1992

A Data Structure for Dynamically Maintaining Rooted Trees

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Report Number:
92-066

A DATA STRUCTURE FOR DYNAMICALLY MAINTAINING ROOTED TREES

Greg N. Frederickson

CSD-TR-92-066
July 17, 1992

A data structure for dynamically maintaining rooted trees

Greg N. Frederickson*

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

gnf@cs.purdue.edu

July 17, 1992

Abstract. The directed topology tree data structure is developed for maintaining binary trees dynamically. Each of a certain set of tree operations is shown to take $O(\log n)$ time, where n is the number of vertices in the trees. The directed topology trees are used to implement link-cut trees and dynamic expression trees. The results of experimental comparisons with the dynamic trees of Sleator and Tarjan are presented.

Key words and phrases. Analysis of algorithms, data structures, dynamic expression tree, dynamic tree, experimental evaluation, link-cut tree, maximum flow, splay tree, topology tree.

*This research was supported in part by the National Science Foundation under grant CCR-9001241 and by the Office of Naval Research under contract N00014-86-K-0689.

1 Introduction

A fundamental problem in data structures is to maintain and query a structure that represents a dynamically-changing tree. Several approaches have been given in the last decade, including dynamic trees [ST1], splay trees [ST2], topology trees [F1], and restricted topology trees [F2], [F3]. The dynamic trees and the splay trees have found application in algorithms for finding a maximum flow [ST1], [ST2], [GT], in data structures for updating minimum spanning trees of plane graphs [EITTWY], in data structures for maintaining dynamic expression trees [CT], and in finding a separator decomposition of a planar graph [G]. The topology trees and restricted topology trees have found application in data structures for updating minimum spanning trees and maintaining connectivity information [F1], in data structures for updating 2-edge-connectivity information [F2], [F3] and in an algorithm to find the k smallest spanning trees of a graph [F2], [F3].

The above approaches may all be loosely characterized as maintaining a tree as a set of paths, with the main differences being how the representations of the paths are combined together. In this paper we show how to adapt topology trees to handle the applications for which dynamic trees are useful. In particular, we present a version of restricted topology trees, which we call *directed topology trees*, which can be used to represent binary trees. Each operation in a directed topology tree uses $O(\log n)$ time, where n is the number of nodes in the trees. We give an implementation of link-cut trees [ST1] in terms of directed topology trees, and show how these can be used in the Goldberg-Tarjan maximum flow algorithm. (We use the term *link-cut tree* to refer to a generic data structure that admits *link*, *cut* and related operations, and refer specifically to the implementations of link-cut trees in [ST1] as dynamic trees.) We also give an implementation of dynamic expression trees in terms of directed topology

trees. Our implementation of dynamic expression trees appears to be conceptually simpler than that in [CT]. Finally, we give the results of computational experiments designed to test the competitiveness of directed topology trees against splay trees as used in the Goldberg-Tarjan maximum flow algorithm.

While we do not improve on the worst-case operation times, we do present an alternative data structure for these problems. One advantage of directed topology trees appears to be that the paths that collectively constitute a tree are more cleanly organized into a single unified data structure. It is not necessary to represent each path by a complicated structure such as a biased search tree [BST], or by a self-adjusting structure with less intuitive rebalancing rules and delicate amortization arguments. As claimed previously, directed topology trees seem to lead to simpler representations of dynamic expression trees than dynamic trees. As a disadvantage, we note that the directed topology trees appear to use more space (by a constant factor) than splay trees. It is not clear whether directed topology trees use more space on average than dynamic trees. Another potential disadvantage is that directed topology trees represent binary trees, and trees with greater degree must be adapted by specific techniques. We note that this does not appear to incur an unreasonable penalty, as the results in our experiments with regard to the maximum flow problem show.

