i) and also satisfies (*). Thus by Theorem 13 we have:

Corollary 1. For any $m \geq n$, there is a rooted tree T of n vertices and a set of m pairs (v_i,w_i) of related pairs such that any computation sequence using $\wedge$ and $\vee$ to compute $\wedge(v_i, w_i)$ for all pairs has length at least $km\alpha(m,n)$ for some constant k .

The lower bounds for operations (3) and (4) follow from Corollary 1; lower bounds for operations (1) and (2) are immediate from Theorem 13.

Several plausible lower bounds remain conjectures. We leave them as open problems.

(1) Prove a $km\alpha(m,n)$ lower bound for any operation $\oplus$ which has an inverse.

(2) Prove a $km\alpha(m,n)$ lower bound for computing $\bigvee_{i=1}^{m} [\wedge(v_i,w_i)]$ using $\wedge$ and $\vee$, where $\{(v_i,w_i)\}$ is a set of pairs of related vertices in a tree T .

(3) Prove Corollary 1 if negation is also allowed as an operation.

(4) Prove that **verifying** a minimum spanning tree requires $k m \alpha(m,n)$ comparisons in the worst case.

Acknowledgments.

I would like to thank Andrew and Frances Yao for several stimulating discussions on the minimum spanning tree problem which sparked the ideas in Section 7 and the writing of this paper; Adrian Bondy and Ron Graham for criticism which led to correct formulation of the ideas in Section 5; Mark Wegman, for many long and rewarding talks about algorithms for global flow analysis; and Jeff Barth, for providing monetary stimulus for this research.

References

[1] W. Ackermann, "Zum Hilbertshen Aufbau der reelen Zahlen," Math. Ann. 99 (1928), 118-133.

E1 A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "On computing least common ancestors in trees," Proc. Fifth Annual ACM Symposium on Theory of Computing (1973), 253-265.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (1974), 129-134.

[4] ibid, 50-52.

[5] A. V. Aho and J. D. Ullman, "Node listings for reducible flow graphs," Proc. Seventh Annual ACM Symposium on Theory of Computing (1975), 177-185.

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

[7] F. E. Allen, "Control flow analysis," SIGPLAN Notices 5 (1970), 1-19.

[8] B. W. Arden, B. A. Galler, and R. M. Graham, "An algorithm for equivalence declarations," Comm. ACM, 4 (1961), 310-314.

[9] R. C. Backhouse and B. A. Carré, "Regular algebra applied to pathfinding problems," J. Inst. Maths. Applics., 15 (1975), 161-186.

[10] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees," submitted to SIAM J. Comput.

[11] V. Chvátal, D. A. Klarner, and D. E. Knuth, "Selected combinatorial research problems," STAN-CS-72-292, Computer Science Department, Stanford University (1972).

[12] J. Cocke, "Global common subexpression elimination," SIGPLAN Notices, 5 (1970), 20-24.

[13] M. J. Fischer, "Efficiency of equivalence algorithms," Complexity of Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York (1972), 153-168.

[14] A. Fong, J. Kam, and J. Ullman, "Application of lattice algebra to loop optimization," Conf. Record of the Second ACM Symposium on Principles of Prog. Lang. (1975), 1-9.

[15] G. E. Forsythe and C. B. Moler, Computer Solution of Linear Algebraic Systems, Prentice-Hall, Englewood Cliffs, N.J. (1967), 27-33.

[16] B. A. Galler and M. J. Fischer, "An improved equivalence algorithm," Comm. ACM, 7 (1964), 301-303.

[17] S. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," Conf. Record of the Second ACM Symposium on Principles of Prog. Lang. (1975), 22-34.

[18] M. S. Hecht and J. D .Ullman, "Flow graph reducibility," SIAM J. Comput. 1 (1972), 188-202.

[19] M. S. Hecht and J. D. Ullman, "Characterizations of reducible flow graphs," J. ACM., 21 (1974), 367-375.

[20] J. E. Hopcroft, private communication.

[21] J. E. Hopcroft and J. D. Ullman, "Set-merging algorithms," SIAM J. Comput., 2 (1973), 294-303.

[22] T. C. Hu, Integer Programming and Network Flows, Addison-Wesley, Reading, Mass. (1969), 129-150.

[23] K. W. Kennedy, "Node listings applied to data flow analysis," Conf. Record of the Second ACM Symposium on Principles of Prog. Lang. (1973), 10-21.

[24] D. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968), 315-346.

[25] ibid, 353-355.

[26] ibid, 295-304.

[27] D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass. (1973), 170-178.

