Reprinted from JOURNAL OF COMPUTER AND SYSTEM SCIENCES
All Rights Reserved by Academic Press, New York and London

Vol. 26, No. 3, June 1983
Printed in Belgium

A Data Structure for Dynamic Trees

DANIEL D. SLEATOR AND ROBERT ENDRE TARJAN

Bell Laboratories, Murray Hill, New Jersey 07974

Received May 8, 1982; revised October 18, 1982

A data structure is proposed to maintain a collection of vertex-disjoint trees under a sequence of two kinds of operations: a *link* operation that combines two trees into one by adding an edge, and a *cut* operation that divides one tree into two by deleting an edge. Each operation requires $O(\log n)$ time. Using this data structure, new fast algorithms are obtained for the following problems:

(1) Computing nearest common ancestors.

(2) Solving various network flow problems including finding maximum flows, blocking flows, and acyclic flows.

(3) Computing certain kinds of constrained minimum spanning trees.

(4) Implementing the network simplex algorithm for minimum-cost flows.

The most significant application is (2); an $O(mn \log n)$-time algorithm is obtained to find a maximum flow in a network of n vertices and m edges, beating by a factor of $\log n$ the fastest algorithm previously known for sparse graphs.

1. INTRODUCTION

In this paper we consider the following problem: We are given a collection of vertex-disjoint rooted trees. We want to represent the trees by a data structure that allows us to easily extract certain information about the trees and to easily update the structure to reflect changes in the trees caused by three kinds of operations:

link(v, w): If v is a tree root and w is a vertex in another tree, link the trees containing v and w by adding the edge(v, w), making w the parent of v.

cut(v): If node v is not a tree root, divide the tree containing v into two trees by deleting the edge from v to its parent.

evert(v): Turn the tree containing vertex v "inside out" by making v the root of the tree.

We propose a data structure that solves this *dynamic trees problem.* We give two versions of the data structure. The first has a time bound of $O(\log n)$ per operation when the time is amortized over a worst-case sequence of operations; the second,

0022-0000/83 $3.00

slightly more complicated, has a worst-case per-operation time bound of $O(\log n)$.[1] We use our data structure to devise new fast algorithms for the following graph-theoretic problems:

(1) Computing nearest common ancestors in $O(\log n)$ time per operation.

(2) Finding various kinds of network flows, including maximum flows in $O(nm \log n)$ time,[2] blocking flows in $O(m \log n)$ time, and acyclic flows in $O(m \log n)$ time.

(3) Computing certain kinds of constrained minimum spanning trees in $O(m \log n)$ time.

(4) Implementing the network simplex algorithm for the transportation problem so that updating a feasible tree solution takes $O(\log n)$ time per pivot step.

The paper consists of six sections. In Section 2 we formulate a precise version of the dynamic trees problem and briefly discuss variations of the problem. In Section 3 we present a high-level description of the first version of our data structure and carry out some preliminary running time analysis. The key idea presented in this section is the partitioning of each tree into a collection of vertex-disjoint paths. In Section 4 we discuss how to represent individual paths as biased trees and complete the description and analysis of the data structure. In Section 5 we develop the second version of our data structure. Section 6 contains applications, related work, and additional remarks. Our results extend and improve the preliminary work of Sleator and Tarjan [18].

2. The Dynamic Trees Problem

We shall consider the following version of the dynamic trees problem. We wish to maintain a forest of vertex-disjoint rooted trees,[3] each of whose edges has a real-valued *cost*, under a sequence of eight kinds of operations, which can be intermixed in any order (see Fig. 1):

parent(**vertex** v): Return the parent of v. If v has no parent (it is a tree root), return a special value **null**.

root(**vertex** v): Return the root of the tree containing v.

cost(**vertex** v): Return the cost of the edge $(v, parent(v))$. This operation assumes that v is not a tree root.

mincost(**vertex** v): Return the vertex w closest to $root(v)$ such that the edge

[1] If f and g are functions of x, the notation "$f(x)$ is $O(g(x))$" means there are positive constants c_1 and c_2 such that $f(x) \leqslant c_1 g(x) + c_2$ for all x.

[2] When discussing graph problems we denote by n the number of vertices and by m the number of edges in the graph.

[3] Our tree terminology is the same as that of Bent *et al.* [2], except that we use the term *external node* for a leaf of a binary tree.

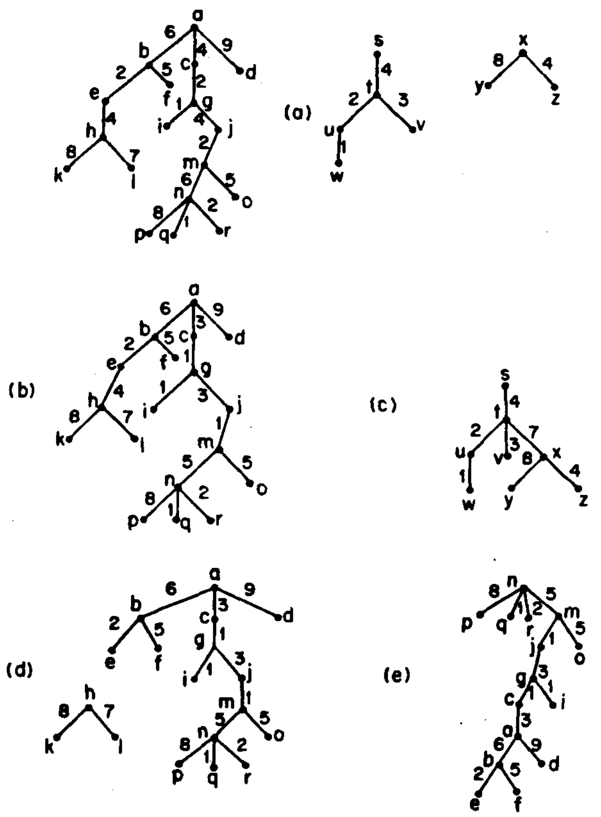

FIG. 1. Operations on dynamic trees. (a) Three trees. Operation *parent*(*n*) returns *m*, *root*(*n*) returns *a*, *cost*(*n*) returns 6, *mincost*(*n*) returns *g*. (b) Tree containing *n* after *update*(*n*, −1). (c) Tree formed by *link*(*x*, *t*, 7). (d) Trees formed by *cut*(*h*) on tree in part (b). Value returned is 4. (e) Tree formed by *evert*(*n*) on tree in part (d).

(w, parent(w)) has minimum cost among edges on the tree path from v to *root*(v). This operation assumes that v is not a tree root.

update(**vertex** v, **real** x): Modify the costs of all edges on the tree path from v to *root*(v) by adding x to the cost of each edge.

link(**vertex** v, w, **real** x): Combine the trees containing v and w by adding the edge (v, w) of cost x, making w the parent of v. This operation assumes that v and w are in different trees and v is a tree root.

cut(**vertex** v): Divide the tree containing vertex v into two trees by deleting the edge (v, *parent*(v)); return the cost of this edge. This operation assumes that v is not a tree root.

evert(**vertex** v): Modify the tree containing vertex v by making v the root. (This operation can be regarded as reversing the direction of every edge on the path from v to the original root.)

The operations *parent*, *root*, *cost*, and *mincost* extract information from the forest without altering it. The operation *update* changes edge costs but not the structure of the forest. The operations *link*, *cut*, and *evert* change the forest. These eight operations allow us to solve a number of graph-theoretic problems, as we shall see in Section 6.

The data structure we shall develop to support these operations can be modified to handle slightly different operations as well. Some possible variations are the following:

(1) We can drop the operation *evert*, allowing some simplification of the data structure. We have included the *evert* operation to allow representation of free (unrooted) trees: We represent each free tree by a rooted tree and apply *evert* as necessary to change tree roots. In applications involving rooted trees directly, this operation is generally unnecessary.

(2) We can add the operation *update edge*(v, x), which adds x to the cost of the edge $(v, \mathit{parent}(v))$. Note that if we allow *evert*, the operation *update edge*(v, x) can be simulated by the sequence $w := \mathit{root}(v)$, $\mathit{evert}(\mathit{parent}(v))$, $\mathit{update}(v, x)$, $\mathit{evert}(w)$.

(3) We can add the operation *update all*(v, x), which adds x to the cost of all edges in the tree with root v.

(4) We can associate costs with the vertices rather than with the edges.

(5) Instead of real-valued costs combined by minimization and updated by addition, we can allow the costs to the elements of an arbitrary (but fixed) semigroup, with the operations redefined appropriately. For a discussion of this generalization in the case that *link* is allowed but neither *cut* nor *evert* see Tarjan [20].

In our discussion of the dynamic trees problem we shall assume that the initial forest consists of n single-vertex trees; we shall use m to denote the total number of operations of the eight types. In stating time bounds we assume $n \geqslant 2$.

Before considering sophisticated solutions to the dynamic trees problem, it is worthwhile to examine the obvious solution: with each vertex v, we store its parent $p(v)$ and the cost of the edge$(v, p(v))$. Using this representation we can carry out a *parent*, *cost*, *link*, or *cut* operation in $O(1)$ time. The time for each of the other four operations is proportional to the length of the tree path from v to $\mathit{root}(v)$, which is $O(n)$ in the worst case.

