Applications of Path Compression on Balanced Trees

ROBERT ENDRE TARJAN

Stanford University, Stanford, California

ABSTRACT. Several fast algorithms are presented for computing functions defined on paths in trees under various assumptions. The algorithms are based on tree manipulation methods first used to efficiently represent equivalence relations. The algorithms have $O((m + n)\alpha(m + n, n))$ running times, where m and n are measures of the problem size and α is a functional inverse of Ackermann's function.

By using one or more of these algorithms in combination with other techniques, it is possible to solve the following graph problems in $O(m\alpha(m, n))$ time, where m is the number of edges and n is the number of vertices in the problem graph.

A. Verifying a minimum spanning tree in an undirected graph. (Best previously known time bound: $O(m \log \log n)$.)

B. Finding dominators in a flow graph. (Best previously known time bound: $O(n \log n + m)$.)

C. Solving a path problem on a reducible flow graph. (Best previously known time bound: $O(m \log n)$.)

Application A is discussed.

KEY WORDS AND PHRASES: balanced tree, dominators, equivalence relation, global flow analysis, graph algorithm, minimum spanning tree, path compression, path problem, tree

CR CATEGORIES: 4.12, 4.34, 5.25, 5.32

1. *Introduction*

There is a small collection of techniques which are useful in building efficient algorithms for a wide variety of graph problems. Here we study one such technique, path compression on balanced trees. The technique was first used for efficiently representing equivalence relations and was subsequently applied to a variety of problems [2, 3, 8, 10, 15, 23, 24].

We extend the range of application of the technique by using it to compute functions defined on paths in trees.[1] Let $(S, \odot)$ be a semigroup with associative operation $\odot$. Consider a sequence of instructions of the following three kinds. These instructions build and manipulate a forest whose vertices are labeled by elements in S.

EVAL(v): Find the root of the tree currently containing v, say r, and return the product of all labels on the path from r to v.

LINK(v, w): Combine the trees with roots v and w into a single tree by adding an edge (v, w) (this makes v the parent of w).

UPDATE(r, x): If r is the root of a tree and r has label l, replace l by $x \odot l$.

We present algorithms for carrying out on-line an arbitrary sequence of m EVAL, LINK, and UPDATE instructions on a forest initially consisting of n one-vertex trees. Our first and simplest algorithm uses path compression to solve the EVAL-LINK-UPDATE

This research was partially supported by a Miller Research Fellowship at the University of California, Berkeley, by the Office of Naval Research, Contract N00014-76-C-0688, and by the National Science Foundation, Grant MCS75-22870.

Author's address: Department of Computer Science, Stanford University, Stanford, CA 94305.

[1] The appendix contains the graph-theoretic definitions used in this paper.

problem in $O((m + n)\cdot\max\{1, \log_2(n^2/(m + n))/\log_2(2(m + n)/n)\})$ time.[2] If the forest built by the LINK instructions is balanced (in a suitably defined sense), the algorithm is even faster, having a running time of $O((m + n)\alpha(m + n, n))$ where $\alpha(m + n, n)$ is a functional inverse of Ackermann's function. If the forest is not balanced, we can in certain situations modify it to make it balanced. We use this idea to obtain $O((m + n)\alpha(m + n, n))$-time algorithms for the following special cases.

1. Each element of S is either a right zero or has a right inverse.
2. S is a totally ordered set and $\odot$ is max.
3. The sequence of EVAL, LINK, and UPDATE instructions is given off-line except for the modifications to the vertex labels specified in the UPDATE instructions.

These algorithms have several important applications. By using the appropriate EVAL-LINK-UPDATE method, we can solve the following graph problems in $O(m\alpha(m, n))$ time, where m is the number of edges and n is the number of vertices in the problem graph.

A. Verifying a minimum spanning tree in an undirected graph. (Best previously known time bound: $O(m \log \log n)$ [6, 31].)

B. Finding dominators in a directed flow graph. (Best previously known time bound: $O(n \log n + m)$ [22].)

C. Solving a path problem in a reducible flow graph. (Best previously known time bound: $O(m \log n)$ [4, 11, 13, 17, 30].)

This paper is a revised and improved version of [25]. It contains eight sections. Section 2 describes the simple algorithm. Sections 3, 5, and 6 describe the more complicated but faster algorithms for handling special cases 1, 2, and 3, respectively. Section 4 surveys previous work on the use of path compression to maintain disjoint sets and to find least common ancestors in trees. Section 7 applies the results in Sections 4, 5, and 6 to the problem of verifying minimum spanning trees. Section 8 contains some further remarks. Applications B and C require additional results that are beyond the scope of this paper; companion papers discuss these applications [19, 26, 27].

2. *A Simple Algorithm Using Path Compression*

In this section we present a simple algorithm for solving the EVAL-LINK-UPDATE problem. The algorithm uses the technique of path compression to achieve an $O((m + n)\cdot\max\{1, \log_2(n^2/(m + n))/\log_2(2(m + n)/n)\})$ running time. The running time is even faster if the tree built by the LINK instructions is balanced.

It is useful to have a little notation to represent products of labels along tree paths. Let v be a descendant of w in the forest built by the LINK instructions. Suppose the path from v to w is $v = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k = w$. We define

$$\odot(v, w) = label(v_0) \odot label(v_1) \odot label(v_2) \odot \cdots \odot label(v_k).$$

If v is any vertex and r is the root of the tree containing v, we define $\odot(v) = \odot(r, v)$. With this notation the value returned by EVAL(v) is $\odot(v)$. Henceforth we use "the value of v" in a technical sense to mean $\odot(v)$.

Consider the following very simple method of carrying out EVAL and LINK instructions. We represent the forest by two arrays, *parent* and *label*. For any vertex v, *parent*(v) is the parent of v in the forest and *label*(v) is the label of v. Initially *parent*(v) = 0 and *label*(v) is the initial label of v.

To carry out LINK(v, w), we execute the assignment *parent*(w) := v. To carry out UPDATE(r, x), we execute the assignment *label*(r) := $x \odot$ *label*(r). To carry out EVAL(v), we follow parent pointers to the root of the tree containing v, multiplying together the labels of the vertices along the path.

This algorithm is not very efficient; if the LINK instructions construct a tree consisting

[2] If f and g are functions of m and n, the notation "$f(m, n)$ is $O(g(m, n))$" means there is a positive constant c such that $f(m, n) \le cg(m, n)$ for all but finitely many values of m and n. The notation "$f(m, n)$ is $\Omega(g(m, n))$" means $g(m, n)$ is $O(f(m, n))$.

of one long path, then an EVAL instruction requires $\Omega(n)$ time in the worst case, and m EVAL instructions require $\Omega(mn)$ time.

We can improve the efficiency of the algorithm by using the associativity of $\odot$. To carry out the EVALs properly, we need not explicitly represent the forest built by the LINKs (henceforth called the *real forest*). Instead, we use a *virtual forest*, which contains the same vertices as the real forest but different edges and labels. The virtual forest satisfies the following properties:

(i) For each tree T of the real forest, there is a corresponding tree VT of the virtual forest which contains the same vertices as T.

(ii) Corresponding trees T and VT have the same root with the same label.

(iii) If v is any vertex, $\odot_F(v) = \odot_{VF}(v)$, where $\odot_F$ denotes evaluation in the real forest and $\odot_{VF}$ denotes evaluation in the virtual forest.

In the virtual forest we can use the following *compression operation*: If $u \rightarrow v \rightarrow w$ in a virtual tree, v has label l_1, and w has label l_2, replace the edge (v, w) by an edge (u, w) and replace the label of w by $l_1 \odot l_2$. It is easy to see that this operation preserves properties (i)–(iii) under the assumption that $\odot$ is associative.

The improved algorithm uses the arrays *parent* and *label* to represent the virtual forest. Initially $parent(v) = 0$ and $label(v)$ is the initial label of v in the real forest. We carry out LINK(v, w) and UPDATE(r, x) as before. We carry out EVAL(v) as follows. Let r be the root of the virtual tree containing v. If $r = v$, we return $label(v)$. Otherwise, we compress the path from r to v using compression operations so that every vertex on the path except r becomes a child of r (see Figure 1). Then we return $label(r) \odot label(v)$.

The following Algol-like recursive procedure COMPRESS(v) carries out the required path compression.

```
procedure COMPRESS($v$);
    comment this procedure assumes $parent(v) \neq 0$;
    if $parent(parent(v)) \neq 0$ then
        COMPRESS($parent(v)$);
        $label(v) := label(parent(v)) \odot label(v)$;
        $parent(v) := parent(parent(v))$ fi;
```

The following procedure implements EVAL using COMPRESS. (In any actual application, the keyword **suitable** would be replaced by the data type of S.)

```
suitable procedure EVAL($v$);
    if $parent(v) = 0$ then EVAL := $label(v)$
        else COMPRESS($v$); EVAL := $label(parent(v)) \odot label(v)$ fi;
```

Knuth [8] attributes the path compression idea to Tritter; independently, McIlroy and Morris used it in an algorithm for finding minimum spanning trees [14].

It is possible to modify COMPRESS in various ways to improve its efficiency and decrease its storage requirements. If running time is at a premium, we can rewrite COMPRESS as an iterative instead of a recursive procedure, using a stack to store vertices on the path from r to v. If storage space is at a premium, we can avoid using an auxiliary stack. Instead, we carry out COMPRESS by following parent pointers from v to r, reversing their directions as we go. Then we follow pointers from r to v while compressing the path.

The path compression method of carrying out EVALs, LINKs, and UPDATEs requires constant time for each instruction plus time proportional to the length of the path compressed for each of the m or fewer executions of COMPRESS. The following theorem bounds the total length of all path compressions. Let F be an arbitrary forest. By a *path compression on* F we mean the following operation: For some pair of vertices v, w such that $v \overset{*}{\rightarrow} w$ in F, modify F by making each vertex except v on the path from v to w a child of v (see Figure 2). Note that v need not be a tree root.

THEOREM 1. *The total length of an arbitrary sequence of m path compressions in an arbitrary n-vertex forest is*

$$O((m + n) \cdot max\{1, log_2(n^2/(m + n))/log_2(2(m + n)/n)\}).$$

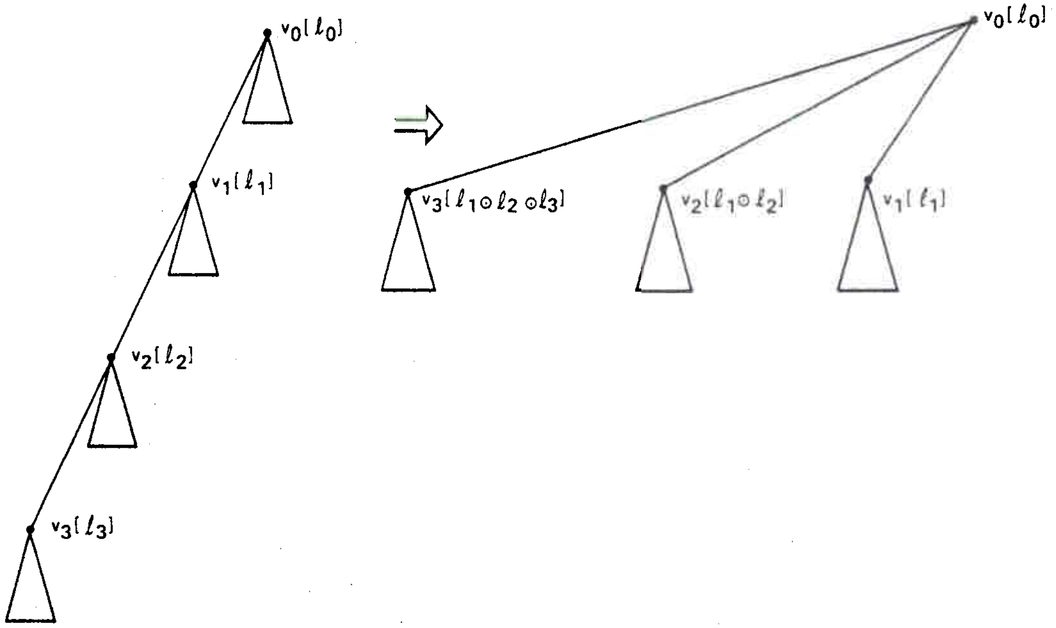


FIG. 1. Path compression to the root of a tree. Triangles denote subtrees; labels are in brackets