Our paper is organized as follows. In section 2 we review the topology tree data structure. In section 3 we review link-cut trees. In section 4 we describe our directed topology tree data structure. In section 5 we describe our implementation of link-cut trees in terms of directed topology trees, and discuss modifications necessary to use them in the Goldberg-Tarjan algorithm for finding a maximum flow. In section 6 we describe our implementation of dynamic expression trees. In section 7 we present the result of experiments to compare the performance of the dynamic trees of Sleator and

Tarjan with our directed topology trees.

2 Review of topology trees

In this section we review basic data structures from [F2], [F3]. We recall from [F2], [F3] the definitions of a vertex cluster, a restricted partition, and a restricted multi-level partition. Following [F2], [F3], we define a "topology tree" based on the restricted multi-level partition.

Let T be an undirected tree defined on the vertex set V with maximum vertex degree at most 3. A *vertex cluster* with respect to T is a set of vertices such that the subgraph of T induced on the cluster is connected. The *degree* of a vertex cluster is the number of edges with precisely one endpoint in the cluster. Two vertex clusters are *adjacent* if there is an edge with one endpoint in each of the clusters.

We define a partition of a set of vertices so that the resulting vertex clusters possess certain nice properties. A *restricted partition* of a tree T of maximum degree 3 is a partition of V such that

1. Each cluster of degree 3 is of cardinality 1.
2. Each cluster of degree less than 3 is of cardinality at most 2.
3. No two adjacent clusters can be combined and still satisfy the above.

Note that each set in the partition will be a vertex cluster of degree at most 3.

We next define our restricted multi-level partition. A *restricted multi-level partition* is a set of partitions of V that satisfy the following:

1. For each level $l = 0, 1, \cdots, q$, the vertex clusters at level l form a partition of V.
2. The clusters at level 0 each contain one vertex.
3. The clusters at any level $l > 0$ constitute a restricted partition with respect to the tree resulting by viewing each cluster at level $l-1$ as a vertex.
4. There is precisely one vertex cluster at level q, which contains all vertices.

A vertex cluster at level 0 of a restricted multi-level partition is called a *basic vertex cluster.* Since any basic vertex cluster contains just one vertex, and any cluster resulting from the union of two clusters will have degree at most 2, any cluster of degree 3 will consist of a single vertex.

From [F2], [F3] we have the following result.

Theorem 2.1 *[F2], [F3] The number of levels in a restricted multi-level partition is* $\Theta(\log n)$. □

We next recall a structure from [F2], [F3]. A *topology tree* for a tree T is a tree in which each nonleaf node has at most two children, and all leaves are at the same depth, such that:

1. A node at level l in the topology tree represents a vertex cluster at level l in the restricted multi-level partition.
2. A node at level $l > 0$ has children that represent the vertex clusters at level $l-1$ whose union is the vertex cluster it represents.

When an edge is removed from T, leaving two trees, T_1 and T_2, we call the operation of splitting the topology tree for T into topology trees for T_1 and T_2 an *edge deletion.* When two vertex disjoint trees T_1 and T_2 are combined into one tree T by adding an edge with one endpoint in each of the trees, we call the operation of combining the topology trees for T_1 and T_2 into a topology tree for T an *edge insertion.* Once the adjacency of the endpoints of the inserted or deleted edge is adjusted, the topology tree or trees are adjusted by a sweep upwards, updating all affected nodes. These operations are described in detail in [F3]. From [F2], [F3] we have the following result.

Lemma 2.2 *[F2], [F3] The time to perform an edge insertion or edge deletion with respect to topology trees is* $O(\log n)$. □

3 Review of link-cut trees

In this section, we review link-cut trees, which were introduced by Sleator and Tarjan in [ST1]. As stated in the introduction, we take the term *link-cut tree* to refer to a generic data structure that admits *link*, *cut* and related operations. Implementations of link-cut trees have been given in [ST1] and [ST2]. Slight variations in the specifications of link-cut trees occur in [ST1] and [ST2]. We consider the following specification, found in [ST1]. Link-cut trees maintain a forest of vertex-disjoint rooted trees, each of whose edges has a real-valued *cost*, under a sequence of eight operations.

parent(**vertex** v): If v is not a tree root, then return the parent of v, else return **null**.