By using an implicit representation of the structure of the forest, we can reduce the time for *root*, *min*, *update*, and *evert* to $O(\log n)$, at the cost of increasing the time for the other operations to $O(\log n)$. In the next three sections we develop two such implicit representations.

3. Dynamic Trees as Sets of Paths

We shall present our solution to the dynamic trees problem in a top-down fashion. We begin by assuming that we know how to solve a version of the problem for the special case in which the trees are paths. More precisely, suppose we know how to carry out an intermixed sequence of the following 11 kinds of operations on a collection of vertex-disjoint paths, each of whose edges has a real-valued cost:

path(**vertex** v): Return the path containing v. (We assume each path has a unique identifier.)

head(**path** p): Return the head (first vertex) of p.

tail(**path** p): Return the tail (last vertex) of p.

before(**vertex** v): Return the vertex before v on *path*(v). If v is the head of the path, return **null**.

after(**vertex** v): Return the vertex after v on *path*(v). If v is the tail of the path, return null.

pcost(**vertex** v): Return the cost of the edge (v, *after*(v)). This operation assumes that v is not the tail of *path*(v).

pmincost(**path** p): Return the vertex v closest to *tail*(p) such that (v, *after*(v)) has minimum cost among edges on p. This operation assumes that p contains more than one vertex.

pupdate(**path** p, **real** x): Add x to the cost of every edge on p.

reverse(**path** p): Reverse the direction of p, making the head the tail and vice versa.

concatenate(**path** p, q, **real** x): Combine p and q by adding the edge (*tail*(p), *head*(q)) of cost x. Return the combined path.

split(**vertex** v): Divide *path*(v) into (up to) three parts by deleting the edges incident to v. Return a list $[p, q, x, y]$, where p is the subpath consisting of all vertices from *head*(*path*(v)) to *before*(v), q is the subpath consisting of all vertices from *after*(v) to *tail*(*path*(v)), x is the cost of the deleted edge(*before*(v), v), and y is the cost of the deleted edge(v, *after*(v)). If v is originally the head of *path*(v), p is **null** and x is undefined; if v is originally the tail of *path*(v), q is **null** and y is undefined.

Using these path operations as primitives, we can develop a solution to the dynamic trees problem. We partition each tree into a collection of vertex-disjoint paths and carry out each tree operation by means of one or more path operations. We shall present two variations of this approach: *naive partitioning*, which gives an $O(\log n)$ amortized time bound per tree operation, and *partitioning by size*, which gives an $O(\log n)$ worst-case time bound per tree operation. In this and the next section we develop the naive partitioning method; in Section 5 we discuss partitioning by size.

Naive Partitioning

In the naive partitioning method, the partition of each tree into paths is determined not by the structure of the tree but by the sequence of tree operations so far performed. We partition the edges of each tree into two kinds, *solid* and *dashed*, with the property that at most one solid edge enters any vertex. (See Fig. 2.) Thus the solid edges define a collection of *solid paths* that partition the vertices. (A vertex with no incident solid edge is a one-vertex solid path.) The head of a path is its bottommost vertex; the tail is its topmost vertex.

We represent the dashed edges using the obvious method discussed at the end of Section 2: with each vertex v that is the tail of a solid path, we store $dparent(v)$, the parent of v (via the outgoing dashed edge), and $dcost(v)$, the cost of the edge $(v, parent(v))$. If v is a tree root, $dparent(v) = \textbf{null}$ and $dcost(v)$ is undefined. We manipulate the solid paths using the eleven path operations defined. In addition we need two composite operations (see Fig. 3):

splice(**path** p): Extend the solid path p by converting the dashed edge leaving $tail(p)$ to solid and converting the original solid edge entering $parent(tail(p))$ (if any) to dashed. Return the extended path. This operation assumes that $tail(p)$ is not a tree root (that is, there is a dashed edge leaving $tail(p)$).

expose(vertex v): Create a single solid path with head v and tail $root(v)$ by converting dashed edges to solid along the tree path from v to $root(v)$ and converting solid edges incident to this path to dashed. Return the resulting solid path.

We implement *splice* and *expose* as follows, where our algorithmic notation is a version of Dijkstra's guarded command language [5] augmented with functions and procedures and with the vertical bar "|" used in place of the box "▯":

```
function splice(path p):
        vertex v; path q, r; real x, y,
        v := dparent(tail(p));
1.      [q, r, x, y] := split(v);
        if q ≠ null → dparent(tail(q)), dcost(tail(q)) := v, x fi;
2.      p := concatenate(p, path(v), dcost(tail(p)));
        return if r = null → p
3.                | r ≠ null → concatenate(p, r, y)
                fi
end splice;
```

Note. Line 1 of *splice* converts to dashed the solid edges (if any) incident to $v = parent(tail(p))$. Line 2 converts to solid the dashed edge leaving $tail(p)$. Line 3

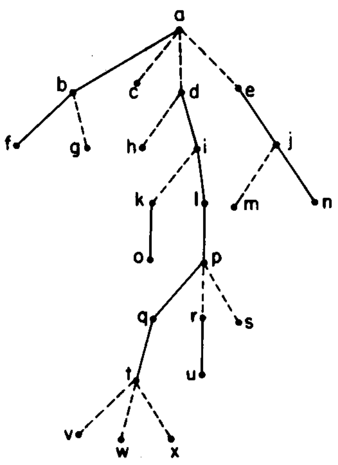

FIG. 2. A tree partitioned into solid paths. Path $|t, q, p, l, i, d|$ has head t and tail d.

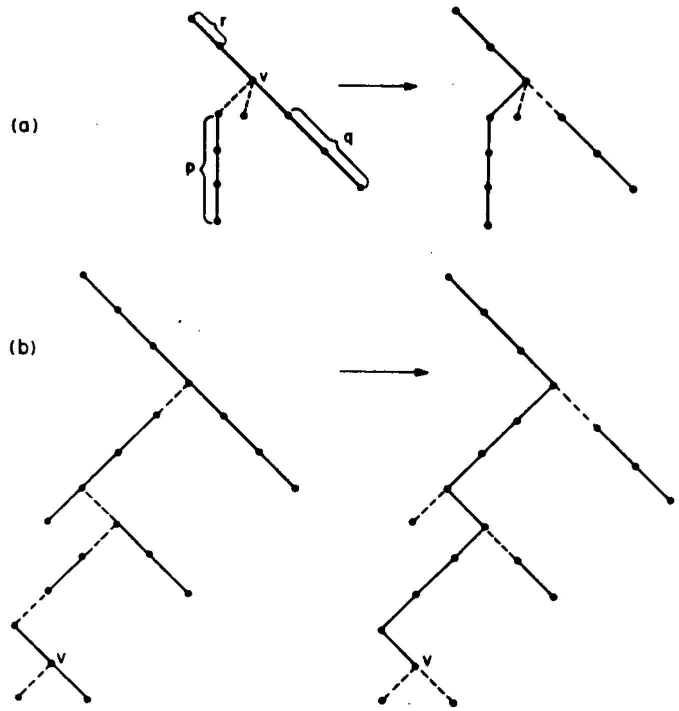

FIG. 3. Splice and expose. (a) The effect of *splice*(p). The letters "q," "r," and "v" refer to the corresponding variables in the program for splice. (b) The effect of *expose*(v).

converts to solid the dashed edge leaving v. To make this program robust, an error check should be added to ensure that on entry $dparent(tail(p)) \neq$ **null**. ∎

```
function expose(vertex $v$);
    path $p, q, r$; real $x, y$;
1.  $[q, r, x, y] := split(v)$;
    if $q \neq$ null $\rightarrow$ $dparent(tail(q)), dcost(tail(q)) := v, x$ fi;
    if $r =$ null $\rightarrow$ $p := path(v)$;
2.   | $r \neq$ null $\rightarrow$ $p := concatenate(path(v), r, y)$
    fi,
3.  do $dparent(tail(p)) \neq$ null $\rightarrow$ $p := splice(p)$ od;
    return $p$
end expose;
```

Note. Line 1 of expose converts to dashed the solid edges (if any) incident to v. Line 2 restores to solid the edge out of v if it has just become dashed. Line 3, the main part of *expose*, is a **do** loop that extends the path containing v by splicing until its tail is the tree root. ∎

We implement the eight tree operations as follows:

```
function parent(vertex $v$);
    return if $v = tail(path(v)) \rightarrow dparent(v)$
            | $v \neq tail(path(v)) \rightarrow after(v)$
            fi
end parent;

function root(vertex $v$);
    return $tail(expose(v))$
end root;

function cost(vertex $v$); .
    return if $v = tail(path(v)) \rightarrow dcost(v)$
            | $v \neq tail(path(v)) \rightarrow pcost(v)$
            fi
end cost;

function mincost(vertex $v$);
    return $pmincost(expose(v))$
end min;

procedure update(vertex $v$, real $x$);
    $pupdate(expose(v), x)$
end update;

procedure link(vertex $v, w$, real $x$);
    $concatenate(path(v), expose(w), x)$
end link;
```

```
function cut(vertex v);
    path p, q; real x, y;
    expose(v);
    [p, q, x, y] := split(v);
    dparent(v) := null;
    return y
end cut;

procedure evert(v);
    reverse(expose(v)); dparent(v) := null
end evert;
```

Remark. We allow a function to be used as a procedure. In such a use the value returned is ignored. Function *concatenate* is so used in *link*, and *expose* is so used in *cut*. There are simpler ways to implement *link* and *cut*; we have chosen these methods for technical reasons discussed in Section 4. ▮

Analysis of Expose

Having specified an implementation of the dynamic tree operations in terms of path operations, we can begin a running time analysis. At this level of detail, the only nontrivial task is to count the number of splice operations per expose. In the remainder of this section we shall derive an $O(n + m \log n)$ bound on the number of splices caused by a sequence of m tree operations. This bound implies that there are $O(\log n)$ splices per expose amortized over the sequence.