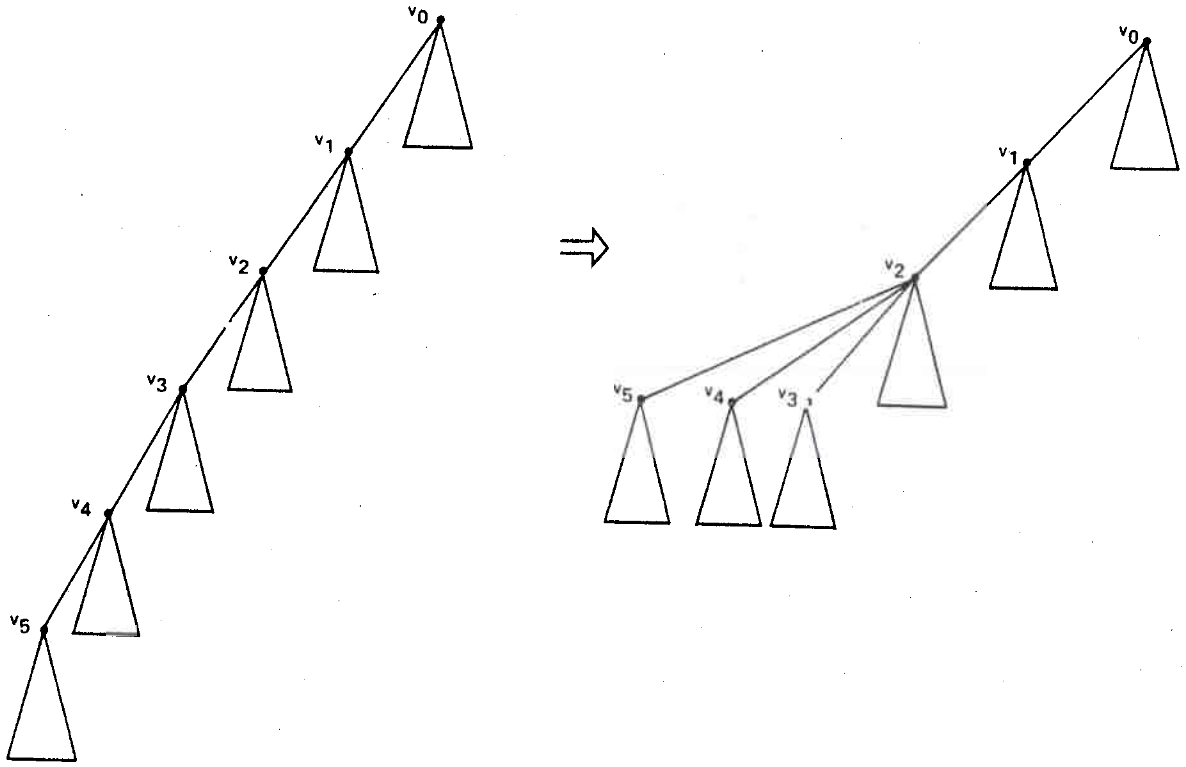


FIG. 2. General path compression. Path compressed is $v_2 \longrightarrow v_3 \longrightarrow v_{\cdot} \longrightarrow v_5$

Paterson [20] proved Theorem 1 for the case $m = O(n)$; a proof for arbitrary m appears in [24]. The bound in Theorem 1 is known to be tight for values of m and n satisfying, for some positive constants c and ϵ, $m \leq cn$ [10] or $m \geq cn^{1+\epsilon}$ [24].

The effect of m executions of EVAL is to carry out m or fewer path compressions on the real forest built by the entire sequence of LINK instructions. Thus Theorem 1 gives a bound on the time required for m EVAL, LINK, and UPDATE instructions.

If the forest built by the LINK instructions is balanced, the running time of the path compression algorithm is faster than Theorem 1 indicates. Let F be any n-vertex forest. F is *balanced* for constants $a > 1$, $c > 0$ if for all i the number of vertices in F of height i does not exceed cn/a^i. Intuitively, this means that most of the vertices of F have small height. Note that any path in a balanced forest has length $O(\log n)$.

Let the function $A(i, j)$ on integers $i, j \geq 0$ be defined by $A(i, 0) = 0$, $A(0, j) = 2^j$ for $j \geq 1$, $A(i, 1) = A(i - 1, 2)$ for $i \geq 1$, and $A(i, j) = A(i - 1, A(i, j - 1))$ for $i \geq 1, j \geq 2$. $A(i, j)$ is a variant of Ackermann's function [1] slightly different from the version used in [24]. Let $\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor 2m/n \rfloor) > \log_2 n\}$, where $\lfloor x \rfloor$ denotes the greatest integer not larger than x. For $n < 2^{16} = 65{,}536$, $\alpha(m + n, n) = 1$; for all larger feasible n, $\alpha(m + n, n) = 2$.

THEOREM 2 [24]. *The total length of an arbitrary sequence of* m *path compressions in an* n*-vertex forest balanced for* a, c *is* $O((m + n)\alpha(m + n, n))$, *where the constant depends on* a *and* c.

Our goal is to devise an algorithm for the EVAL-LINK-UPDATE problem which requires $O((m + n)\alpha(m + n, n))$ time for any forest. We shall succeed in doing this in several important special cases, although not in the general case. Our approach will be to modify the implementation of the LINK instruction so that the virtual forest it builds is balanced.

3. *An Algorithm for Semigroups with Inverses and Zeros*

In this section we present an $O((m + n)\alpha(m, n))$-time algorithm for the special case in which each element of S is either a right zero or has a right inverse. More precisely, we assume that for each element $x \in S$, either

(a) $y \odot x = x$ for all $y \in S$ (x is a *right zero*), or

(b) there is an element x^{-1} such that $y \odot x \odot x^{-1} = y$ for all $y \in S$ (x has a *right inverse*).

We shall ignore right zeros for the moment and assume that every element of S has a right inverse. To carry out EVAL, LINK, and UPDATE instructions, we maintain a virtual forest which satisfies properties (i) and (iii) of Section 2 but not property (ii). That is, to each tree T of the real forest corresponds a tree VT of the virtual forest, having the same vertices as T but not the same root. Furthermore vertex values in the real forest and in the virtual forest agree. When combining two trees in the virtual forest, we always make the root of the smaller tree a child of the root of the larger tree. This guarantees that the virtual forest is balanced, and Theorem 2 gives an $O((m + n)\alpha(m + n, n))$ time bound.

The algorithm uses three arrays, *parent*, *label*, and *size*. Arrays *parent* and *label* represent the virtual forest as in Section 2. If v is the root of a virtual tree, *size*(v) is the number of vertices in the tree. Initially *parent*$(v) = 0$, *size*$(v) = 1$, and *label*(v) is the initial label of v in the real forest.

We carry out EVAL(v) exactly as in the path compression algorithm; i.e., if *parent*$(v) = 0$, we return *label*(v); otherwise we execute COMPRESS(v) and return *label*(*parent*(v)) $\odot$ *label*(v). We also carry out UPDATE(r, x) by using path compression. The desired effect of UPDATE(r, x) is to multiply the value of every vertex in the real tree containing r by x. We carry out this instruction by finding the root of the virtual tree containing r and multiplying its label in the virtual tree by x. The following procedure carries out UPDATE(r, x) by using the fact that if r is not the root of a virtual tree, COMPRESS(r) makes r a child of the root.

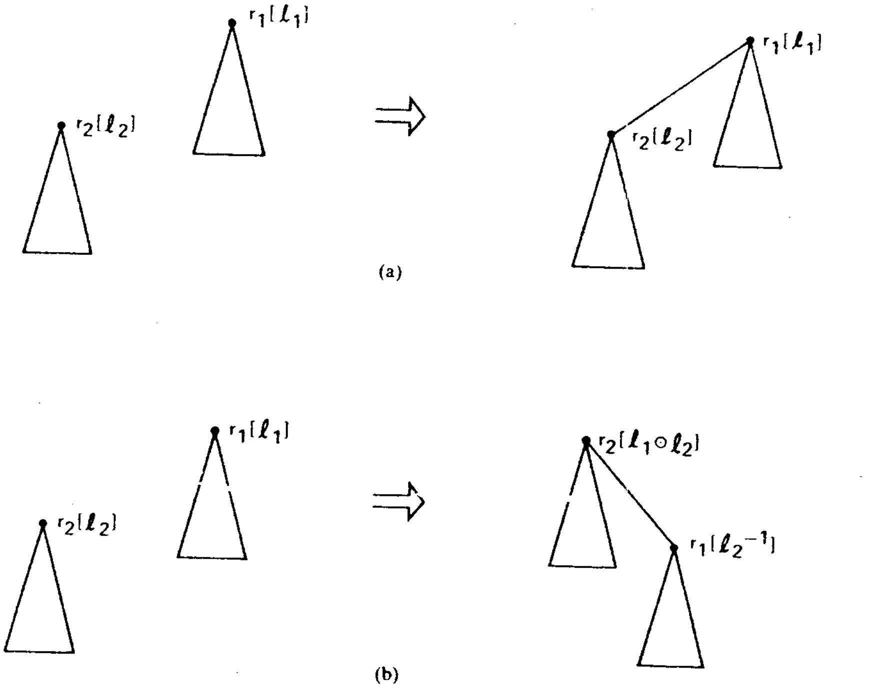


FIG. 3. Last part of LINK(v, w) with balancing. Labels are in brackets. (a) $size(r_1) \geq size(r_2)$; (b) $size(r_1) < size(r_2)$

```
procedure UPDATE($r$, $x$);
    if $parent(r) = 0$ then $label(r) := x \odot label(r)$
        else COMPRESS($r$); $label(parent(r)) := x \odot label(parent(r))$ fi;
```

We carry out LINK(v, w) by using *balancing*. The desired effect of LINK(v, w) is to combine the real trees with roots v and w, and to multiply the value of each vertex in the real tree with root w by the value of v. To carry out LINK(v, w), it suffices to update values in the virtual tree VT_2 containing w and to combine VT_2 with the virtual tree VT_1 containing v. In order to do this we need access to the roots r_1, r_2 of VT_1, VT_2, respectively. If $r_1 \neq v$, we execute COMPRESS(v), making v a child of r_1. If $r_2 \neq w$, we execute COMPRESS(w), making w a child of r_2. The value of v is then $label(r_1)$ if $v = r_1$, $label(r_1) \odot label(v)$ otherwise. If $v \neq r_1$, we replace $label(r_2)$ by $label(v) \odot label(r_2)$. It remains for us to multiply vertex values in VT_2 by $label(r_1)$ and combine the trees.

What we do next depends on the relative sizes of VT_1 and VT_2. Figure 3 illustrates the two cases. If VT_1 contains at least as many vertices as VT_2, we make r_2 a child of r_1. This has the effect of multiplying the value of each vertex in VT_2 by $label(r_1)$ and thus completes LINK(v, w) while preserving (i) and (iii). If VT_1 has fewer vertices than VT_2, we make r_1 a child of r_2. In addition, we simultaneously replace $label(r_1)$ by $label(r_2)^{-1}$ and $label(r_2)$ by $label(r_1) \odot label(r_2)$. This has the effect of leaving the values of vertices in VT_1 alone while multiplying the values of vertices in VT_2 by the old $label(r_1)$, thus completing LINK(v, w) while preserving (i) and (iii).