[28] E. I. Neciporuk, "A Boolean function," Soviet Math. Dokl., 7 (1966), 999-1000.

[29] M. Paterson, unpublished report, Univ. of Warwick, Coventry, Great Britain (1972).

[30] P. W. Purdom and E. F. Moore, "Algorithm 430: Immediate predominator6 in a directed graph," Comm. ACM., 15 (1972), 777-778.

[31] V. Pratt, "The power of negative thinking in multiplying Boolean matrices," Sixth Annual ACM Symposium on Theory of Computing (1974), 80-83.

[32] A. Salomaa, Theory of Automata, Pergammon Press, N. Y. (1969), 120-127.

[33] R. Tarjan, "Finding minimum spanning trees," Mem. No. ERL-M501, Electronics Research Laboratory, University of California, Berkeley, (1975).

[34] R. Tarjan, "Finding dominators in directed graphs," SIAM J. Comput., 5 (1974), 62-89.

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

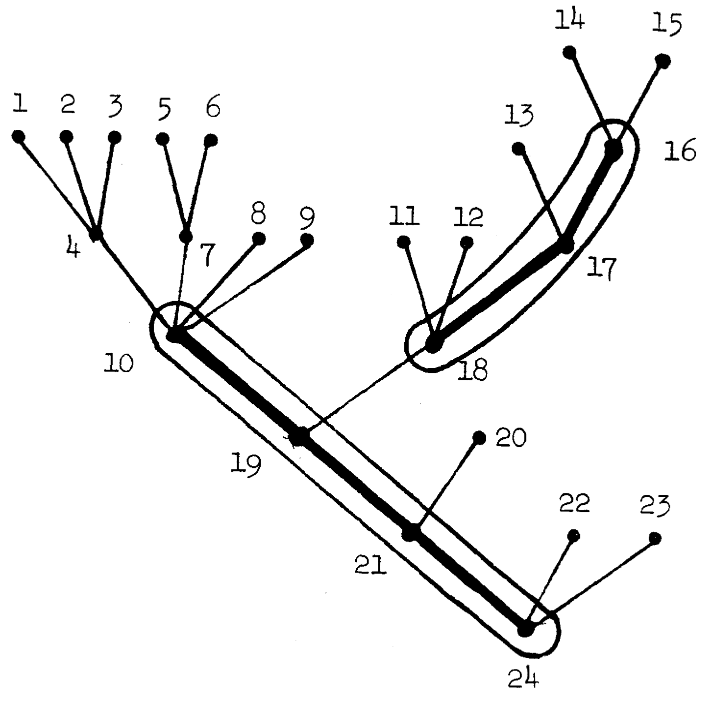
[36] R. Tarjan, "Efficiency of a good but not linear set union algorithm," J. ACM., 22 (1975), 215-225.

[37] R. Tarjan, "Testing flow graph reducibility," J. Comp. Sys. Sciences, 9 (1974), 355-365.

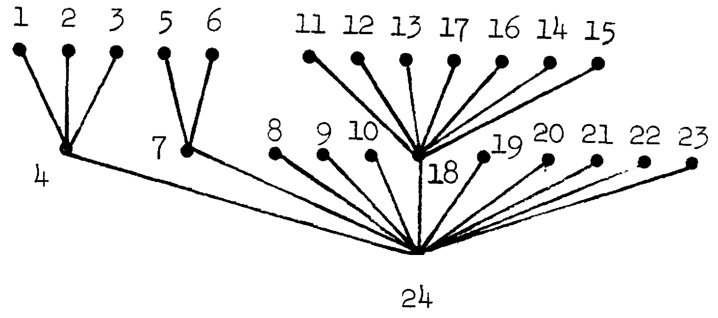
[38] R. Tarjan, "Edge-disjoint spanning trees, dominators, and depth-first search," STAN-CS-74-455, Computer Science Department, Stanford University (1974).

[39] J. D. Ullman, "A fast algorithm for the elimination of common subexpressions," Acta Informatica, 2 (1973), 191-213.

[40] A. C. Yao, "An O(|E| log log |V|) algorithm for finding minimum spanning trees," Info. Proc. Letters, to appear.



(a)



(b)

Figure 1. Representing a tree by a balanced tree and a set of paths.

(a) Tree, with bad edges indicated by heavy lines.

(b) Corresponding balanced tree.