The $O(n + m \log n)$ bound on splices is implicit in the work of Galil and Naamad [8] and Shiloach [16], although they did not consider evert operations. Galil and Naamad obtained the bound by applying an upper bound for path comparison [19]; Shiloach gave a somewhat obscure direct proof. We shall give a simple direct proof that accommodates evert.

To carry out the proof we need one new concept. We define the *size* of a vertex v in the forest, denoted by *size*(v), to be the number of descendants of v, including v itself. We define a tree edge (v, *parent*(v)) to be *heavy* if $2 \cdot size(v) > size(parent(v))$ and *light*, otherwise. The following result is obvious:

LEMMA 1. *Let v be any vertex. Then $1 \leqslant size(v) \leqslant n$, there is at most one heavy edge entering v, and there are at most $\lfloor \lg n \rfloor$[4] light edges on the tree path from v to root(v).*

By considering the dashed versus solid and heavy versus light edge partitions, we can divide the edges into four classes: heavy dashed, heavy solid, light dashed, and light solid. By studying the effect of the various tree operations on these classes, we can bound the total number of splices. We call an operation *splice*(p) *special* if on entry to splice *parent*(*tail*(p)) is the head of a path and *normal*, otherwise. A special

[4] We use $\lg n$ to denote $\log_2 n$.

splice increases the number of solid edges by one; a normal splice leaves the number of solid edges unchanged.

THEOREM 1. *There are at most m special splices.*

Proof. Let $^\#solids$, $^\#exposes$, $^\#specials$, $^\#links$, and $^\#cuts$ be the number of solid edges, expose operations, special splices, links, and cuts, respectively, all as a function of time. There are $^\#links - {}^\#cuts$ tree edges; thus $^\#\text{solids} \leqslant {}^\#links - {}^\#cuts$. Lines 1 and 2 of *expose* decrease $^\#solids$ by at most one; line 3 of *expose* increases $^\#solids$ by at most one per special splice. Outside of *expose*, *link* increases $^\#solids$ by one and *cut* decreases $^\#solids$ by at most one. Thus $^\#solids \geqslant {}^\#specials + {}^\#links - {}^\#exposes - {}^\#cuts$, which means $^\#\text{specials} \leqslant {}^\#exposes + {}^\#solids - {}^\#links + {}^\#cuts \leqslant {}^\#exposes \leqslant m$. ∎

THEOREM 2. *There are at most $m(3\lfloor\lg n\rfloor + 1)$ splices.*

Proof. Call an operation *splice*(p) *light* if the edge $(tail(p), parent(tail(p)))$ is light and *heavy*, otherwise. During a single expose there are at most $\lfloor\lg n\rfloor$ light splices by Lemma 1, since all the corresponding light edges are on a single tree path. Thus there are $\lfloor\lg n\rfloor \cdot {}^\#exposes$ light splices altogether. To bound the number of heavy splices, we keep track of the number $^\#hs$ of heavy solid edges.

During an *expose*, each heavy splice increases $^\#hs$ by one, each light splice decreases $^\#hs$ by at most one, and lines 1 and 2 of *expose* decrease $^\#hs$ by at most one.

The operation *link*(v, w, x) increases the size of all nodes on the tree path from w to *root*(w), possibly converting edges on this path from light to heavy and edges incident to the path from heavy to light. After the operation *expose*(w) in *link*, all the edges incident to the path are dashed, and adding the edge (v, w) does not decrease $^\#hs$.

The operation *cut*(v) decreases the size of all nodes except v on the (original) tree path from v to *root*(v). Up to $\lfloor\lg n\rfloor$ of the edges on this path may become light; thus the *cut* may decrease $^\#hs$ by up to $\lfloor\lg n\rfloor + 1$ including the edge deleted by the cut.

The operation *evert*(v) changes the size of all nodes on the (original) tree path from v to root(v). After the operation *expose*(v) in *evert*, all the edges on this path are solid and all incident edges are dashed. Reversing the path may cause up to $\lfloor\lg n\rfloor$ of its edges to become light; thus the *evert* may decrease $^\#hs$ by up to $\lfloor\lg n\rfloor$.

We conclude that $^\#links - {}^\#cuts \geqslant {}^\#hs \geqslant {}^\#heavy\_splices - {}^\#light\_splices - {}^\#exposes - (\lfloor\lg n\rfloor + 1) \cdot {}^\#cuts + \lfloor\lg n\rfloor \cdot {}^\#everts$, where our notation is similar to that in the proof of Theorem 1. This means that $^\#heavy\_splices \leqslant {}^\#links - {}^\#cuts + {}^\#light\_splices + {}^\#exposes + (\lfloor\lg n\rfloor + 1) \cdot {}^\#cuts + \lfloor\lg n\rfloor \cdot {}^\#everts \leqslant m(2\lfloor\lg n\rfloor + 1)$, and the total number of splices is at most $m(3\lfloor\lg n\rfloor + 1)$. ∎

THEOREM 3. *In a sequence of m dynamic tree operations there are $O(m \log n)$ path operations (with splice and expose broken down into their component operations).*

Proof. The proof is immediate from Theorem 2. ∎

4. Dynamic Paths as Biased Binary Trees

To complete our solution we need a data structure to represent dynamic paths. For this purpose we use full binary trees. As usual we develop the data structure in an incremental fashion, introducing new ideas as we need them.

We represent each path by a binary tree whose external nodes in left-to-right order correspond to the vertices on the path from head to tail and whose internal nodes in symmetric order correspond to the edges in the path from head to tail. (See Fig. 4.) We shall generally not distinguish between a tree node and the corresponding vertex or edge on the path. Every node in the tree corresponds to the subpath whose vertices are its external descendants. Thus we can regard the root of the tree as identifying the path, and we shall generally not distinguish between the path and this root.

To facilitate the various path operations, we store with each node of a binary tree information about the path it represents. (See Fig. 5.) Each node v contains a bit *external*(v) indicating whether it is an external node and a pointer *bparent*(v) to its parent in the binary tree; if v is the root of the binary tree, *bparent*(v) = **null**.

Each internal node v contains four additional pointers: *bleft*(v) and *bright*(v), which point to the left and right child of v, and *bhead*(v) and *btail*(v), which point to the head and tail of the subpath corresponding to v (the leftmost and rightmost external descendants of v). To handle reversal, each internal node v also contains a bit *reversed*(v). We define the *reversal state* of v to be the exclusive or of the *reversed* bits on the path from v to the root of the binary tree; if the reversal state of v is **true**, the meanings of left and right are reversed at v. (Pointer *bleft*(v) points to the right child of v, *bhead*(v) points to the rightmost external descendant of v, and similarly for *bright* and *btail*.)

The internal nodes also contain information about the edge costs. If v is an internal node, we define *grosscost*(v) to be the cost of the corresponding edge on the path and *grossmin*(v) to be the minimum of *grosscost*(w) for w an internal descendant of v (*grosscost*(v) is the minimum cost of an edge on the subpath corresponding to v). We

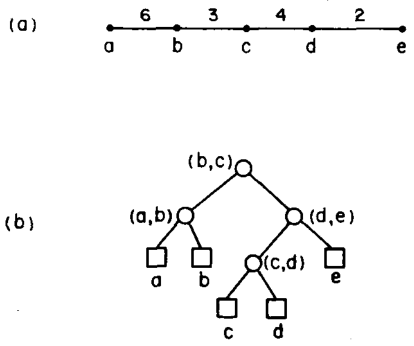

FIG. 4. A path and a binary tree representing it. (a) Path with head a and tail e. (b) Binary tree. External nodes (squares) are labeled with corresponding vertices, internal nodes (circles) with corresponding edges.

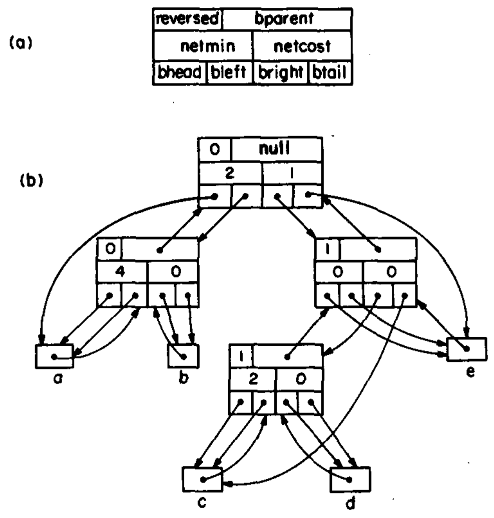

FIG. 5. Details of the binary tree representation. (a) Format of internal nodes, omitting *external* bits. (b) Data structure representing the path in Fig. 4.

represent *grosscost* and *mincost* implicitly by storing with each internal node v two values, *netcost*(v) and *netmin*(v), defined as follows:

$$netcost(v) = grosscost(v) - grossmin(v).$$

$$netmin(v) = \begin{cases} grossmin(v) & \text{if } v \text{ is a binary tree root,} \\ grossmin(v) - grossmin(bparent(v)), & \text{otherwise.} \end{cases}$$

The value of *netcost*(v) is nonnegative for any internal node v; the value of *netmin*(v) is nonnegative unless v is a binary tree root. We can compute *grossmin*(v) for any internal vertex by summing *netmin* on the path from v to the binary tree root, and *grosscost*(v) as *netcost*(v) plus *grossmin*(v).