The following procedure implements LINK. Note that *zerotest* always takes the **true** branch in the case we are considering, namely, when every element of S has a right inverse.

```
procedure LINK($v$, $w$);
    begin
        if $parent(w) = 0$ then $r_2 := w$
            else COMPRESS($w$); $r_2 := parent(w)$ fi;
zerotest: if $label(r_2)$ has a right inverse then
```

```
            if $parent(v) = 0$ then $r_1 := v$,
                else COMPRESS($v$); $r_1 := parent(v)$;
                    $label(r_2) := label(v) \odot label(r_2)$ fi;
            if $size(r_1) \geq size(r_2)$ then
                $parent(r_2) := r_1$; $size(r_1) := size(r_1) + size(r_2)$
            else $parent(r_1) := r_2$; $size(r_2) := size(r_1) + size(r_2)$;
                $label(r_1), label(r_2) := label(r_2)^{-1}, label(r_1) \odot label(r_2)$ fi fi
    end LINK;
```

We call this implementation of EVAL, LINK, and UPDATE *path compression with balancing*. Morris apparently originated the balancing idea [14]. It also appears in [12].

The time required to execute m instructions using path compression with balancing is $O(m)$ plus the time to execute no more than $2m$ path compressions. Let U be the uncompressed virtual forest; i.e., the virtual forest built by the LINKs, ignoring all compression. To obtain U, we construct a new edge (x, y) each time LINK assigns $x := parent(y)$. It is not hard to show that U is balanced for constants $a = 2$, $c = 1$. (This is proved in [24]; see also Section 5.) Theorem 2 implies that path compression with balancing requires $O((m + n)\alpha(m + n, n))$ time.

If necessary we can reduce the storage requirements of this algorithm. Since $size(v)$ is only useful if $parent(v) = 0$, we can store both sizes and parents in one array by using negative numbers to represent sizes and positive numbers to represent parents.

We now extend the algorithm to handle right zeros. Such elements cause problems only in the LINK instruction. Note that if a right zero appears in a product representing the value of a vertex, all terms to the left of the zero can be ignored. In other words, if the value of w is a right zero, the instruction LINK(v, w) can be ignored because it, and subsequent LINKs and UPDATEs, cannot change the value of any vertex in the tree containing w. This means that the previous implementation of LINK is still valid; if the value of w is a right zero, *zerotest* will cause LINK to do nothing. Proving the validity of the implementation requires the following lemma.

LEMMA 1. *If $x \in S$ and $y \in S$ have right inverses, then so do $x \odot y$ and x^{-1}.*

PROOF. $z \odot x \odot y \odot y^{-1} \odot x^{-1} = z \odot x \odot x^{-1} = z$, which means that $y^{-1} \odot x^{-1}$ is a right inverse for $x \odot y$. Suppose x^{-1} has no right inverse. Then x^{-1} is a right zero, which means $x \odot x^{-1} = x^{-1}$. But then $x = x \odot x \odot x^{-1} = x \odot x^{-1} = x^{-1}$, which contradicts the fact that x has a right inverse. □

Lemma 1 and the inner workings of LINK, EVAL, and UPDATE guarantee that $label(v)$ has no right inverse only for vertices v which are virtual tree roots. Referring to LINK, it follows that $label(r_2)$ has a right inverse if and only if the value of w does, and our implementation is correct.

The algorithm of this section applies to any group, such as the real numbers under addition, and to the real numbers under multiplication. An algorithm for the real numbers under addition is useful to keep track of vertex depths in a forest, and the algorithm we have presented here is based on Aho, Hopcroft, and Ullman's method of solving the forest depth problem [2, 3]. An algorithm for the real numbers under multiplication is useful for solving systems of linear equations defined on reducible flow graphs [27].

4. *Disjoint Sets and Least Common Ancestors*

Path compression with balancing was originally developed to solve the following problem. Suppose we are given n disjoint sets, each containing one element, and each having a distinct name. We wish to carry out two types of instructions on these sets.

FIND(e): return the name of the set containing element e;
UNION(A, B): add the elements in set B to set A, destroying set A.

We can apply the algorithm of Section 3 to solve this problem. (In this application the procedures for EVAL and LINK can be simplified somewhat.) We use a forest, each vertex of which represents an element. The initial label of a vertex is the name of the set initially containing the corresponding element. S is the collection of set names, with operation $x \odot y = x$. Note that each set name is its own right inverse. For each element e, let $v(e)$ be the vertex representing e. For each set name A, let $v(A)$ be the vertex initially labeled by A. We implement UNION and FIND as follows:

```
suitable procedure FIND($e$);
    FIND := EVAL($v(e)$);

procedure UNION($A$, $B$);
    LINK($v(A)$, $v(B)$);
```

The time required for m instructions is $O((m + n)\alpha(m + n, n))$. This set union algorithm is useful for handling EQUIVALENCE and COMMON statements in Fortran [5, 12], finding minimum spanning trees [6, 31], and testing flow graphs for reducibility [23].

Aho, Hopcroft, and Ullman have used the set union algorithm to compute least common ancestors in a tree off-line [3]. We shall need an algorithm for this problem in Section 7, so we present a version of their method here. They proceed by way of an intermediate problem, called the *off-line min problem*, but it is much cleaner to use the set union algorithm directly. Let T be a tree with root r and let $pairs = \{\{v_i, w_i\} \mid 1 \le i \le m\}$ be a set of m vertex pairs. We wish to compute LCA(v_i, w_i) for each pair. The following algorithm carries out the computation.

```
procedure LCA;
    begin
        for each $\{v, w\} \in pairs$ do unmark $\{v, w\}$ od;
        for each $v \in V$ do create a set $\{v\}$ named $v$ od;
        SEARCH($r$)
    end LCA;
```

Recursive procedure SEARCH is defined by

```
procedure SEARCH($v$);
    begin
        for each $w \in children(v)$ do SEARCH($w$); UNION($v$, $w$) od;
        for each $\{v, w\} \in pairs$ do if $\{v, w\}$ not marked then mark $\{v, w\}$
                                     else $lca(v, w)$ := FIND($w$) fi od
    end SEARCH;
```

This algorithm assumes that V is the set of tree vertices and that $children(v)$ is the set of children of v for each vertex v. SEARCH carries out a depth-first search of the tree. During the search, each pair $\{v, w\}$ is examined twice, once when the search is at v and once when the search is at w. The second time $\{v, w\}$ is examined, its least common ancestor is computed and stored in $lca(v, w)$. It is not hard to prove the correctness of this method by using properties of depth-first search. See [2, 3, 21, 22]. The algorithm requires $O((m + n)\alpha(m + n, n))$ time and $O(m + n)$ space if the set *pairs* is represented so that for each vertex v the pairs $\{v, w\}$ can be retrieved in constant time per pair. An adjacency structure [3, 21] is suitable for this purpose.

5. *An Algorithm for Totally Ordered Sets with Operation max*

There are important situations in which the algorithm of Section 3 does not apply. For instance, if S is totally ordered under an ordering $\preccurlyeq$ and $\odot$ is max, only the minimum element of S (if any) has an inverse. We shall devise a different and somewhat more complicated algorithm for this case.

Our algorithm uses a virtual forest satisfying properties (i)–(iii) of Section 2 and the following additional property:

(iv) Each virtual tree VT consists of a set of *subtrees* $\mathrm{ST}_0, \mathrm{ST}_1, \ldots, \mathrm{ST}_k$ with roots $r_0, r_1, \ldots, r_k$, respectively, joined by a path $r_0 \to r_1 \to \cdots \to r_k$. The subtree roots satisfy $label(r_j) \preccurlyeq label(r_{j+1})$ for $0 \le j < k$. (See Figure 4.)

Suppose v is a vertex in some subtree ST_j. The value of v does *not* depend on $label(r_i)$ for $i < j$, since $label(r_i) \preccurlyeq label(r_j)$ for any $i < j$. This means that to compute the value of v, we

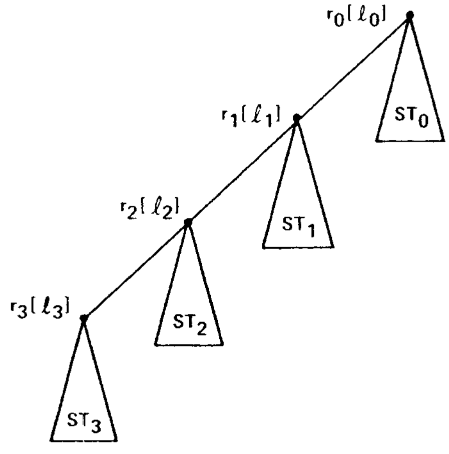


FIG. 4. Structure of virtual trees for operation max. Labels satisfy $l_0 \leq l_1 \leq l_2 \leq l_3$

need examine labels only of vertices in ST_j. We can thus use path compression *within the subtrees* to compute values. By properly manipulating the subtrees, we shall be able to guarantee that they are balanced. In this way we achieve an $O((m + n)\alpha(m + n, n))$ time bound.

To represent the subtrees, we use two arrays, *parent* and *label*. For each vertex v, *label*(v) is the label of v in the virtual forest; *parent*(v) is the parent of v in the virtual forest unless v is the root of a subtree, in which case *parent*(v) = 0. We implement EVAL(v) exactly as in Sections 2 and 3.

To represent the way the subtrees fit together into virtual trees, we use two additional arrays, *child* and *size*. These arrays are defined only for subtree roots. Let VT be a virtual tree with subtrees $\mathrm{ST}_0, \mathrm{ST}_1, \ldots, \mathrm{ST}_k$ having roots $r_0, r_1, \ldots, r_k$, respectively. For any root r_j, *child*(r_j) denotes the child of r_j (if any) that is a subtree root; i.e., *child*(r_j) = r_{j+1} if $j < k$; *child*(r_j) = 0 if $j = k$. For any root r_j, *size*(r_j) is the number of descendants of r_j in VT; i.e., *size*(r_j) = $\sum_{i=j}^{k} |\mathrm{ST}_i|$. For purposes of analysis we shall denote $|\mathrm{ST}_i|$, the number of vertices in ST_i, by *subsize*(r_i). For convenience we assume that *size*(0) = 0; then *subsize*(r_j) = *size*(r_j) − *size*(*child*(r_j)) for $1 \leq j \leq k$.

To carry out UPDATE(r, x), we first replace *label*(r) by max{x, *label*(r)}. This preserves properties (i)–(iii) but may invalidate (iv). Let VT with subtrees $\mathrm{ST}_0, \mathrm{ST}_1, \ldots, \mathrm{ST}_k$ having roots $r = r_0, r_1, \ldots, r_k$ be the virtual tree with root r. We determine the maximum value of j such that *label*(r) > *label*(r_j). We then combine subtrees $\mathrm{ST}_1, \mathrm{ST}_2, \ldots, \mathrm{ST}_j$ into a single subtree, replacing the label of the root of this subtree by *label*(r).

To combine the subtrees, we first replace ST_1 by a combination of ST_1 and ST_2, then replace the new ST_1 by a combination of ST_1 and ST_3, and so on, until all the subtrees ST_i, for $2 \leq i \leq j$, are combined with ST_1. When combining ST_i with ST_1 to form the new ST_1, we make the root of the larger subtree the parent of the root of the smaller. (See Figure 5.) Note that this process leaves ST_0 intact, thus preserving (ii).