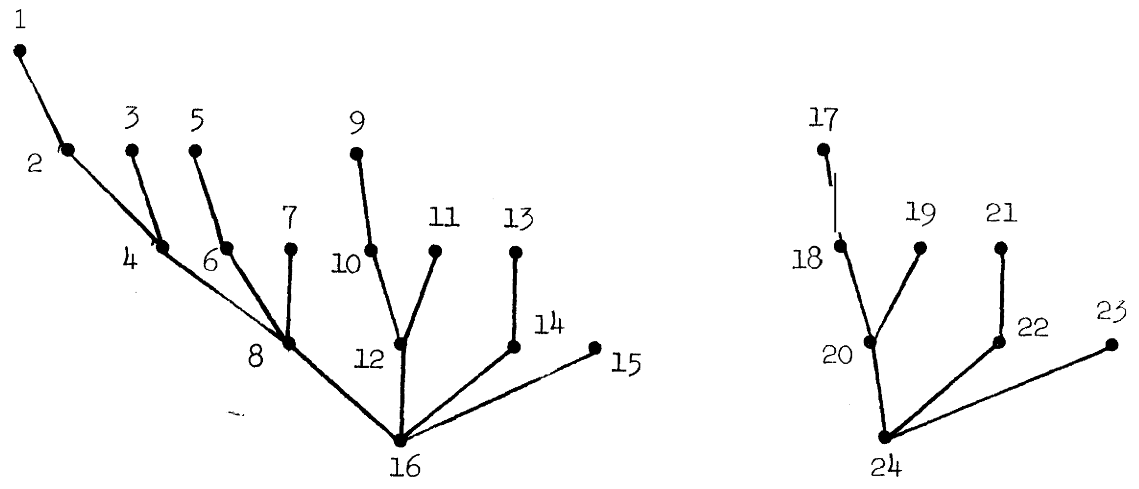


Figure 2. The set of trees TR for k = 24 .

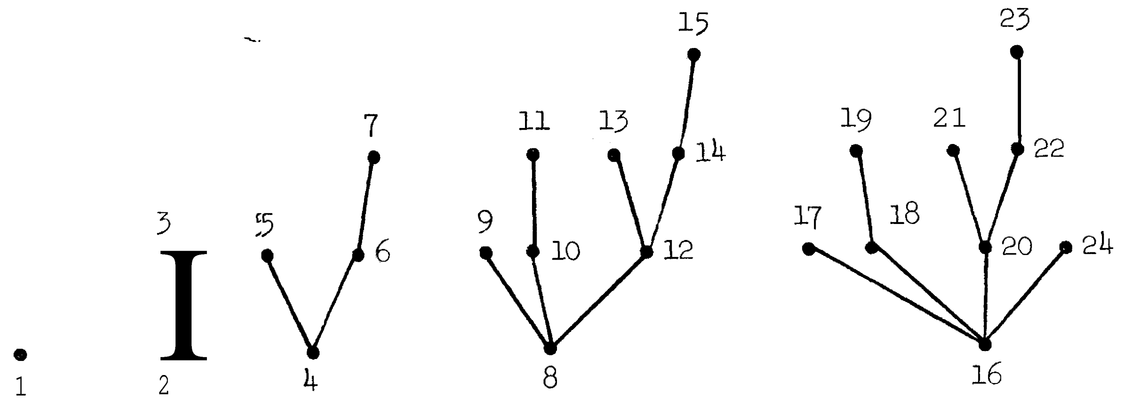


Figure 3. The set of trees TL for k = 24 .

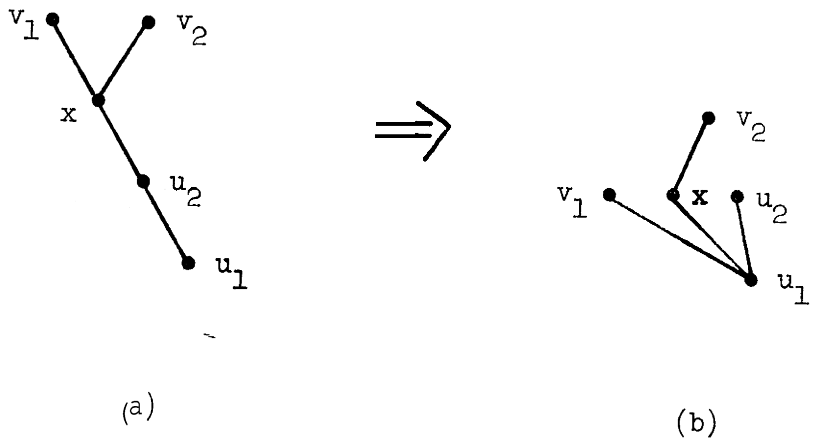


Figure 4: Invalid path compression.

(a) Before compression of path (u_1, v_1) .

(b) After compression of path (u_1, v_1) .

In this tree $\neg(u_2 \overset{*}{\to} v_2)$