Implementation of the Static Path Operations

This representation allows us to efficiently perform all the static path operations. (The static path operations are all those except *concatenate* and *split*, which change the structure of the paths.) We implement these operations as follows:

path(v): Follow *bparent* pointers from v until reaching a node w with *bparent*(w) = **null**, and return w. This operation takes time proportional to the depth of v in the binary tree containing it.

head(p): If *reversed*(p) is true, return *btail*(p); otherwise return *bhead*(p). This takes $O(1)$ time.

tail(p): Symmetric to *head*.

before(v): Traverse the binary tree path from v to *path*(v). Back up along this path computing the reversal state of each internal node on the path. Find the deepest node w on the path that is the right child of its parent. Return the rightmost external descendant of the sibling u of w. (Node u is the child of *bparent*(w) other than w; its rightmost external descendant is u if u is external, *bhead*(u) if u is internal with a **true** reversal state, and *btail*(u) if u is internal with a **false** reversal state.) This takes time proportional to the depth of v.

after(v): Symmetric to *before*.

pcost(v): Traverse the binary tree path from v to *path*(v). Back up along this path computing the reversal state and *grossmin* of each internal node on the path. Find the deepest node w on the path that is the left child of its parent. Return *grosscost*(*bparent*(w)), computed as *netcost*(*bparent*(w)) plus *grossmin*(*bparent*(w)). This takes time proportional to the depth of v.

pmincost(p): Starting from p, which is the root of a binary tree, proceed downward in the tree, keeping track of reversal states, until finding the node u last in symmetric order such that *grosscost*(u) = *grosscost*(p). This can be done by initializing u to be p and repeating the following step until u has *netcost* zero and its right child is either external or has positive *netmin*: If the right child of u is internal and has *netcost* zero, replace u by its right child, otherwise if u has positive *netcost*, replace u by its left child. Once u is computed, return the rightmost external descendant of its left child. This takes time proportional to the depth of u.

pupdate(p, x): Add x to *netmin*(p). This takes $O(1)$ time.

reverse(p): Negate *reversed*(p). This takes $O(1)$ time.

Remark. In some applications not requiring all the path operations, the remaining operations are easier to implement and we can drop some of the node fields. In particular, if we do not need *evert* we can drop the *reversed* bits and the *bhead* fields and carry out the parent function in $O(1)$ time by maintaining a *parent* field for all vertices instead of a *dparent* field just for the vertices with an outgoing dashed edge. Additional simplification is possible in this case. (See Sleator [17].) ∎

Implementation of the Dynamic Path Operations

In order to implement *concatenate* and *split*, we need four operations that assemble, take apart, and modify binary trees:

construct(**node** v, w, **real** x): Given the roots v and w of two binary trees and a real value x, combine the trees into a single tree by constructing a new root node with left child v, right child w, and *grosscost* x.

destroy(u): Given the root of a nontrivial binary tree, divide the tree into its component parts: the subtree whose root, say v, is the left child of u and the subtree whose root, say w, is the right child of u. Let x be the cost of edge u. Destroy node u and return the list $[v, w, x]$.

rotateleft(**node** v): Perform a single left rotation at node v. (See Fig. 6.) Node v must have an internal right child.

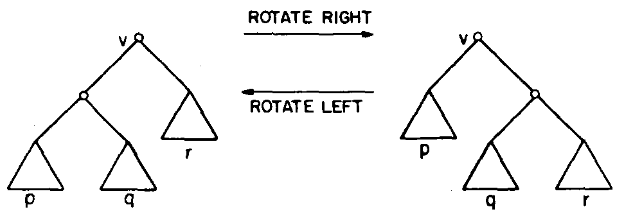

FIG. 6. A single rotation. Triangles denote subtrees.

rotateright(**node** *v*): Perform a single right rotation at node v. (See Fig. 6.) Node v must have an internal left child.

Note. Left rotation and right rotation are symmetric and are inverses of each other. ▮

It is easy to verify that with our binary tree representation each of these operations takes $O(1)$ time, including updating all the node fields. These operations suffice for concatenating and splitting paths represented by any of the standard classes of balanced binary trees, such as height-balanced trees [12], weight-balanced trees [15], or red-black trees [9]. For any of these classes, the depth of a tree of n external nodes is $O(\log n)$, and concatenation and splitting take $O(\log n)$ time. Thus any path operation takes $O(\log n)$ time. The splice operation also takes $O(\log n)$ time, but this is not true of expose. From Theorem 3 we obtain the following bound, which generalizes the corresponding bound of Galil and Naamad [8] and Shiloach [16] for a sequence of dynamic tree operations not including evert:

THEOREM 4. *With a representation of solid paths as balanced binary trees, a sequence of m dynamic tree operations takes* $O(m(\log n)^2)$ *time.*

In order to improve this result by a factor of $\log n$, we use biased binary trees [3] (see also the earlier paper [2]) to represent the solid paths. In a biased binary tree, we are allowed to specify a positive weight $wt(v)$ for each external node v. Each node v has an integer rank denoted by $rank(v)$, whose relevant properties for our purposes are the following (see Fig. 7):

(i) If v is external, $rank(v) = \lfloor \lg wt(v) \rfloor$. If v is any node, $rank(v) \leqslant 1 + \lfloor \lg wt(v) \rfloor$, where we inductively define the weight of an internal node to be the sum of the weights of its children.

(ii) If node w has parent v, $rank(w) \leqslant rank(v)$, with the inequality strict if w is external. If w has grandparent u, $rank(w) < rank(u)$.

LEMMA 2. *If v is an external node in a biased binary tree with root u, the depth of v is at most* $2(rank(u) - rank(v)) \leqslant 2 \lg(wt(u)/wt(v)) + 4$.

Proof. Property (ii) implies that the depth of v is at most $2(rank(u) - rank(v))$. Property (i) implies that $rank(u) - rank(v) \leqslant \lg(wt(u)/wt(v)) + 2$. Combining these gives Lemma 2. ▮

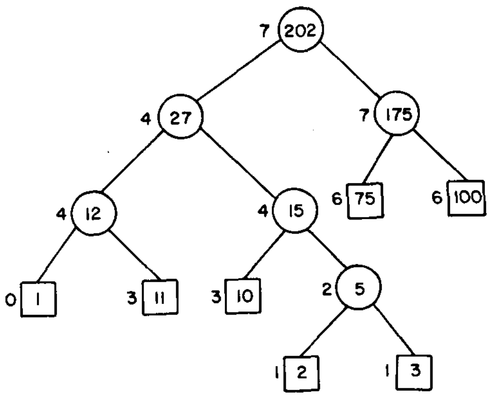

FIG. 7. A biased binary tree. Weights are inside nodes, ranks next to nodes.

Biased binary trees can be concatenated and split using node construction and destruction, single rotation, and the operation of increasing the rank of a node by one. In order to understand the time bounds for concatenation and splitting, we need the concept of *amortization by credits*. A credit represents one unit of computing time. To perform a concatenate or split we are given a certain number of credits. Spending one credit allows us to perform $O(1)$ computational steps. If we complete the operation before running out of credits we can save the unused credits to use on future operations. If we run out of credits before completing the operation we can spend credits previously saved. If we can perform a sequence of operations starting with no saved credits and without running out of credits before the sequence is complete, then the number of credits allocated for each operation gives an upper bound on its running time, amortized over the sequence. We call this the *amortized time* of the operation. The following bounds hold for the class of "locally" biased binary trees:

LEMMA 3 [3]. *A concatenation of two trees with roots p and q takes* $|rank(p) - rank(q)| + 1$ *credits and thus* $O(|rank(p) - rank(q)|)$ *amortized time. The root of the new tree has rank* $\max\{rank(p), rank(q)\}$ *or* $\max\{rank(p), rank(q)\} + 1$.