The following procedure implements this idea. For convenience, the procedure assumes that *label*(0) = ∞, where ∞ is the maximum element of S. (If S has no maximum element, we add one.)

```
procedure UPDATE($r$, $x$);
    begin
              comment this procedure assumes $size(0) = 0$ and $label(0) = \infty$;
              $label(r) := \max\{x, label(r)\}$;
labeltest1:   if $label(r) > label(child(r))$ do
                  $r_1 := child(r)$;
labeltest2:       while $label(r) > label(child(r_1))$ do
sizetest:             if $size(r_1) + size(child(child(r_1))) \geq 2^* size(child(r_1))$ then
                          $parent(child(r_1)) := r_1$; $child(r_1) := child(child(r_1))$
```

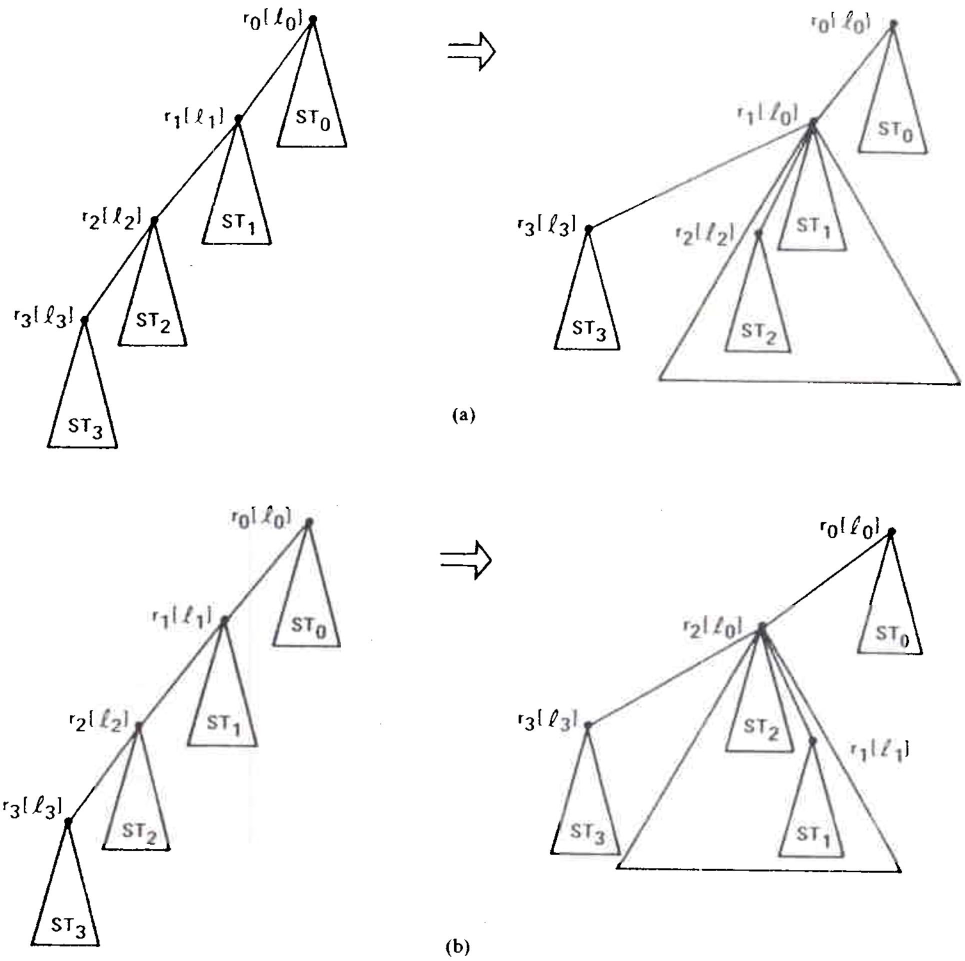
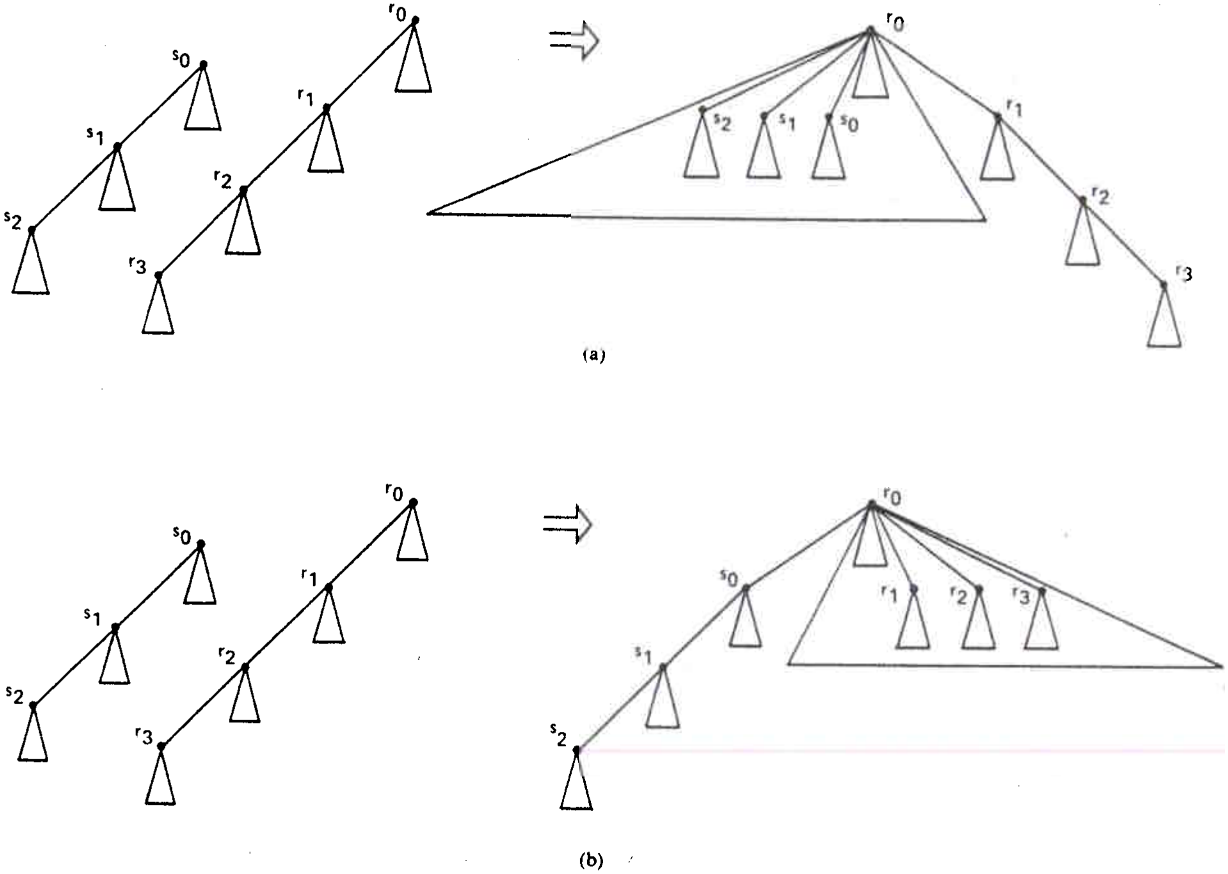


FIG. 5. Combination of subtrees ST_1, ST_2 by UPDATE if $l_0 > l_2$. (a) *subsize*$(r_1) \geq$ *subsize*(r_2); (b) *subsize*$(r_1) <$ *subsize*(r_2). This operation is repeated until there are no more subtrees or the next subtree ST_{j+1} has $l_0 \leq l_{j+1}$

```
                    else $size(child(r_1)) := size(r_1)$;
                        $r_1 := parent(r_1) := child(r_1)$ fi od;
                $label(r_1) := label(r); child(r) := r_1$ fi
    end UPDATE;
```

Note that the choice of *label*(0) = ∞ guarantees that *labeltest*1 fails when *child*$(r) = 0$ and that *labeltest*2 fails when *child*$(r_1) = 0$. Note also that *sizetest* succeeds exactly when $size(r_1) + size(child(child(r_1))) \geq 2^* \ size(child(r_1))$, i.e., when $subsize(r_1) = size(r_1) - size(child(r_1)) \geq size(child(r_1)) - size(child(child(r_1))) = subsize(child(r_1))$. It is easy to show that this procedure carries out UPDATE(r, x) correctly while maintaining (i)–(iv).

To carry out LINK(v, w), we first perform UPDATE$(w, label(v))$ This correctly updates values in the virtual tree VT_2 containing w. Let $w = s_0, s_1, \ldots, s_l$ be the subtree roots in VT_2 after UPDATE$(w, label(v))$ is performed, and let VT_1 with subtree roots $v = r_0, r_1, \ldots, r_k$ be the virtual tree containing v. After the UPDATE, $label(v) \preccurlyeq label(w)$. If VT_1 is larger than VT_2, we combine all the subtrees in VT_2 with the subtree of VT_1 rooted at v by making v the parent of $s_0, s_1, \ldots, s_l$. This combines VT_1 and VT_2 into a single virtual tree. If VT_1 is smaller than VT_2, we combine all subtrees of VT_1 into a single subtree by making v the parent of $r_1, r_2, \ldots, r_k$. Then we combine VT_1 and VT_2 into a single virtual tree by making w the child of v. Figure 6 illustrates the two cases. The following procedure implements LINK. The operator "↔" denotes exchange of values.



(a)

(b)

FIG. 6. Combination of subtrees by last part of LINK(v, w). Labels are not changed. (a) $size(r_0) \geq size(s_0)$; (b) $size(r_0) < size(s_0)$

```
procedure LINK(v, w):
    begin
        UPDATE(w, label(v));
        size(v) := size(v) + size(w);
        s := w;
        if size(v) < 2* size(w) then s ↔ child(v) fi;
        while s ≠ 0 do parent(s) := v; s := child(s) od
    end LINK;
```

Verifying that this implementation carries out LINK(v, w) while preserving (i)–(iv) is straightforward. As in Section 3, it is possible to save an array by combining either *child* or *size* with *parent*, since *child* and *size* are needed only for subtree roots and *parent* is zero for subtree roots.

It remains for us to show that the subtrees built by UPDATE and LINK are balanced. Let U be the uncompressed set of subtrees built by LINK and UPDATE. To obtain U, we construct a new edge (x, y) each time either LINK or UPDATE assigns $x := parent(y)$. Each such new edge joins a pair of subtree roots.

For any vertex v, let $subsize(v)$ be the number of descendants of v in U. We call an edge (x, y) of U *good* if $subsize(x) \geq 2 \cdot subsize(y)$. If (x, y) and (y, z) are edges of U, we call (y, z) *mediocre* if $subsize(x) \geq 2 \cdot subsize(z)$. Once an edge becomes good (or mediocre) it stays good (or mediocre), since when (x, y) becomes an edge of U, $subsize(y)$ (and $subsize(z)$ if $y \rightarrow z$) is fixed, and $subsize(x)$ can only increase.

Each edge added to U by UPDATE is good. It is not hard to show that a forest consisting only of good edges is balanced [24]. Unfortunately, LINK can add edges that are not good to U. We shall show, however, that such edges eventually become mediocre.

LEMMA 2. *If $x \rightarrow y \rightarrow z$ in U, then (y, z) is mediocre.*

PROOF. If (y, z) is added to U by UPDATE (including the call on UPDATE in LINK), then $subsize(y) \geq 2 \cdot subsize(z)$. Adding additional edges to U preserves this relationship, and when (x, y) is added to U, $subsize(x) \geq subsize(y) \geq 2 \cdot subsize(z)$. Thus (y, z) is mediocre. A similar proof works if (x, y) is added to U by UPDATE.

Suppose on the other hand that both (x, y) and (y, z) are added to U by LINK outside the call on UPDATE. An inspection of LINK shows that when (y, z) is added to U, $size(y) \geq 2 \cdot subsize(z)$. This relationship is preserved until (x, y) is added to U, at which time all descendants of y in the virtual tree containing y become descendants of x in the subtree containing x; i.e., $subsize(x) \geq 2 \cdot subsize(z)$. Thus (y, z) is mediocre. □

THEOREM 3. *A forest F is balanced for constants $a = \sqrt{b}$, $c = \sqrt{b}$ if it satisfies the following property: If $x \rightarrow y \rightarrow z$ in F, then $d(x) \geq b \cdot d(z)$, where $d(v)$ is the number of descendants of vertex v in F.*

PROOF. We prove by induction that any vertex of height h in F has at least $b^{\lfloor h/2 \rfloor}$ descendants. The result is obvious for $h = 0, 1$. Suppose the result holds for $h - 2$ and let x be a vertex of height h. Then there are vertices y, z in F of height $h - 1$, $h - 2$, respectively, such that $x \rightarrow y \rightarrow z$. By the induction hypothesis, z has at least $b^{\lfloor (h-2)/2 \rfloor}$ descendants, and by the assumption of the theorem, $d(x) \geq b \cdot b^{\lfloor (h-2)/2 \rfloor} := b^{\lfloor h/2 \rfloor}$. This proves the result for h.