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

cost(**vertex** v): If v is not a tree root, then return the cost of the edge from v to its parent.

mincost(**vertex** v): If v is not a tree root, then return the vertex w closest to $root(v)$ such that the edge from w to $parent(w)$ is of minimum cost among edges on the path from v to $root(v)$.

update(**vertex** v, **real** x): Add x to the cost of each edge on the path from v to $root(v)$.

link(**vertex** v, w, **real** x): If v is a tree root and w is in a different tree, add an edge of cost x that combines the two trees, making v a child of w.

cut(**vertex** v): If v is not a tree root, remove the edge from v to its parent, thus dividing the tree into two trees.

evert(**vertex** v): Modify the tree containing v by making v the root.

As noted in [ST1], *evert* can generally be dropped in applications involving rooted trees. We shall not supply an implementation of *evert* for directed topology trees, although it is possible to give an efficient implementation.

Also, costs could be associated with vertices rather than edges. (Indeed, for every vertex other than the root, the cost of the edge to the parent can be viewed as the cost of that vertex.) In [ST2], costs are explicitly associated with vertices, and all operations except *parent* and *evert* are discussed (though sometimes with different names).

An amortized time of $O(\log n)$ per operation has been achieved using dynamic trees [ST1] or splay trees [ST2]. A worst-case time of $O(\log n)$ per operation has been achieved using biased search trees [BST] together with dynamic trees [ST1].

4 Directed topology trees

We introduce our new data structure, directed topology trees in this section. We show that *link* and *cut* operations can be performed in $O(\log n)$ time. Finally we present an example and give some terminology.

Consider a binary (hence rooted) tree. We modify the approach in section 2 to handle such trees. We first change the definition of a restricted partition, to simplify the various tests relating to clustering or reclustering. A *restricted partition of a binary tree* T is a partition of V into clusters such that

1. Each cluster has cardinality at most 2.
2. Each cluster that has two children (in the induced tree) has cardinality 1.
3. No two adjacent clusters can be combined and still satisfy the above.

The only difference with the rules for an undirected tree occurs at the root: We do not allow a root with just one child to be clustered with that child if that child has 2 children. (In such a case, the root would have tree degree one if the tree were undirected.)

Using this revised partition, we define a *multi-level partition* and a *directed topology tree* in a fashion similar to that in section 2. We note that the restricted multi-level

partition is somewhat similar to a structure that may be inferred from applying the rake-and-compress paradigm to a rooted binary tree [MR], [CV], [ADKP].

Theorem 4.1 *A directed topology tree representing a binary tree of n nodes is of height $O(\log n)$. Operations link and cut can be performed in $O(\log n)$ time.*

Proof. Using an analysis similar to that in [F2], [F3], the number of levels in the resulting multi-level partition is still $O(\log n)$. The *link* and *cut* operations are analogous to edge insertion and edge deletion operations, and can be performed in $O(\log n)$ time. □

Consider the binary tree with the edge costs and vertex names shown in Figure 4.1. Also shown in the figure is the multi-level partition of this tree. The corresponding topology tree is shown in Figure 4.2. Each leaf, labeled with a small letter, represents a vertex in the binary tree, and each interior node, labeled by a capital letter, represents a nonbasic vertex cluster of the tree.

The rooted tree defined at any level of the partition will be called an *induced tree.* The children of a node in an induced tree will be designated as *leftchild* and *rightchild,* and the parent in the induced tree will be called *indparent.* When two nodes in an induced tree are clustered, one node u will be the parent of the other node v. Node u will be the *topchild* of the cluster, and node v will be the *botchild* of the cluster in the topology tree. The cluster will be the *topoparent* of u and v. If a node u is clustered by itself, then node u will be the *topchild* of the cluster, and *botchild* will be null.