LEMMA 4 [3]. *An operation split(v) returning the list* $[q, r, x, y]$ *takes* $O(rank(p) - rank(v))$ *amortized time, where p is the root of the original binary tree containing v. If* $q \neq \textbf{null}$, $rank(q) \leqslant rank(p) + 1$; *if* $r \neq \textbf{null}$, $rank(r) \leqslant rank(p) + 1$. *As a side effect, the split leaves an excess of* $rank(p) - rank(q) + 1$ *credits if* $q \neq \textbf{null}$ *and* $rank(p) - rank(r) + 1$ *different tokens if* $r \neq null$. *These excess credits can be used for any purpose.*

To represent solid paths as biased binary trees we must define a weight for each vertex. Recall that in Section 3 we defined *size*(v) for a vertex v to be the number of descendants of v in its dynamic tree. We define the weight of v as

$$wt(v) = size(v) \text{ if no solid edge enters } v;$$
$$\qquad\qquad size(v) - size(w) \text{ if the solid edge } (w, v) \text{ enters } v.$$

LEMMA 5. *Every node v in a biased binary tree representing a solid path has $1 \leqslant wt(v) \leqslant n$, where n is the total number of vertices. Thus $0 \leqslant rank(v) \leqslant 1 + \lg n$.*

Proof. Let p be the root of the binary tree containing node v. The definition of weight implies that $1 \leqslant wt(v) \leqslant wt(p) = size(tail(p)) \leqslant n$. ▮

With each node in a biased tree we store its rank and weight. In order to use biased binary trees to represent solid paths, we must modify the implementations of some of the tree operations to update weights. With the implementation presented in Section 3, only *construct*, *rotateleft*, *rotateright*, *splice*, and *expose* need changing; *link*, *cut*, and *evert* are defined so that they do not change any node weight except inside *expose*. We augment *construct*, *rotateleft*, and *rotateright* to keep track of weights, and we augment *splice* and *expose* as follows, where the arrows denote the additions:

```
function splice(path p);
       vertex v; path q, r; real x, y;
       v := dparent(tail(p));
1.     [q, r, x, y] := split(v);
⇒      wt(v) := wt(v) − wt(p);
       if q ≠ null →
           dparent(tail(q)), dcost(tail(q)) := v, x;
⇒          wt(v) := wt(v) + wt(q)
       fi;
2.     p := concatenate(p, path(v), dcost(tail(p)));
       return if r = null → p
3.            | r ≠ null → concatenate(p, r, y)
              fi
end splice;

function expose(vertex v);
       path p, q, r; real x, y;
1.     [q, r, x, y] := split(v);
       if q ≠ null →
           dparent(tail(q)), dcost(tail(q)) := v, x;
⇒          wt(v) := wt(v) + wt(q)
       fi;
       if r = null → p := path(v)
2.     | r ≠ null → p := concatenate(path(v), r, y)
       fi;
3.     do dparent(tail(p)) ≠ null → p := splice(p) od;
       return p
end expose;
```

This completes our description of the data structure.

Running Time Analysis

Our main theorem bounds the running time of the method we have just developed.

THEOREM 5. *With naive partitioning and representation of solid paths as locally biased binary trees, a sequence of m dynamic tree operations takes $O(m \log n)$ time.*

Proof. Lemmas 2–5 imply that any path operation takes $O(\log n)$ time, worst case for the static path operations and amortized for *concatenate* and *split*. A splice also takes $O(\log n)$ amortized time. This means by Theorems 1 and 2 that the time for m dynamic tree operations is $O(m \log n)$ plus the time for the $O(m \log n)$ normal splices that take place during exposes. To bound the time for the normal splices, let us consider the ith normal splice that takes place during the **do** loop of an *expose*. (See Fig. 8.)

Let u be the tail of the path being extended by the splice, p the path containing u before the first splice, p' the path containing u before the ith splice, v the parent of u, s the path containing v before the first splice, and s' the path containing v (and u) after the ith splice. We also use p, p', s, s' to denote the roots of the binary trees representing the corresponding paths. We shall prove that the splice has an amortized time bound of $O(rank(s) - rank(p) + rank(s') - rank(p'))$. In the process we shall prove that $rank(s) \geqslant rank(p)$ and $rank(s') \geqslant rank(p')$.

Let $wt(v)$ be the weight of v before the splice and $wt'(v)$ the weight of v after the splice; similarly for $rank(v)$ and $rank'(v)$. Then $rank(s) \geqslant rank(v) + 1 = 1 + \lfloor \lg wt(v) \rfloor \geqslant 1 + \lfloor \lg wt(p) \rfloor \geqslant rank(p)$. (Since the splice is normal, $s \neq v$. Since v is an external node, it has rank strictly less than that of its parent by property (ii). The remaining inequalities follow from property (i) and the definition of weights.) The same argument shows that $rank(s) \geqslant rank(p')$.

The amortized time for the split in line 1 of *splice* is $O(rank(s) - rank(v)) = O(rank(s) - rank(p))$. Since the split is normal, it leaves an excess of $rank(s) - rank(q) + 1 \geqslant 0$ credits, where q is as defined in the implementation of splice. We have $rank(s') \geqslant rank'(v) + 1 \geqslant 1 + \lfloor \lg wt'(v) \rfloor \geqslant 1 + \lfloor \lg wt(q) \rfloor \geqslant rank(q)$. Furthermore, $rank(s') \geqslant rank(p')$ by Lemma 3. The number of credits needed for the

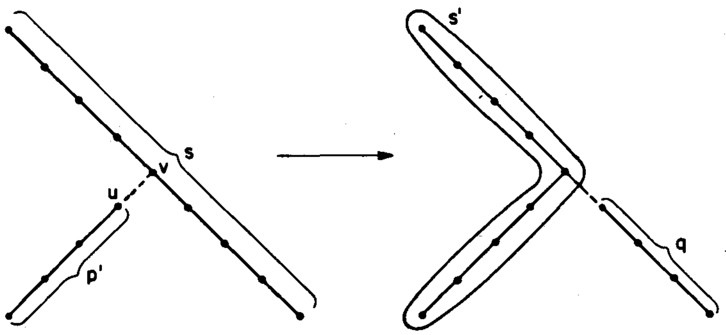

FIG. 8. A normal splice.

concatenation in line 2 of splice is $|rank(p') - rank'(v)| + 1 \leqslant 2\,rank(s') - rank(p') - rank'(v) + 1 \leqslant rank(s') - rank(p') + rank(s') - rank(q) + 2$, since property (i) implies $rank'(v) \geqslant rank(q) - 1$. Since we have $rank(s) - rank(q) + 1$ credits on hand, the number of new credits we need is $rank(s') - rank(p') + rank(s') - rank(s) + 1 \leqslant 2(rank(s') - rank(p')) + 1$.

The last operation is the concatenation in line 3 of splice. If it takes place, there are $rank(s) - rank(r) + 1 \geqslant 0$ credits still available from the split in line 1, where r is as defined in the implementation of splice. The number of credits needed for the concatenation is at most $2\,rank(s') - rank(p') - rank(r) + 1$ since $rank(s') \geqslant \max\{rank(p'), rank(r)\}$ by Lemma 3. Thus the number of new credits needed is at most $2(rank(s') - rank(p'))$ by the same argument.

Combining the estimates for the split and the two concatenates, we obtain the claimed bound of $O(rank(s) - rank(p) + rank(s') - rank(p'))$ on the amortized time of the splice. Note that $rank(s) \geqslant rank(p)$ and $rank(s') \geqslant rank(p')$. If we sum this estimate over all the normal splices that take place during a single expose, the sum telescopes, and Lemma 5 gives a bound of $O(k + \log n)$ for the amortized time taken for the entire expose, where k is the number of normal splices. From this and Theorem 2 we obtain an $O(m \log n)$ bound on the time for the normal splices during all m dynamic tree operations, and hence an $O(m \log n)$ bound on the total time for the tree operations. ▮

5. Partitioning by Size

The data structure described in Sections 3 and 4 is a good one if amortized running time is the relevant complexity measure, which is the case in most of the applications known to the authors. If worst-case per-operation running time is important, we can modify the data structure so that each individual dynamic tree operation takes $O(\log n)$ time. In this section we present the required modifications. The modified structure is more complicated than the original, meaning that the original is likely to perform better in practice. Thus the new structure is mainly of theoretical interest. We shall omit some of the details, concentrating on the main ideas.

As in Section 3, we decompose each tree into vertex-disjoint solid paths, but we define the solid edges according to the tree structure rather than according to the sequence of previously performed tree operations. Specifically, we call an edge (v, w) solid if and only if it is heavy, namely, if and only if $2 \cdot size(v) > size(w)$. Thus, whereas in Section 3 we used light and heavy edges only as an analytical tool, here they become part of the data structure. With this definition a tree path from a vertex v to $root(v)$ contains at most $\lfloor \lg n \rfloor$ dashed edges by Lemma 1 and thus intersects at most $\lfloor \lg n \rfloor + 1$ solid paths.

We implement the dynamic tree operations using *expose* as we did in Section 3. A new problem arises: an expose can convert light edges to solid and heavy edges to dashed, violating the definition of solid edges. Thus we must conclude each tree operation by repairing the damage caused by the *expose*. For this purpose we use the following operation:

conceal(**path** *p*): Convert to dashed every light edge on path *p*, and to solid every heavy edge incident to *p*, including the heavy edge, if any, incident to *head*(*p*).