Any two vertices of the same height have disjoint sets of descendants. Thus the number of vertices of height h in F is at most $n/b^{\lfloor h/2 \rfloor} \leq n/b^{(h-1)/2} = \sqrt{b}n/(\sqrt{b})^h$. □

COROLLARY 1. *The forest U is balanced for constants $a = \sqrt{2}$, $c = \sqrt{2}$.*

It follows from Corollary 1 and Theorem 2 that the algorithm of this section performs its path compressions on a balanced forest. Thus the total time for m EVAL instructions is $O((m + n)\alpha(m + n, n))$. The time for the LINK and UPDATE instructions is a constant per instruction plus time proportional to the number of edges in U. Thus the LINK and UPDATE instructions require $O(m + n)$ time, and the total time for m instructions is $O((m + n)\alpha(m + n, n))$. The space required is $O(n)$.

6. *An Off-Line Algorithm*

The algorithms of Sections 3 and 5 still do not cover all interesting cases of the EVAL-

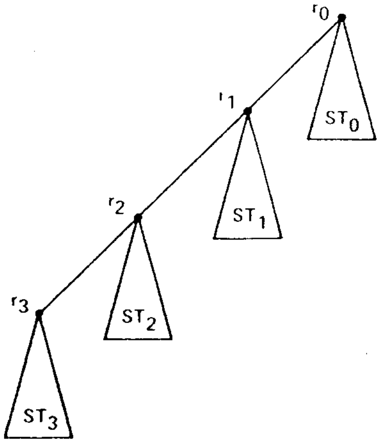


FIG. 7. Structure of virtual trees for off-line algorithm. Subtrees satisfy $2 \cdot subsize(r_j) \le subsize(r_{j+1})$

LINK-UPDATE problem. For instance, in global flow analysis it is necessary to implement EVAL, LINK, and UPDATE for a semigroup consisting of a set of bit vectors under an operation such as bitwise "and" or bitwise "or" [11, 13, 17, 30]. In this section we present an algorithm that solves the EVAL-LINK-UPDATE problem for *any* semigroup $(S, \odot)$, under the assumption that the sequence of instructions is given off-line *except* for the modifications to the vertex labels specified in the UPDATE instructions. (Henceforth we shall call this the *half-line* case.)

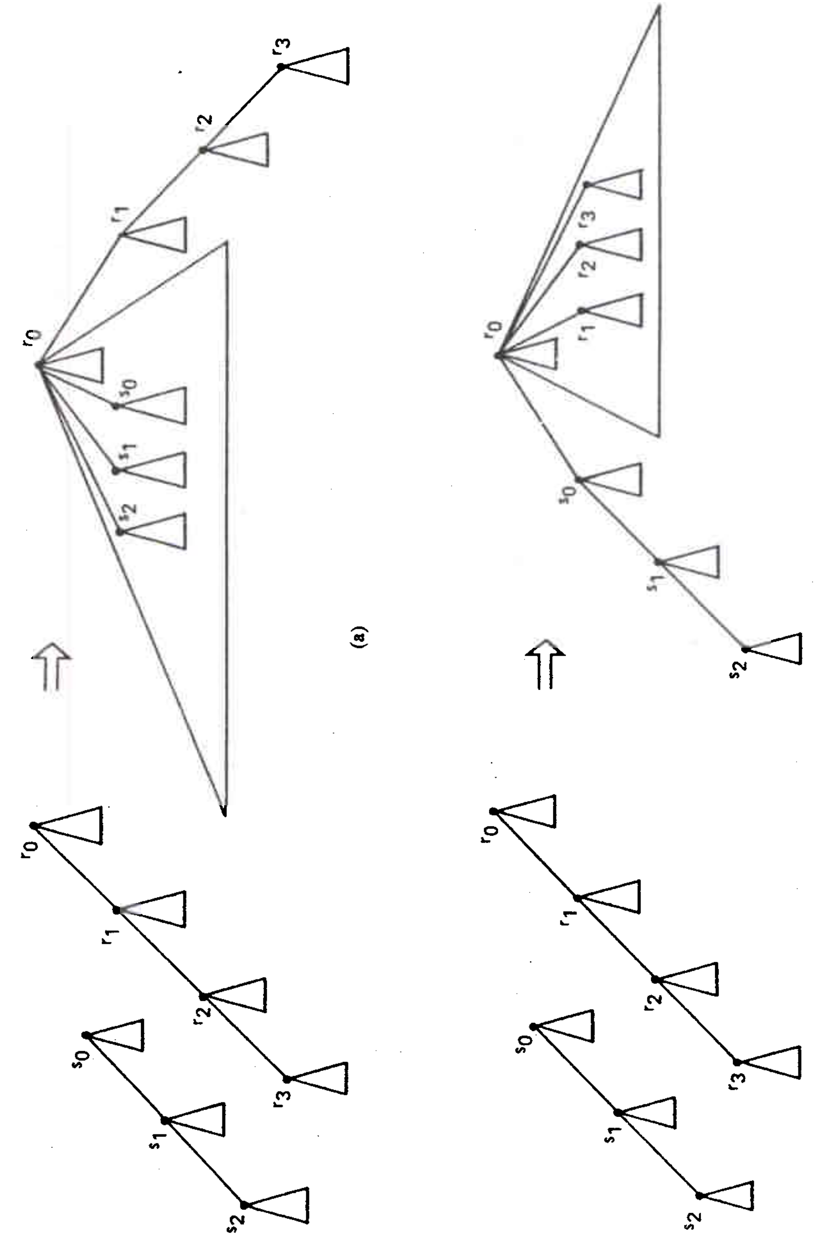
We shall deal first with the somewhat simpler situation in which the sequence of instructions is given completely off-line; i.e., the values for the EVAL(v) instructions need not be returned until the entire instruction sequence is scanned. The algorithm consists of two passes. The first pass resembles the method of Section 5. We scan the instructions, maintaining a virtual forest to represent vertex values. The virtual forest satisfies properties (i)–(iii) of Section 2, as well as the following property, which replaces property (iv) of Section 5.

(v) Each virtual tree consists of a set of subtrees ST_0, ST_1, ST_2, ..., ST_k with roots r_0, r_1, ..., r_k, respectively, joined by a path $r_0 \to r_1 \to \cdots \to r_k$. The subtrees satisfy $2|ST_j| \le |ST_{j+1}|$ for $0 \le j < k$. (See Figure 7.)

To represent the subtrees, we use arrays *parent* and *label* as in Section 5. For each vertex v, *label*(v) is the label of v in the virtual forest; *parent*(v) is the parent of v in the virtual forest unless v is a subtree root, in which case *parent*(v) = 0. To represent the way the subtrees fit together into virtual trees, we use arrays *child* and *size* as in Section 5. Let VT be a virtual tree with subtrees ST_0, ST_1, ..., ST_k having roots r_0, r_1, ..., r_k, respectively. For any root r_j, *child*(r_j) is the child of r_j (if any) which is a subtree root; i.e., $child(r_j) = r_{j+1}$ if $j < k$, $child(r_j) = 0$ if $j = k$. For any root r_j, *size*(r_j) is the number of descendants of r_j in VT; i.e., $size(r_j) = \sum_{i=j}^{k} |ST_i|$. As in Section 5, we shall denote $|ST_i|$ by $subsize(r_i)$ and let $subsize(0) = 0$; then $subsize(r_j) = size(r_j) - size(child(r_j))$ for $0 \le j \le k$.

To carry out UPDATE(r, x), we execute the assignment $label(r) := x \odot label(r)$. To carry out LINK($v$, w), we use a method similar to that used in Section 5. Let VT_1 with subtree roots $v = r_0, r_1, \ldots, r_k$ and VT_2 with subtree roots $w = s_0, s_1, \ldots, s_l$ be the virtual trees containing v and w, respectively.

If $|VT_1| \ge |VT_2|$, we combine each subtree of VT_2 with the subtree of VT_1 rooted at v, by making v the parent of $s_0, s_1, \ldots, s_l$. In doing so, we modify the labels of $s_1, s_2, \ldots, s_l$ to preserve property (iii). Since this process increases the size of the subtree rooted at v, property (v) may no longer hold. To restore (v), we combine the subtrees rooted at $r_1, r_2, \ldots, r_j$ with the subtree rooted at v, by making v the parent of $r_1, r_2, \ldots, r_j$, until we find a subtree root r_{j+1} whose subtree size is at least twice that of the enlarged subtree rooted at v. In the process we modify the labels of $r_1, r_2, \ldots, r_j$ to preserve (iii). (See Figures 8 and 9.)



(a)

(b)

FIG. 8. First part of LINK(*v*, *w*). Labels are updated as in path compression. (a) *size*(r_0) ≥ *size*(s_0); (b) *size*(r_0) < *size*(s_0)

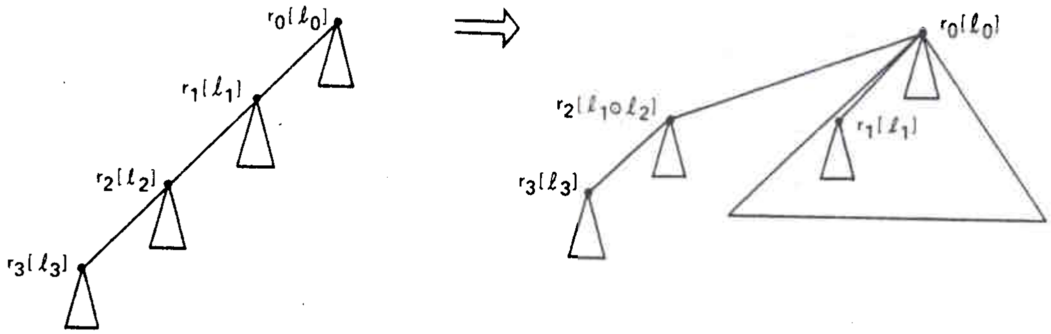


FIG. 9. Second part of LINK(v, w). This operation is repeated until all subtrees are combined or the next subtree ST_{j+1} has at least twice the size of the expanded ST_1

The process is similar if $|\mathrm{VT}_1| < |\mathrm{VT}_2|$. First we replace r_1 as *child*(v) by s_1. Then we combine the subtrees rooted at $r_1, r_2, \ldots, r_k$ with the subtree rooted at v, by making v the parent of $r_1, r_2, \ldots, r_k$. In doing so, we modify the labels of $r_1, r_2, \ldots, r_k$ to preserve (iii). Finally we combine the subtrees of VT_2 rooted at $s_0, s_1, \ldots, s_j$ with the subtree rooted at v, by making v the parent of $s_0, s_1, \ldots, s_j$, until we find a subtree root s_{j+1} whose subtree size is at least twice that of the enlarged subtree rooted at v. In doing so, we modify the labels of $s_0, s_1, \ldots, s_j$ to preserve (iii). (See Figures 8 and 9.)

Our implementation of LINK updates three arrays, *root*, *vertex*, and *extra label*, in addition to *parent*, *label*, *child*, and *size*. These arrays are used in the evaluation process. The array *root* allows easy access to the root of a virtual tree from the roots of its subtrees. Let $r_0, r_1, \ldots, r_k$ be the subtree roots of a virtual tree. If $0 \le j < k$, $root(r_j) = r_k$; $root(r_k) = r_0$. Thus from any subtree root we can reach the root of its virtual tree in one or two steps.

LINK numbers vertices consecutively from 1 as they cease to be virtual tree roots. (A vertex w ceases to be a tree root exactly when an instruction LINK(v, w) is executed.) Variable *number of vertices* counts the number of vertices so numbered; initially *number of vertices* = 0. Array *vertex* records the numbering; for each number i, *vertex*(i) is the vertex receiving number i. When a vertex w ceases to be a tree root, LINK saves its current label in *extra label*(w).

An Algol-like implementation of LINK(v, w) appears below. The implementation uses the procedure PRODUCT(y, z) to compute $y \odot z$. Eventually we shall solve the half-line case by modifying PRODUCT.

```
procedure LINK(v, w);
    begin
       comment this procedure assumes that size(0) = 0;
       size(v) := size(v) + size(w);
       extra label(w) := label(w);
       number of vertices := number of vertices + 1;
       vertex(number of vertices) := s := w;
       if size(v) < 2* size(w) then
            s ↔ child(v);
            root(v) := root(w); root(root(w)) := v fi;
loop1: if s ≠ 0 then
            parent(s) = v;
            while child(s) ≠ 0 do
                 label(child(s)) := PRODUCT(label(s), label(child(s)));
                 s := child(s); parent(s) = v od;
       s := child(v);
loop2: while 2* size(v) + size(child(s)) > 3* size(s) do
            parent(s) := v; child(v) := child(s);
            if child(s) = 0 then go to exit loop2
                 else label(child(s)) := PRODUCT(label(s), label(child(s)));
                      s := child(s) fi od; exit loop2:
    end LINK;
suitable procedure PRODUCT(y, z);
    PRODUCT := y ⊙ z;
```

Note that at the beginning of each iteration of *loop*2 it is always the case that $s = child(v)$, and that the test in *loop*2 succeeds exactly when $2 \cdot size(v) + size(child(s)) \ge 3 \cdot size(s)$, i.e., when $2 \cdot subsize(v) = 2(size(v) - size(s)) > size(s) - size(child(s)) = subsize(s)$.

The last instruction we must implement is EVAL(v). Let VT with subtree roots $r = r_0, r_1, \ldots, r_k$ be the virtual tree containing v, and let r_j be the root of the subtree containing v. We compute the value of v in three parts: the label of r (if $r \ne r_j$), the product of labels on the path of subtree roots from r to r_j (not including the labels of r and r_j), and the product of labels on the path from r_j to v. We compute the first part by looking up the label of r and the third part by using path compression on the subtree containing v.

Computing the second part of the vertex value, namely, $label(r_1) \odot label(r_2) \odot \cdots \odot label(r_{j-1})$, is more complicated. Here we use the fact that the sequence of instructions is given off-line. We perform the evaluation in an *auxiliary forest* AF, which is defined by

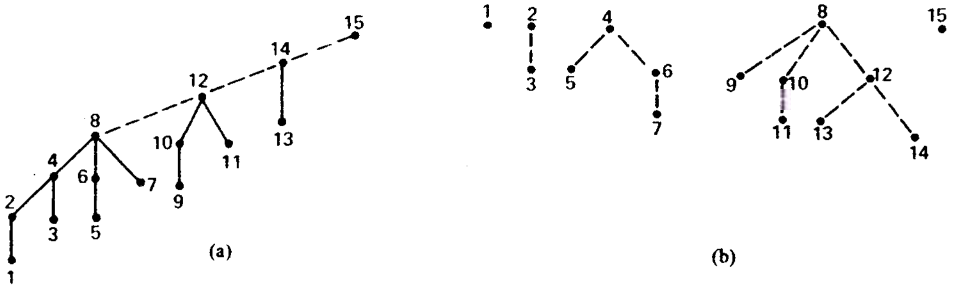


FIG. 10. Virtual and auxiliary forests after the instruction sequence LINK(2, 1), LINK(3, 2), ..., LINK(15, 14) is executed. (a) Virtual forest (one tree): Solid edges denote subtrees. (b) Auxiliary forest: Note that vertex 15 is still a tree root in the virtual forest; thus it is still a single-vertex tree in the auxiliary forest

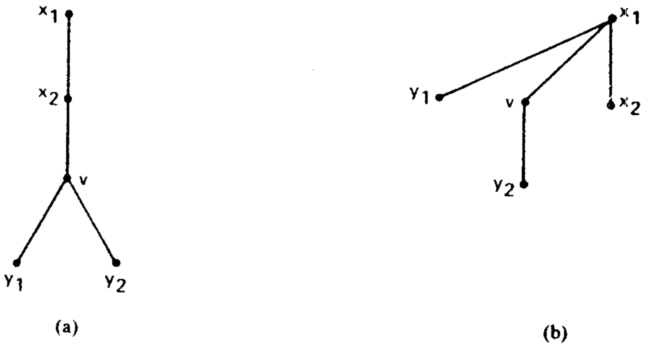


FIG. 11. Invalid path compression on auxiliary forest. (a) Original auxiliary tree: Paths to be evaluated are $x_1 \overset{*}{\rightarrow} y_1$ and $x_2 \overset{*}{\rightarrow} y_2$. (b) After compression of $x_1 \overset{*}{\rightarrow} y_1$ path: In the new tree x_2 is not an ancestor of y_2

the values of the child pointers as follows: Initially the auxiliary forest consists of a set of single-vertex trees, one for each vertex in the virtual forest. While a vertex x is a tree root in the virtual forest, it remains a single-vertex tree in the auxiliary forest. When x ceases to be a tree root in the virtual forest, the vertex $y = child(x)$ becomes the parent of x in the auxiliary forest. (If $child(x) = 0$ at this time, then x permanently remains a single-vertex tree in the auxiliary forest.) (See Figure 10.) Since $child(x)$ is numbered smaller than x for any x, the auxiliary forest is indeed a forest. Note also that once a vertex x ceases to be a virtual tree root, additional LINKs and UPDATEs do not change $child(x)$. Furthermore after x ceases to be a virtual tree root, $label(x)$ does not change until x ceases to be a subtree root. The array *extra label*(x), for each vertex x, thus records the value of $label(x)$ during the time when x is a virtual subtree root but not a virtual tree root.

If $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_k$ in the auxiliary forest, we define $\odot_{\mathrm{AF}}(x_0, x_k) = extra\ label(x_k) \odot extra\ label(x_{k-1}) \odot \cdots \odot extra\ label(x_0)$. It follows from the observations above that the second part of the vertex value, $label(r_1) \odot label(r_2) \odot \cdots \odot label(r_{j-1})$, is equal to $\odot_{\mathrm{AF}}(r_{j-1}, r_1)$. We can thus compute the second part of the vertex value for each EVAL(v) instruction by forming the product of *extra labels* along a path in the auxiliary forest. (The order of terms in the product is reversed from the usual order, however.) We would like to use path compression in the auxiliary forest to compute these products. Unfortunately, this requires reordering the products. (See Figure 11.)

The following lemma expresses the ordering we need.

LEMMA 3. *Let F be a forest. Let (x_i, y_i) for $i = 1, 2$ be a pair of vertex pairs such that $x_i \overset{*}{\rightarrow} y_i$ in F for $i = 1, 2$ and x_2 is not a proper descendant of x_1. If we carry out a path compression in F by making each vertex except x_1 on the path from x_1 to y_1 a child of x_1, then x_2 remains an ancestor of y_2.*

PROOF. Obvious. □

We compute the second parts of vertex values as follows. For each EVAL(v) instruction, we determine a pair of vertices (x_i, y_i) such that the second part of the value to be returned by EVAL(v) is $\odot_{AF}(x_i, y_i)$. We reorder the pairs so that, for any $i < j$, x_i is *not* a proper ancestor of x_j. Then, in a second pass, we compute $\odot_{AF}(x_i, y_i)$ for each pair by using path compression in the auxiliary forest.

We carry out EVAL(v) by means of the following steps. First, we assign a number to the EVAL instruction. Next, we determine the root of the subtree containing v, and the third part of the vertex value, by using path compression in the subtree. We store the third part of the vertex value in array *answer*, which is indexed by instruction number. Next, we determine the root of the virtual tree containing v. Finally, if the second part of the vertex value requires later computation, we store a triple consisting of the instruction number, the label of the tree root, and the child of the tree root in a *bucket* associated with the subtree root. This triple contains enough information to allow the second pass to complete the computation of the vertex value. If the second part of the vertex value does not require later computation (because the corresponding path is empty), we finish computing the vertex value and store the result in the *answer* array. The following procedure implements this method.

```
procedure EVAL(v);
    begin
        comment r is the root of the tree containing v, and s is the root of the subtree containing v;
        instruction := instruction + 1;
        if parent(v) = 0 then s := v; answer(instruction) := label(v)
            else COMPRESS(v); s := parent(v);
                answer(instruction) := PRODUCT(label(s), label(v)) fi;
        r := if child(s) = 0 then root(s) else root(root(s));
        if (r ≠ s) then
            if child(r) = s then
                answer(instruction) := PRODUCT(label(r), answer(instruction))
            else add ⟨instruction, label(r), child(r)⟩ to bucket(s) fi fi
    end EVAL;
```

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

```
procedure COMPRESS(v);
    if parent(parent(v)) ≠ 0 then
        COMPRESS(parent(v));
        label(v) := PRODUCT(label(parent(v)), label(v));
        parent(v) := parent(parent(v)) fi;
```

After carrying out EVAL, LINK, and UPDATE as described above, we must complete the computation of the vertex values. The procedure EXTRA__PASS appearing below does the job. For each triple $\langle i, l, y\rangle$ stored in the *bucket* of some vertex x, EXTRA__PASS computes, using path compression, the product of *extra labels* (excluding that of x) on the auxiliary tree path from x to y. This product is the second part of the vertex value for EVAL instruction i. EXTRA__PASS completes the computation of the vertex value and stores the result in *answer*(i). EXTRA__PASS unloads the buckets in decreasing order by vertex number. Since LINK numbers vertices in increasing order from the roots to the leaves in the auxiliary forest, EXTRA__PASS processes triples in an order which satisfies Lemma 3 and which is thus suitable for using path compression.

```
procedure EXTRA__PASS;
    for j := number of vertices by −1 until 1 do
        for each ⟨i, root label, v⟩ ∈ bucket(vertex(j)) do
            EXTRA COMPRESS(vertex(j), v);
            answer(i) := PRODUCT(extra label(v), answer(i));
            answer(i) := PRODUCT(root label, answer(i)) od od;
```

EXTRA__PASS uses the following recursive procedure to compress paths in the auxiliary forest. Recall that the order of terms in a product defining a value in the auxiliary forest is reversed from the normal order.

```
procedure EXTRA_COMPRESS($r$, $v$);
    if $child(v) \neq r$ then
        EXTRA_COMPRESS($r$, $child(v)$);
        $extra\ label(v)$ := PRODUCT($extra\ label(v)$, $extra\ label(child(v))$);
        $child(v) := child(child(v))$ fi;
```

This completes our description of the algorithm except for initialization of the variables. Initially $parent(v) = child(v) = 0$, $size(v) = 1$, $root(v) = v$, $bucket(v) = \emptyset$, and $label(v)$ is the initial label of v. Also *number of vertices* = *instruction* = 0.

The time required by this algorithm is $O(n + m)$ plus time for one path compression in the subtrees and one path compression in the auxiliary forest for each EVAL. If we can prove that both the set of subtrees and the auxiliary forest are balanced, then Theorem 2 gives an $O((m + n)\alpha(m + n, n))$ running time for the algorithm.

Consider first the subtrees. Let U be the uncompressed set of subtrees. To form U, we add an edge (x, y) to U when LINK assigns $parent(y) := x$. For any vertex x, let $subsize(x)$ be the number of descendants of x in U.

LEMMA 4. *If $x \rightarrow y \rightarrow z$ in U, then $subsize(x) \geq \frac{3}{2}\, subsize(z)$.*

PROOF. This proof resembles that of Lemma 2 in Section 5. LINK has only two ways to add edges to U. If (y, z) is added to U by *loop*2 of LINK, $subsize(y) \geq \frac{3}{2}\, subsize(z)$ just after the addition. Adding more edges to U preserves this relationship, and when (x, y) is added to U, $subsize(x) \geq subsize(y) \geq \frac{3}{2}\, subsize(z)$. Thus the lemma holds. A similar proof works if (x, y) is added to U by *loop*2.