We implement the eight tree operations as follows:

```
function parent(vertex v);
    (same as in Section 3)
function root(vertex v);
    path p;
    p := expose(v);
    v := tail(p);
    conceal(p);
    return v
end root;
function cost(vertex v);
    (same as in Section 3)
function mincost(vertex v);
    path p; real x;
    p := expose(v);
    x := pmincost(p);
    conceal(p);
    return x
end min;
function update(vertex v, real x);
    path p;
    p := expose(v);
    pupdate(p, x);
    conceal(p)
end update;
procedure link(vertex v, w, real x);
    conceal(concatenate(path(v), expose(w), x))
end link;
function cut(vertex v);
    path p, q, r; real x, y;
    p := expose(v);
    [q, r, x, y] := split(v);
    dparent(v) := null;
    conceal(r);
    conceal(path(v));
    return y
end cut;
```

Note. In the program for *cut*, the exposed path p is broken by the cut into two parts, the path consisting only of vertex v and the path r. Both parts must be concealed. ▌

```
procedure evert(v);
    path p;
    p := expose(v);
    reverse(p);
    dparent(v) := null;
    conceal(p)
end evert;
```

New Features of the Data Structure

In order to implement *conceal* efficiently, we must make three changes in the data structure. The first change is to add two fields to each internal node in a binary tree. (See Fig. 9.) If u is an internal node corresponding to an edge (v, w) on the solid path p, we define $lefttilt(u)$ to be the sum of weights of vertices on the subpath of p from $head(p)$ to v minus the weight of w. The condition that (v, w) is heavy is equivalent to $lefttilt(u) > 0$. We define $leftmin(u)$ to be the minimum of $lefttilt(t)$ for t an internal descendant of u. We need *leftmin* to locate light edges during *conceal* operations. To handle *reverse*, we also need the symmetric values $righttilt(u)$, defined to be the sum of weights of vertices on the subpath of p from w to $tail(p)$ minus the weight of u, and $rightmin(u)$, defined to be the minimum of $righttilt(t)$ for t an internal descendant of u.

To represent these values, we use two fields for each internal node u:

$$netleftmin(u) = \begin{cases} leftmin(u) & \text{if } u \text{ is a binary tree root;} \\ leftmin(u) - leftmin(bparent(u)), & \text{otherwise.} \end{cases}$$

$$netrightmin(u) = \begin{cases} rightmin(u) & \text{if } u \text{ is a binary tree root;} \\ rightmin(u) - rightmin(bparent(u)), & \text{otherwise.} \end{cases}$$

As with *bleft* and *bright*, and *bhead* and *btail*, the values of $netleftmin(u)$ and $netrightmin(u)$ are interchanged if the reversal state of u is **true**.

Starting from the binary tree root corresponding to a solid path p and proceeding downward in the tree, we can compute *leftmin* and *rightmin* for each internal node visited in $O(1)$ time per node. We can also compute *lefttilt* and *righttilt* for each node visited in $O(1)$ time per node, using the following observations: If u is an internal node representing an edge (v, w), both v and w are accessible from u in $O(1)$ time via *bleft*, *bright*, *bhead*, and *btail*. Thus $wt(v)$ and $wt(w)$ are available in $O(1)$ time. Furthermore the sum of weights of vertices before w (after v) on p is the sum of weights of internal nodes t such that t is the left (right) child of an ancestor of u but t is not an ancestor of u. We can compute these two sums for each internal node u in $O(1)$ time per node while descending from the tree root. Henceforth we shall assume

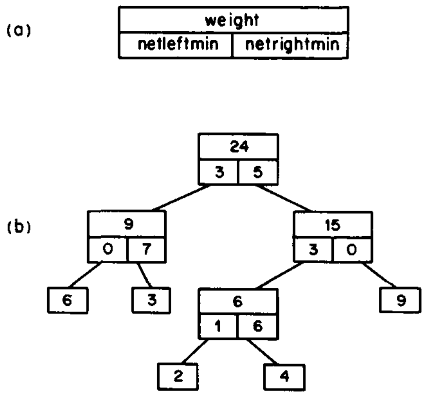

FIG. 9. Fields to represent *lefttilt* and *righttilt*. (a) Format of internal nodes, omitting *external* bits and fields shown in Fig. 5. (b) Values of fields representing a path whose vertices have weights 6, 3, 2, 4, 9, assuming all *reversed* bits are **false**.

that *leftmin*(u), *rightmin*(u), *lefttilt*(u), and *righttilt*(u) are available whenever we visit an internal node u.

It is easy to augment *construct*, *destroy*, *rotateleft*, and *rotateright* so that they update the *netleftmin* and *netrightmin* fields and still take $O(1)$ time. We need one additional path operation:

light(**path** p): Return the vertex w closest to *tail*(p) such that (*before*(v), v) is light; return **null** if there is no such vertex.

We implement *light* in the same way as *pmincost*. Starting from the root of the binary tree representing p, we proceed downward, using *leftmin* and *lefttilt* to guide the descent, until reaching the internal node u last in symmetric order such that *lefttilt*(u) $\leqslant 0$; then we return as w the leftmost external descendant of the right child of u. This takes time proportional to the depth of w in the binary tree; if **null** is returned, the time is $O(1)$.

The second change is to use globally biased instead of locally biased binary trees to represent the solid paths. Globally biased binary trees [3] have the same static properties as locally biased trees, namely (i) and (ii) of Section 4, but *concatenate* and *split* have better worst-case running times, given in Lemmas 6 and 7.

LEMMA 6 [3]. *An operation concatenate*(p, q, x) *takes* $O(\max\{rank(p), rank(q)\} - \max\{rank(v), rank(w)\})$ *time, where* $v = tail(p)$ *and* $w = head(q)$, *and produces a tree whose root has rank* $\max\{rank(p), rank(q)\}$ *or* $\max\{rank(p), rank(q)\} + 1$.

LEMMA 7 [3]. *An operation split*(v) *takes* $O(rank(p) - rank(v))$ *time, where* $p = path(v)$.

We use the same definition of weight as in Section 4. Lemmas 2 and 5 remain valid; thus any single path operation takes $O(\log n)$ time.

The third change is to add an extra part to our data structure. For each vertex v, we maintain a *path set* containing each path p such that $(tail(p), v)$ is a dashed edge. We manipulate path sets by means of three operations:

maxwt(**vertex** v): Return the path of maximum weight in the path set of v; return **null** if the path set is empty. (Recall that the weight of a path is the sum of the weights of its vertices.)

insert(**path** p, **vertex** v): Insert path p into the path set of v.

delete(**path** p, **vertex** v): Delete path p from the path set of v.

We represent the path set of a vertex by a globally biased binary tree, with the paths appearing as external nodes in left-to-right order by decreasing weight. The weight of a path is also used to determine its rank in the tree. For each internal node we maintain a pointer to its leftmost leaf descendant. With this representation the time to perform a *maxwt* operation is $O(1)$, and the time to perform $insert(p, v)$ or $delete(p, v)$ is $O(\log(W/wt(p)))$, where W is the sum of the weights of the paths in the path set of v before the operation. The bounds for *insert* and *delete* follow from Lemmas 6 and 7 and the ordering of paths in the path set by weight.

Implementation and Analysis of Expose and Conceal

Since each path operation takes $O(\log n)$ time, so does each dynamic tree operation, not counting the time for *expose* and *conceal* operations. We have implemented the dynamic tree operations so that the only necessary manipulation of weights and path sets takes place inside *expose* and *conceal*. Thus it remains for us to implement and analyze *expose* and *conceal*. We implement *expose* as in Section 4, with additional statements to update the path sets. Here are the details, with the additions indicated:

```
function splice(path p);
    vertex v; path q, r; real x, y;
    v := dparent(tail(p));
    [q, r, x, y] := split(v);
    wt(v) := wt(v) − wt(p);
⇒   delete(p, v);
    if q ≠ null →
        dparent(tail(q)), dcost(tail(q)) := v, x;
        wt(v) := wt(v) + wt(q);
⇒       insert(q, v)
    fi
    p := concatenate(p, path(v), dcost(tail(p)));
    return if r = null → p
              | r ≠ null → concatenate(p, r, y)
           fi
end splice;
```

```
function expose(vertex v);
    path p, q, r; real x, y;
    [q, r, x, y] := split(v);
    if q ≠ null →
        dparent(tail(q)), dcost(tail(q)) := v, x;
        wt(v) := wt(v) + wt(q);
⇒       insert(q, v)
    fi;
    if r = null → p := path(v)
     | r ≠ null → p := concatenate(path(v), r, y)
    fi;
    do dparent(tail(p)) ≠ null → p := splice(p) od;
    return p
end expose;
```

THEOREM 6. *An expose operation takes $O(\log n)$ time.*

Proof. The argument is much like that in the proof of Theorem 5. An *expose* operation takes $O(\log n)$ time plus time for $O(\log n)$ *splice* operations (one per light edge on the dynamic tree path from the exposed vertex to the root). Consider the ith splice. Let u be the tail of the path being spliced, p the path containing u before the first *splice*, p' the path containing u just before the ith splice, v the parent of u, s the path containing v before the first splice, and s' the path containing v (and u) after the ith splice. (See Fig. 8.)

The *split* operation in *splice* takes $O(rank(s) - rank(v))$ time. The *delete* operation takes $O(\log(wt(v)/wt(p)))$ time. The *insert* operation, if performed, takes $O(\log wt(v)/wt(q)) = O(1)$ time, where q is the inserted path, since the edge $(tail(q), v)$ is heavy. It follows from Lemmas 1 and 6 that the one or two *concatenate* operations needed to complete the splice take $O(rank(s') - rank'(v))$ time, where $rank'(v)$ is the rank of v after the ith splice.