Suppose, on the other hand, that both (x, y) and (y, z) are added to U by *loop*1. Just after (y, z) is added, $size(y) \geq 2 \cdot subsize(z)$. As long as y remains a subtree root, $size(y)$ cannot change. When (x, y) is added to U by *loop*1, all descendants of y in the virtual tree become descendants of x in the subtree. Thus $subsize(x) \geq 2 \cdot subsize(z)$ just after the addition of (x, y). Hence the lemma holds. □

COROLLARY 2. *The forest U is balanced for constants $a = \sqrt{3/2}$, $c = \sqrt{3/2}$.*

PROOF. Immediate from Lemma 4 and Theorem 3. □

Now consider the auxiliary forest. Let A be the auxiliary forest as it exists after all the EVALs, LINKs, and UPDATEs have been carried out but before the extra pass is executed.

THEOREM 4. *The forest A is balanced for constants $a = 2$, $c = 2$.*

PROOF. Consider a fixed height $h \geq 1$. We must show that A contains no more than $n/2^{h-1}$ vertices of height h. To accomplish this, we use a charging argument. Let v be a vertex of height h in A. Consider the time at which v first attains height h. Just before this time the virtual tree VT containing v has subtree roots $r_0, r_1, \ldots, r_h = v, \ldots, r_k$. The height of v increases to h when r_0 ceases to be a virtual tree root, which occurs because a LINK(x, r_0) operation is initiated. When this happens, we assign a *charge* of 2^{h-1} to the subtree with root v. Subtrees retain their charges as EVALs, LINKs, and UPDATEs are executed; when execution of a LINK combines two subtrees, the charge of the new subtree is the sum of the charges of the component subtrees.

We shall show that no subtree ever has charge exceeding the number of vertices in it, which implies that the total charge is at most n and that the number of vertices of height h does not exceed $n/2^{h-1}$. We call a subtree ST *good* if $charge(\mathrm{ST}) = 0$ or $|\mathrm{ST}| - charge(\mathrm{ST}) \geq 2^{h-1}$, *bad* otherwise. We claim that new charge can only be assigned in a tree all of whose subtrees are good. We prove the claim by induction on the sequence of EVAL, LINK, and UPDATE instructions.

The claim is certainly true before any charge is assigned. Consider a LINK that causes a new charge to be assigned for which the claim holds. Let r be the vertex about to attain height h in A and let $r_0, r_1, \ldots, r_h = v, \ldots, r_k$ be the sequence of subtree roots in the virtual tree containing v. Let ST_{h-1} and ST_h be the subtrees with roots r_{h-1}, r_h, respectively. By (v),

$|\mathrm{ST}_{h-1}| \geq 2^{h-1}$ and $|\mathrm{ST}_h| \geq 2^h$. Since the claim holds at this time, $|\mathrm{ST}_{h-1}| - \mathit{charge}(\mathrm{ST}_{h-1}) \geq 2^{h-1}$ and $|\mathrm{ST}_h| - \mathit{charge}(\mathrm{ST}_h) \geq 2^{h-1}$. After the charge for v is assigned, $|\mathrm{ST}_{h-1}| - \mathit{charge}(\mathrm{ST}_{h-1}) \geq 2^{h-1}$ and $|\mathrm{ST}_h| - \mathit{charge}(\mathrm{ST}_h) \geq 0$. It may now be the case that ST_h is bad.

No two vertices of height h in A are related in A. Thus no virtual tree can contain two different vertices of height h in A as subtree roots, unless one of them is the root of the virtual tree. It follows that no new charge can be assigned to the virtual tree containing ST_h until ST_h becomes a component of a larger subtree. Such a subtree must also contain ST_{h-1} and hence must be good. Thus no new charge can be assigned to the tree containing ST_h until all its subtrees are again good. The claim follows by induction, and the theorem follows from the claim. □

Theorem 2, Corollary 2, and Theorem 4 imply that the running time of the off-line algorithm is $O((m + n)\alpha(m + n, n))$. The storage space is $O(m + n)$, which exceeds the $O(n)$ storage requirements of the algorithms in the previous sections. It is possible to save a small constant factor in storage by combining arrays; we leave this as an exercise.

We now modify the algorithm to handle the half-line case. The modified algorithm consists of two stages. The information available to the first stage consists of the complete sequence of EVAL, LINK, and UPDATE instructions, excluding the initial vertex labels and the modifications to the vertex labels specified in the UPDATE instructions. After the first stage does a certain amount of preprocessing, the second stage must execute the sequence of instructions on-line.

The first stage does not actually compute vertex values. Instead, it constructs a *straight-line program*, consisting of a set of *assignments* of the form $x \leftarrow y \odot z$, which specifies how to compute the desired vertex values. The second stage executes the straight-line program, thereby computing the answers to all the EVALs.

For convenience we shall assume that the set of vertices in the forest is $\{v \mid 1 \leq v \leq n\}$. We shall use an array *value*(i) to represent the values to be computed by the straight-line program. For $1 \leq i \leq n$, i is an *input index*; *value*(i) will, during the second stage, be the label of vertex i in the real forest. For $i > n$, *value*(i) will be defined by an assignment *value*(i) $\leftarrow$ *value*(j) $\odot$ *value*(k); each noninput index will appear on the left side of exactly one such assignment. Associated with each EVAL(v) instruction will be an *output index* i such that *value*(i) is the value to be returned by the EVAL instruction. Some indices may be both input and output indices; if v is the root of a real tree when EVAL(v) is executed, then the index associated with EVAL(v) will be v. An output index which is not also an input index will appear on the right side of *no* assignment. To represent assignments, we shall use two arrays, *first* and *second*. If *value*(i) $\leftarrow$ *value*(j) $\odot$ *value*(k) is an assignment, then *first*(i) $= j$ and *second*(i) $= k$.

The first stage of the algorithm is identical to the off-line algorithm, except that instead of computing values the algorithm constructs the arrays *first* and *second* representing the straight-line program for computing values. The first stage also *ignores* all UPDATE instructions. Each *label*(v) now specifies not an element of S, but rather a position in the *value* array where the appropriate element of S will be stored during the second stage. Similarly, *answer*(i), for each EVAL instruction i, specifies not the appropriate vertex value but instead a position in *value* where the vertex value will be stored during the second stage.

To convert the off-line algorithm into the first stage of the half-line algorithm, all we must do is change the initialization and change the definition of PRODUCT to the following.

```
integer procedure PRODUCT(y, z);
    begin
        new index := new index + 1;
        first(new index) := y;
        second(new index) := z;
        PRODUCT := new index
    end PRODUCT;
```

During the second stage we ignore the LINK instructions and use the following procedures to carry out UPDATE and EVAL.

```
procedure UPDATE2($r$, $x$);
    $value(r) := x \odot value(r)$;
```

```
suitable procedure EVAL2($v$);
    begin
        $instruction := instruction + 1$;
        EVAL := COMPUTE($answer(instruction)$)
    end EVAL2;
```

Recursive procedure COMPUTE, which actually computes the vertex values, is defined by

```
suitable procedure COMPUTE($i$);
    if $value(i) \neq undefined$ then COMPUTE := $value\ (i)$
        else COMPUTE := $value(i)$ := COMPUTE($first(i)$) $\odot$ COMPUTE($second(i)$) fi;
```

The complete half-line algorithm, including initialization, consists of the following steps.

FIRST STAGE

Step 1. Initialize $parent(v) := child(v) = 0$, $size(v) := 1$, $bucket(v) := \varnothing$, $root(v) := label(v) := v$ for every vertex v. Initialize $number\ of\ vertices := instruction := 0$, $new\ index := n$.

Step 2. Carry out the EVAL and LINK instructions using the off-line procedures with the new version of PRODUCT. Ignore all UPDATE instructions.

Step 3. Execute EXTRA_PASS using the new version of PRODUCT.

SECOND STAGE

Step 4. Initialize $value(v) :=$ the initial label of v for every vertex v, $value(i) := undefined$ for $n < i \leq new\ index$, where *undefined* is a value distinguishable from every element of S.

Step 5. Carry out the EVAL and UPDATE instructions on-line using procedures EVAL2 and UPDATE2. Ignore all LINK instructions.

There is one tricky point in this algorithm, involving the question of *when* the assignments $value(i) \leftarrow value(j) \odot value(k)$ are actually carried out. For $i > n$, the second stage computes $value(i)$ only once. However, $value(i)$ for $i \leq n$ can change as UPDATEs are carried out, which might make $value(i)$ obsolete for some indices $i > n$.

Fortunately this is not the case. Consider the ith EVAL(v) instruction. The value to be returned by this instruction is the product of the labels of the vertices $r = v_0, v_1, \ldots, v_k = v$ on the tree path from the root r of the tree containing r to v. At the time of the EVAL, the labels of $v_1, \ldots, v_k$ cannot be affected by subsequent UPDATE instructions. The first stage defines the value that EVAL(v) returns by an assignment $value(answer(i)) \leftarrow value(r) \odot value(j)$, where $value(j)$ is defined by a set of assignments as the product of the labels of $v_1, \ldots, v_k$. It follows that each assignment executed by EVAL2(v) while computing $value(j)$ involves values that cannot be affected by subsequent UPDATEs. Thus $value(i)$ for each noninput, nonoutput index i need only be computed once.

The running time of the half-line algorithm is $O((m + n)\alpha(m + n, n))$. The storage space required is also $O((m + n)\alpha(m + n, n))$, since the total number of assignments that the first stage must encode is proportional to the total length of all the path compressions.

7. *Verifying Minimum Spanning Trees*

In this section we consider a simple application of the results in Sections 4, 5, and 6. Let $(S, \odot)$ be a commutative semigroup and let T be an unrooted n-vertex tree with an element of S, $label(v, w)$, associated with each edge (v, w). Given a set of m vertex pairs $\{\{v_i, w_i\} \mid 1 \leq i \leq m\}$, we wish to compute for each pair $\{v_i, w_i\}$ the product of labels on the path in T joining v_i and w_i.

We can solve this problem by using the least common ancestors algorithm of Section 4 and an appropriate EVAL-LINK-UPDATE method. First we arbitrarily choose a root r

for the tree T. Next, we compute $u_i = \mathrm{LCA}(v_i, w_i)$ for each pair $\{v_i, w_i\}$. Finally, we compute the product of labels on the paths from u_i to v_i and from u_i to w_i, and combine these products to give the answer for each pair $\{v_i, w_i\}$. The following procedure uses EVAL, LINK, and UPDATE to carry out this computation.

```
procedure EVALUATE_PATHS;
    begin
        for each $v \in V - \{r\}$ do
            create a tree with vertex $v$ having $label(parent(v), v)$ as its label;
            $bucket(v) := \emptyset$ od;
        for each $\{v, w\} \in pairs$ do add $\{v, w\}$ to $bucket(\mathrm{LCA}(v, w))$ od;
        SEARCH($r$)
    end EVALUATE_PATHS;
```

Recursive procedure SEARCH is defined by

```
procedure SEARCH($u$);
    begin
        for $v \in children(u)$ do SEARCH($v$) od;
        for $\{v, w\} \in bucket(u)$ do
            $answer(v, w)$ := if $u = v$ then EVAL($w$)
                              else if $u = w$ then EVAL($v$)
                              else EVAL($v$) $\odot$ EVAL($w$) od;
        for $v \in children(u)$ do LINK($u$, $v$) od
    end SEARCH;
```

This algorithm requires $O((m + n)\alpha(m + n, n))$ time. It is not hard to combine the computations of least common ancestors and of path values into a single traversal of the tree; we leave this as an exercise.

We shall give three applications of this algorithm. The first is to determine maximum flow values in a multiterminal network. Gomory and Hu [16] have given a method of constructing, for any undirected graph G with nonnegative edge capacities, an unrooted tree T with edge capacities such that T has the same vertices as G and the value of the maximum flow from v to w in G is equal to the minimum capacity of an edge on the path joining v and w in T, for any vertices v and w.

Assume that such a *cut tree* T is given for some graph G. If we let $(S, \odot)$ be the real numbers under minimization and we use the algorithm above in combination with the EVAL-LINK-UPDATE method of Section 5, we can compute maximum flow values for m pairs of vertices in $O((m + n)\alpha(m + n, n))$ time.

A second application is to verify minimum spanning trees. Suppose $G = (V, E)$ is a graph with real values $c(v, w)$ on its edges and that T is a spanning tree of G. T is a *minimum spanning tree* if $\sum_{(v,w)\in E'} c(v, w)$ is minimum among all spanning trees of G. We can verify that a spanning tree is minimum by using the following well-known lemma.

LEMMA 5. *T is minimum if and only if, for each edge $(v, w) \in E - E'$, $c(v, w) \geq \max\{c(x, y) \mid (x, y)$ is on the tree path joining v and $w\}$.*

If we let $(S, \odot)$ be the real numbers under maximization and we use the algorithm above in combination with the EVAL-LINK-UPDATE method of Section 5, we can compute the maximum cost of an edge along the tree path joining v and w for each nontree edge (v, w). We can then apply Lemma 5 to test whether T is minimum. This algorithm requires $O(m\alpha(m, n))$ time if G has m edges. It is interesting to note that the fastest known algorithms for actually *finding* a minimum spanning tree [6, 31] require $O(m \log \log n)$ time.

Our third application is to a problem of updating minimum spanning trees considered by Chin and Houck [7]. Before addressing this problem, we make a seemingly unrelated observation. If we use the first stage of the half-line algorithm in Section 6 to carry out the EVAL and LINK instructions in EVALUATE_PATHS, we can build a straight-line program for computing label products. We can visualize this program as a directed acyclic graph, with one node of in-degree zero for each edge of T, one node of in-degree two for each $\odot$ operation to be performed, and one node of out-degree zero for each pair $\{v_i, w_i\}$. This graph contains $O((m + n)\alpha(m + n, n))$ nodes, and there is a path from a node of in-degree zero representing an edge (x, y) to a node of out-degree zero representing a pair $\{v_i, w_i\}$ if and only if (x, y) is on the path joining v_i and w_i in T.

Normally we would run this straight-line program forward, assigning initial values to the nodes of in-degree zero and combining these values using $\odot$ operations until assigning a value to each node of out-degree zero. We shall solve the problem of updating minimum spanning trees by what amounts to running the straight-line program *backward*.

Suppose $G = (V, E)$ is a graph with edge values $c(v, w)$, and that $T = (V, E')$ is a minimum spanning tree of G. For each edge (v, w) of T, we wish to determine a nontree edge (x, y) by which (v, w) should be replaced to create a new minimum spanning tree if (v, w) is deleted from the graph. The proper replacement edges are specified by the following lemma.

LEMMA 6. *If (v, w) is a tree edge, then any nontree edge (x, y) of minimum cost such that (v, w) is on the tree path joining x and y is a suitable replacement for (v, w).*

Chin and Houck give an $O(n^2)$-time algorithm for finding replacement edges. We can find replacement edges in $O(m\alpha(m, n))$ time, as follows.

Step 1. Using the algorithm of this section in combination with the first stage of the half-line EVAL-LINK-UPDATE method of Section 6, construct a directed acyclic graph D with the following properties.

(a) D has $O(m\alpha(m, n))$ nodes and edges.
(b) Each tree edge (v, w) corresponds to a node $a(v, w)$ of in-degree zero in D.
(c) Each nontree edge (x, y) corresponds to a node $b(x, y)$ of out-degree zero in D.
(d) There is a path from $a(v, w)$ to $b(x, y)$ in D if and only if (v, w) is on the tree path joining x and y.

Step 2. Label each node $b(x, y)$ of out-degree zero in D by the corresponding edge (x, y). Process the remaining nodes of D in reverse topological order. To process a node c, examine the edges labeling the successors of c. Choose such an edge of minimum cost, and let this edge be the label of c.

After step 2 is completed, each node $a(v, w)$ of in-degree zero will be labeled by a nontree edge suitable for replacing tree edge (v, w). Step 1 of the algorithm requires $O(m\alpha(m, n))$ time; step 2 requires time proportional to the size of D [18, 22], which is $O(m\alpha(m, n))$. Thus the total running time is $O(m\alpha(m, n))$. The space required is also $O(m\alpha(m, n))$.

8. *Remarks*

The algorithms we have presented for the EVAL-LINK-UPDATE problem can easily be adapted to handle somewhat more general instruction types, such as the following.

GLINK(v, w): Combine the tree containing v with the tree having root w by making v the parent of w. (This instruction differs from LINK(v, w) in that v need not be a tree root.)

RELABEL(r, x): If x is the root of a tree, replace the label of r by x.

All our algorithms have a running time of $O((m + n)\alpha(m + n, n))$ or worse. It is natural to ask whether there is a faster, perhaps even linear-time algorithm. Such is not the case. For certain choices of $(S, \odot)$ it is impossible to find an algorithm that runs faster than $O((m + n)\alpha(m + n, n))$, assuming m is $\Omega(n)$ [29]. Furthermore the disjoint set union problem, when solved by any of a large class of pointer manipulation methods, requires $\Omega((m + n)\alpha(m + n, n))$ time if m is $\Omega(n)$ [28].

The remaining major open problem is to find an algorithm that will solve the on-line EVAL-LINK-UPDATE problem, for any semigroup $(S, \odot)$, in $O((m + n)\alpha(m + n, n))$ time. Recently Farrow [9] has presented an $O((m + n)\log^* n)$-time algorithm for this problem, where

$$\log^* n = \min\{i \mid \overbrace{\log \log \cdots \log}^{i \text{ times}} n \leq 1\}.$$

The author is presently developing an $O((m + n)\alpha(m + n, n))$-time algorithm, but it is substantially more complicated than the algorithms presented here.

In practice, the simple algorithm of Section 2 seems to be the best. Lengauer and the author [19] have used the algorithms of Section 2 and Section 5 to compute dominators in flow graphs. Our experiments comparing the two methods were inconclusive; the sophisticated algorithm was about 10 percent faster than the simple algorithm on small problems and about 20 percent faster on large problems. This small gain in speed is offset by the added complexity of the algorithm.

Appendix. Graph-Theoretic Terminology

A *graph* $G = (V, E)$ consists of a finite set V of *vertices* and a set E of *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*). If (v, w) is an undirected edge, v and w are *adjacent*. If (v, w) is a directed edge, v is a *predecessor* of w and w is a *successor* of v. 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 vertices $v = v_0, v_1, \ldots, v_k = w$ such that $(v_i, v_{i+1}) \in E$ for $0 \le i < k$. The path *contains* the edges (v_i, v_{i+1}) for $0 \le i < k$ as well as the vertices v_i for $0 \le i \le k$. The path is *simple* if $v_0, \ldots, v_k$ are distinct (except possibly $v_0 = v_k$) and the path is a *cycle* if $v_0 = v_k$. By convention there is a path of no edges from every vertex to itself, but a cycle must contain at least two edges. An undirected graph is *connected* if there is a path joining every pair of vertices. A graph is *acyclic* if it contains no cycles. A *topological ordering* of the vertices of an acyclic directed graph is an ordering such that, for any edge (v, w), v appears before w in the ordering.

An *unrooted tree* $T = (V, E)$ is an undirected, connected, acyclic graph. If an unrooted 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*. A *forest* is a set of vertex-disjoint rooted trees. 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 u, called the *least common ancestor* of v and w (denoted by $u = \mathrm{LCA}(v, w)$), such that u is on $T(v, w)$, $u \overset{*}{\rightarrow} v$, and $u \overset{*}{\rightarrow} w$. *The path* $T(v, w)$ consists of two parts, a path joining v and u containing descendants of u and ancestors of v, and a path joining u and w containing descendants of u and ancestors of w. A *leaf* of a rooted tree is a vertex with no children. The *height* of a vertex v in a rooted tree is the length of the longest path from v to a leaf.

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

REFERENCES

1. ACKERMANN, W. Zum Hilbershen Aufbau der reellen Zahlen. *Math. Ann.* 99 (1928), 118–133.
2. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
3. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. On computing least common ancestors in trees. *SIAM J. Comptng.* 5 (1976), 115–132.
4. AHO, A.V., AND ULLMAN, J.D. Node listings for reducible flow graphs. *J. Comptr. Syst. Sci. 13* (1976), 286–299.
5. ARDEN, B.W., GALLER, B.A., AND GRAHAM, R.M. An algorithm for equivalence declarations. *Comm. ACM 4*, 7 (July 1961), 310–314.

6. Cheriton, D., and Tarjan, R.E. Finding minimum spanning trees. *SIAM J. Comptng. 5* (1976), 724–742.
7. Chin, F.Y., and Houck, D.J. Algorithms for updating minimal spanning trees. *J. Comptr. Syst. Sci. 16* (1978), 333–344.
8. Chvátal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. STAN-CS-72-292, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1972.
9. Farrow, R. Efficient on-line evaluation of functions defined on paths in trees. Tech. Rep. 476-093-17, Dept. Math. Sci., Rice U., Houston, Tex., 1977.
10. Fischer, M.J. Efficiency of equivalence algorithms. In *Complexity of Computations*, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 153–168.
11. Fong, A., Kam, J., and Ullman, J.D. Application of lattice algebra to loop optimization. Conf. Rec. Second ACM Symp. Principles of Programming Languages, Palo Alto, Calif., 1975, pp. 1–9.
12. Galler, B.A., and Fischer, M.J. An improved equivalence algorithm. *Comm. ACM 7*, 5 (May 1964), 301–303.
13. Graham, S.L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Jan. 1976), 172–202.
14. Hopcroft, J.E. Private communication.
15. Hopcroft, J.E., and Ullman, J.D. Set-merging algorithms. *SIAM J. Comptng. 2* (1973), 294–303.
16. Hu, T.C. *Integer Programming and Network Flows*. Addison-Wesley, Reading, Mass., 1969, pp. 129–150.
17. Kennedy, K.W. Node listings applied to data flow analysis. Conf. Rec. Second ACM Symp. Principles of Programming Languages, Palo Alto, Calif., 1975, pp. 10–21.
18. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.
19. Lengauer, T., and Tarjan, R.E. A fast algorithm for finding dominators in a flow graph. To appear in *ACM Trans. Programming Languages and Syst.*
20. Paterson, M. Unpublished report, U. of Warwick, Conventry, England, 1972.
21. Tarjan, R.E. Depth-first search and linear graph algorithms. *SIAM J. Comptng. 1* (1972), 146–160.
22. Tarjan, R.E. Finding dominators in directed graphs. *SIAM J. Comptng. 5* (1974), 62–89.
23. Tarjan, R.E. Testing flow graph reducibility. *J. Comptr. and Syst. Sci. 9* (1974), 355–365.
24. Tarjan, R.E. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (April 1975), 215–225.
25. Tarjan, R.E. Applications of path compression on balanced trees. Tech. Rep. STAN-CS-75-512, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1975.
26. Tarjan, R.E. Solving path problems on directed graphs. Tech. Rep. STAN-CS-75-528, Comptr. Sci. Dept., Stanford U., Stanford, Calif., 1975.
27. Tarjan, R.E. Graph theory and Gaussian elimination. In *Sparse Matrix Computations*, J.R. Bunch and D.J. Rose, Eds., Academic Press, New York, 1976, pp. 3–22.
28. Tarjan, R.E. A class of algorithms which require non-linear time to maintain disjoint sets. To appear in *J. Comptr. and Syst. Sci.*
29. Tarjan, R.E. Complexity of monotone networks for computing conjunctions. *Annals Discrete Math. 2* (1978), 121–133.
30. Ullman, J.D. A fast algorithm for the elimination of common subexpressions. *Acta Informatica 2* (1973), 191–213.
31. Yao, A.C. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Inform. Processing Letters 4* (1975), 21–23.

RECEIVED SEPTEMBER 1975; REVISED NOVEMBER 1978