We also have $rank(p) \leqslant 1 + \lfloor \lg wt(p) \rfloor \leqslant 1 + \lfloor \lg wt(v) \rfloor \leqslant 1 + rank(v)$. Thus the *split* takes $O(rank(s) - rank(p))$ time, and summed over all splices the total split time is $O(\log n)$. Similarly since $wt(v) \leqslant wt(s)$ the *delete* takes $O(\log(wt(s)/wt(p)))$ time, and summed over all splices the delete time is $O(\log n)$. Finally, $rank(p') \leqslant 1 + \lfloor \lg wt(p') \rfloor \leqslant 1 + \lfloor \lg wt'(v) \rfloor \leqslant rank'(v)$, where $wt'(v)$ is the weight of v after the ith splice, since $wt(p') = wt(p) \leqslant size(v)/2$, which means $wt(p') \leqslant size(v) - wt(p) \leqslant wt'(v)$. Thus the concatenates take $O(rank(s') - rank(p'))$ time, and summed over all splices the concatenate time is $O(\log n)$. Combining these estimates we obtain the theorem. ∎

An operation *conceal*(p) is just like an *expose* run backwards. The only complication is that we must find the light edges on p, which we do using *pmintilt*, and the heavy edges incident to p, which we do using the path sets. The main loop of *conceal* proceeds backward (downward) along p, performing the inverse of a splice, called a *slice*, each time it encounters a light edge. The behavior of *slice* is as follows:

function *slice*(**path** *p*): Convert the edge *light*(*p*) to dashed, dividing *p* into two paths, say *q* and *r*. If any edge entering *head*(*r*) is heavy, make it solid. Return the path *q*. This operation assumes that *p* contains at least one light edge, i.e., $light(p) \neq$ **null**.

The programs below implement *slice* and *conceal*. Their correctness is easy to verify.

```
function slice (path p);
    vertex v; path q, r, s; real x, y;
    $v := light(p)$;
    $[p, r, x, y] := split(v)$;
    $s :=$ if $r =$ null $\rightarrow path(v) \mid r \neq$ null $\rightarrow concatenate(path(v), r, y)$ fi;
    $dparent(tail(p)), dcost(tail(p)) := v, x$;
    $wt(v) := wt(v) + wt(p)$;
    $q := maxwt(v)$;
    if $2 \cdot wt(q) > wt(v) \rightarrow$
        $wt(v) := wt(v) - wt(q)$;
        $delete(q, v)$;
        $concatenate(q, s, dcost(tail(q)))$
    fi;
    $insert(p, v)$;
    return p
end slice;

procedure conceal(path p);
    vertex v; path q, r, s; real x, y;
    do $light(p) \neq$ null $\rightarrow p := slice(p)$ od;
    $v := head(p)$
    $s := maxwt(v)$;
    if $s \neq$ null and $2 \cdot wt(s) > wt(v) \rightarrow$
        $[q, r, x, y] := split(v)$;
        $wt(v) := wt(v) - wt(s)$;
        $delete(s, v)$;
        $s := concatenate(s, path(v), dcost(tail(s)))$;
        if $r \neq$ null $\rightarrow concatenate(s, r, y)$ fi
    fi
end conceal;
```

THEOREM 7. *A conceal* operation takes $O(\log n)$ *time.*

Proof. The proof is like that of Theorem 6. A *conceal* operation takes $O(\log n)$ time plus time for $O(\log n)$ *slice* operations, since the *light* operation preceding each *splice* is repeated inside the *slice*. Consider a typical slice. Let *p* be the path sliced, let $v = light(p)$, let p' be the path containing *v* after the slice (and thus after the

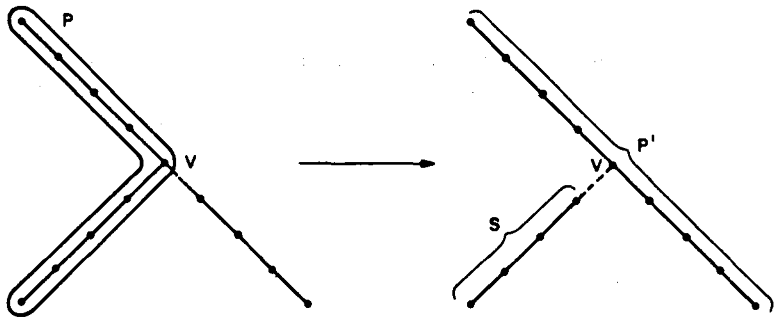

FIG. 10. A slice.

conceal), let s be the path returned by the slice, and let s' be the path containing $tail(s)$ after the *conceal*. (See Fig. 10.)

The *light* and *split* operations take $O(rank(p) - rank(v))$ time. The *delete* operation, if performed, takes $O(\log(wt(v)/wt(q))) = O(1)$ time, where q is the deleted path, since the edge $(tail(q), v)$ is heavy. The one or two *concatenate* operations take $O(rank(p') - rank'(v))$ time, where $rank'(v)$ is the rank of v after the slice. The *insert* operation takes $O(\log(wt'(v)/wt(s)))$, where $wt'(v)$ is the weight of v after the slice.

Since $(tail(s), v)$ is light, $wt(s) \leqslant wt(v)$. Thus $rank(s) \leqslant 1 + \lfloor \lg wt(s) \rfloor \leqslant 1 + \lfloor \lg wt(v) \rfloor = 1 + rank(v)$, and the *light* and *split* operations take $O(rank(p) - rank(s))$ time, which sums to $O(\log n)$ over all splices. Also $rank(s') \leqslant 1 + \lfloor \lg wt(s') \rfloor \leqslant 1 + \lfloor \lg wt'(v) \rfloor = 1 + rank'(v)$, and the *concatenate* operations take $O(rank(p') - rank(s'))$ time, summing to $O(\log n)$ over all splices. Finally, since $wt(s) = wt(s')$ and $wt(v) \leqslant wt(p')$, the time for inserts sums to $O(\log n)$ over all splices. This gives the theorem. ▮

From Theorems 6 and 7 and the $O(\log n)$ time bound for individual path operations, we obtain the following worst-case version of the main theorem:

THEOREM 8. *With partitioning by size and a representation of solid paths as globally biased binary trees, any dynamic tree operation takes $O(\log n)$ time.*

6. APPLICATIONS, RELATED WORK, AND REMARKS

In this section we shall give some applications of our algorithms for the dynamic trees problem, in the process discussing previous work related to ours. Our list of applications is meant to be illustrative, not exhaustive. We conclude the section with a few remarks.

Finding Roots

We obtain a primitive version of the dynamic trees problem if we allow only the operations of *root*, *link*, *cut*, and *evert*. For the even more restricted problem with only *root* and *link*, the "compressed tree" data structure for disjoint set union [19, 21] gives an $O(n + m\alpha(m + n, n))$-time algorithm, where α is a functional inverse of Ackermann's function. The version of the disjoint set union algorithm that uses union by rank or size [21] but not path compression can be adapted to handle *root*, *link*, and *cut* in $O(\log n)$ time per operation. Although this method is relatively simple, it does not seem to extend to allow *evert*, nor to maintain information about the paths in the trees.

No Cuts or Everts

Tarjan [20] has studied a version of the dynamic trees problem in which links are allowed but neither cuts nor everts, obtaining an $O(n + m\alpha(m + n, n))$-time algorithm with a number of applications. Tarjan uses the same idea we have used, of partitioning the tree edges into heavy and light.

Nearest Common Ancestors

Another useful operation on dynamic trees is the following:

nca(**vertex** v, w): Return the nearest common ancestor of v and w.

The problem of implementing this operation along with some combination of *link*, *cut*, and *evert* is the *nearest common ancestor problem* [1, 10, 13]. Our data structure easily adapts to this problem. To find the nearest common ancestor of v and w, we expose v and then expose w, during the second expose noting the first vertex encountered on the previously exposed path from v; this is the nearest common ancestor. (With the data structure of Section 5 we must finish the operation by concealing w and then v.) The previous result closest to this is by Aho *et al.* [1], who proposed a method that allows links but neither cuts nor everts and runs in $O((m + n) \log n)$ time and $O(n \log n)$ space. Maier [13] has given a more complicated method that reduces the space required and allows cuts, though the time bound for a cut is $O(n)$. Our method allows both cuts and everts, uses $O(n)$ space, and runs in $O(\log n)$ time per operation.

Network Flow

A classic problem in network optimization is to find a maximum flow in a network (a directed graph with nonnegative edge capacities) from a given source vertex s to a given sink vertex t. A sequence of faster and faster algorithms have been devised for this problem. For dense graphs, the best time bound known is $O(n^3)$, attained by the algorithms of Karzanov [11] and Malhotra *et al.* [14]. For sparse graphs, the best previously known time bound is $O(nm(\log n)^2)$, attained by the algorithm of Galil and Naamad [8], rediscovered by Shiloach [16]. Here n is the number of vertices and m is the number of edges in the problem graph.

Our data structure for dynamic trees gives an algorithm for this problem whose running time is $O(nm \log n)$. Our algorithm (as are all the fast network flow algorithms) is based on the fundamental work of Dinits [6]. Dinits reduced the maximum network flow problem to the solution of n blocking flow problems. The blocking flow problem is to find a flow in an acyclic network from a source s to a sink t such that every path from s to t contains a saturated edge (an edge whose flow equals its capacity). Dinits' algorithm for finding a blocking flow consists of beginning with the zero flow, discarding all zero-capacity edges, and repeating the following step until there is no path from s to t:

Augmenting Step. Find a path from s to t. Let c be the capacity of a minimum-capacity edge on the path. Send c units of flow through the path, reducing the (residual) capacity of each edge on the path by c. Delete all edges whose capacity is now zero.

Since each execution of the augmenting step saturates at least one edge, there are at most m executions of the step. The hard part of the method is to find augmenting paths and update edge capacities. The straightforward implementation of Dinits' algorithm takes $O(n)$ time per augmenting path plus $O(m)$ time amortized over all paths, for a total time of $O(nm)$. Galil and Naamad [8] and Shiloach [16] discovered an $O(m(\log n)^2)$-time method that uses almost the same structure as described in Sections 3 and 4, with balanced trees in place of biased trees representing solid paths. We have obtained our result by introducing biased trees and doing the required running time analysis.

Our data structure gives an $O(m \log n)$ time bound for the blocking flow problem. We use dynamic trees to find augmenting paths and keep track of edge capacities. The algorithm maintains a collection of trees consisting of at most one unsaturated edge leaving each vertex; these edges are the current candidates to form augmenting paths. Initially each vertex is in a separate tree. We execute Dinits' algorithm as follows:

Step 1. Let $v = root(s)$. If $v = t$, go to Step 4; otherwise, go to Step 2.

Step 2 ($v \neq t$; extend path). If no edges leave vertex v, go to Step 3. Otherwise, select an edge (v, w) leaving v and perform $link(v, w, capacity((v, w)))$. Go to Step 1.

Step 3 (all paths from v to t are blocked). If $v = s$, compute the unused capacity of every tree edge using *cost* and stop. Otherwise, delete from the graph every edge entering v. For each such edge (u, v) that is a tree edge, perform $cut(u)$, recording the unused capacity. Go to Step 1.

Step 4 ($v = t$; the tree path from s to t is augmenting). Let $v = mincost(s)$. (Edge $(v, parent(v))$ is a minimum-capacity edge on the augmenting path.) Let $c = cost(v)$. Perform $update(s, -c)$. Go to Step 5.

Step 5 (delete edges with no remaining capacity). Let $v = mincost(s)$. If $cost(v) = 0$, delete $(v, parent(v))$ from the graph, perform $cut(v)$, recording an unused capacity of zero, and repeat Step 5. Otherwise, ($cost(v) > 0$), go to Step 1.

When the algorithm stops, we can compute the final flow from the original and unused capacities. The algorithm requires $O(m)$ dynamic tree operations and thus takes $O(m \log n)$ time. Using this method for finding blocking flows we obtain an $O(nm \log n)$ time bound for the maximum flow problem.

Another flow problem amenable to this approach is the *acyclic flow problem*: given a flow from s to t in an arbitrary network, reduce it to an acyclic flow by repeatedly finding a cycle of flow and reducing the flow around the cycle to zero. An appropriate version of the blocking flow algorithm will solve this problem in $O(m \log n)$ time. We begin with each vertex in a separate tree, delete all zero-capacity edges, and carry out the following steps:

Step 1. Let $v = root(s)$. If no edges leave vertex v, go to Step 2. Otherwise, select an edge (v, w) leaving v. If $root(w) = v$, go to Step 3. Otherwise, perform $link(v, w, flow((v, w)))$ and repeat Step 1.

Step 2 (all paths from v to t are acyclic). If $v = s$, compute the current flow of every tree edge using *cost* and stop. Otherwise, delete from the graph every edge entering v. For each such edge (u, v) that is a tree edge, perform $cut(u)$, recording the current flow. Go to Step 1.

Step 3 (a cycle of positive flow has been found). Let $u = mincost(w)$. Let $c = \min\{flow((v, w)), cost(u)\}$. Reduce $flow((v, w))$ by c and perform $update(w, -c)$. Go to Step 4.

Step 4 (delete tree edges with no remaining flow). Let $u = mincost(w)$. If $cost(u) = 0$, delete $(u, parent(u))$ from the graph, perform $cut(u)$, recording a current flow of zero, and repeat Step 4. Otherwise $(cost(u) > 0)$, go to Step 1.

Constrained Minimum Spanning Trees

Gabow and Tarjan [7] consider the following problem: Given a connected undirected graph whose edges are of two colors, say white and black, and have real valued costs, find a spanning tree of minimum total edge cost containing exactly k black edges for some integer k. They describe an algorithm for this problem whose critical part consists of repeating the following step $n - 1$ times: given a spanning tree T and a non-tree edge $\{v, w\}$, find the maximum-cost edge, say $\{x, y\}$, on the (unique) tree path joining v and w; form a new tree T' by deleting (x, y) and adding (v, w). We can use our data structure to carry out this process in $O(\log n)$ time per swap for a total of $O(n \log n)$ time.

The Network Simplex Algorithm

An edge-swapping process almost identical to the one described above occurs in the network simplex algorithm for the transportation problem [4]. Operations researchers have proposed various data structures to represent the so-called "feasible tree solutions" in this problem, but they all take $O(n)$ time per swap in the worst case. With our data structure the time per swap is $O(\log n)$. Although theoretically the network simplex algorithm needs exponential time in the worst case, it is heavily used in practice, and our result offers hope of further improving its practical performance.

Remarks and Further Research

Much work remains to be done on the dynamic trees problem. One direction for future work is to implement our data structures and perform empirical experiments to determine their practical value, if any. Sleator has implemented a dynamic trees algorithm for maximum network flow. It is slower by a small constant factor (about two) than the algorithm of Dinits except on specially constructed worst-case example graphs. This merely reflects the fact that of randomly generated graphs only a small fraction make Dinits's algorithm perform poorly. Some experiments with our method applied to the network simplex algorithm might be rewarding.

The amortized-time version of our data structure uses locally biased binary trees, whereas the worst-case version uses globally biased trees. Globally biased trees will work in the amortized-time version but locally biased trees will not work in the worst-case version. Further work to improve and simplify the data structure may be valuable. Recently the authors have discovered a new kind of search tree, called a self-adjusting search tree [22], which can be substituted for biased binary trees in the data structure of Sections 3 and 4. This produces considerable simplification. (See Tarjan [23].)

REFERENCES

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, On finding lowest common ancestors in trees, *SIAM J. Comput.* **5** (1975), 115–132.
2. S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, Biased 2–3 trees, *in* "Proc. Twenty-First Annual IEEE Symp. on Foundations of Computer Science," pp. 248–254, 1980.
3. S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, "Biased Search Trees," *SIAM J. Comput.*, to appear.
4. C. V. CHVATAL, "Linear Programming," Freeman, San Francisco, 1983, to appear.
5. E. W. DIJKSTRA, "A Discipline of Programming," Prentice–Hall, Englewood Cliffs, N.J., 1976.
6. E. A. DINITS, An algorithm for the solution of a problem of maximal flow in a network with power estimation, *Soviet Math. Dokl.* **11** (1970), 1277–1280.
7. H. N. GABOW AND R. E. TARJAN, Efficient algorithms for a family of matroid problems, *J. Algorithms*, to appear.
8. Z. GALIL AND A. NAAMAD, An $O(EV\log^2 V)$ algorithm for the maximal flow problem, *J. Comput. System Sci.* **21** (1980), 203–217.
9. L. G. GUIBAS AND R. SEDGEWICK, A dichromatic framework for balanced trees, *in* "Proc. Nineteenth Annual IEEE Symp. on Foundations of Computer Science," pp. 8–21, 1978.
10. D. HAREL AND R. E. TARJAN, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, to appear.
11. A. V. KARZANOV, Determining the maximal flow in a network by the method of preflows, *Soviet Math. Dokl.* **15** (1974), 434–437.
12. D. E. KNUTH, "The Art of Computer Programming, Vol. 3: Sorting and Searching," Addison–Wesley, Reading, Mass., 1974.
13. D. MAIER, An efficient method for storing ancestor information in trees, *SIAM J. Comput.* **8** (1979), 599–618.

14. V. M. MALHOTRA, M. P. KUMAR, AND S. N. MAHESHWARI, An $O(|V|^3)$ algorithm for maximum flows in networks, *Inform. Proc. Lett.* 7 (1978), 277–278.
15. J. NIEVERGELT AND F. M. REINGOLD, Binary search trees of bounded balance, *SIAM J. Comput.* 2 (1973), 33–43.
16. Y. SHILOACH, "An $O(nI \log^2 I)$ Maximum-Flow Algorithm," Tech. Rep. STAN-CS-78-702, Computer Science Department, Stanford University, Stanford, Calif., 1978.
17. D. D. SLEATOR, "An $O(nm \log n)$ Algorithm for Maximum Network Flow," Tech. Rep. STAN-CS-80-831, Computer Science Department, Stanford University, Stanford, Calif., 1980.
18. D. D. SLEATOR AND R. E. TARJAN, A data structure for dynamic trees, *in* "Proc. Thirteenth Annual ACM Symp. on Theory of Computing," pp. 114–122, 1981.
19. R. E. TARJAN, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* 22 (1975), 215–225.
20. R. E. TARJAN, Applications of path compression on balanced trees, *J. Assoc. Comput. Mach.* 26 (1979), 690–715.
21. R. E. TARJAN AND J. VAN LEEUWEN, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, to appear.
22. D. D. SLEATOR AND R. E. TARJAN, Self-adjusting binary trees, *in* "Proc. Fifteenth Annual ACM Symp. on Theory of Computing," pp. 235–245, 1983.
23. R. E. TARJAN, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, Penn., to appear.

Printed by the St. Catherine Press Ltd., Tempelhof 41, Bruges, Belgium