5 Implementation of link-cut trees

In this section we use directed topology trees to implement link-cut trees. We first define the notion of a restricted path, and then describe how we represent the costs of edges in the link-cut trees. We present the algorithms for the various operations

on link-cut trees, and show that each takes $O(\log n)$ time. Finally, we discuss how to adapt the Goldberg-Tarjan maximum flow algorithm so that topology trees can be used.

To help in understanding how we represent edge costs, we define a *restricted path* for any node w that is not a root in its induced tree. If w is a leaf in the topology tree, then it represents a single vertex v, and its restricted path contains only the edge from v to its parent in the induced tree. If w has two children in the induced tree, then it represents a single vertex v, and its restricted path contains only the edge from v to its parent in the induced tree. Otherwise, w has precisely one child in the induced tree. In this case, if it has one child $topchild(w)$ in the topology tree or two children in the induced tree, then its restricted path is the restricted path for $topchild(w)$; otherwise its restricted path is the concatenation of the restricted path for $botchild(w)$ followed by the restricted path for $topchild(w)$.

For example, consider the tree in Figure 4.1. The restricted path for d is (d, e), and for e is (e, g). The restricted path for node C is (e, g) and for node J is $(e, g), (g, h)$. The restricted path for N is (a, c), (c, e), (e, g), (g, h) and the restricted path for node Q is (a, c), (c, e), (e, g), (g, h), (h, l). Nodes l, H, M, P, R and S do not have restricted paths, since they are the roots of induced trees. The binary tree can be expressed as the union of certain disjoint paths, which we call *maximal* restricted paths. In this case it is the union of the restricted paths for b, d, f, j, L and Q. These maximal restricted paths are shown in dashed lines in Figure 5.1. In this regard, topology trees have a basis similar to dynamic trees.

For each node w in the topology tree, we will have three additional fields, $\Delta cost$, $nodemin$ and $minvert$. We let $\Delta cost(w)$ be the incremental cost associated with w. The actual cost of an edge from a vertex v to its parent will equal the sum of $\Delta cost(w)$ for all ancestors w of v in the topology tree such that v is on the restricted path for w.

We let $\Delta cost$ of the root of any induced tree be infinity. Let $minvert(w)$ be the bottom endpoint of a minimum cost edge on the restricted path for w. Among all possible choices, choose the highest such edge on the restricted path. Then $nodemin(w)$ is the sum of $\Delta cost$ for all ancestors (in the topology tree) of $minvert(w)$ up to and including w. Phrasing $nodemin(w)$ in terms of the children of w in the topology tree gives the following. If w is a leaf in the topology tree, then $nodemin(w)$ is $\Delta cost(w)$. If w has two children in the induced tree, or w has precisely one child in the induced tree and one child in the topology tree, then $nodemin(w)$ is $\Delta cost(w)+nodemin(topchild(w))$. If w has precisely one child in the induced tree and two children in the topology tree, then $nodemin(w)$ is $\Delta cost(w)+\min\{nodemin(topchild(w)), nodemin(botchild(w))\}$.

We consider the example in Figures 4.1 and 4.2. Node f in the topology tree has just one ancestor, itself, whose restricted path contains f. Thus the actual cost of edge $(f, indparent(f))$ is just $\Delta cost(f) = 9$. Node g in the topology tree has ancestors g, D, J, N, and Q whose restricted paths contain g. Thus the actual cost of edge $(g, indparent(g))$ is $3 + 0 + (-1) + 2 + 0 = 4$.

We show how to implement $update(v, x)$ and a slight generalization of $mincost(v)$. We take advantage of the fact that every edge is the edge from some nonroot vertex to its parent. To find the minimum on the path from v to the root, we do the following. We search up in the topology tree from the leaf representing v. We keep track of the smallest cost edge found so far on the portions of restricted paths that we have already explored and are not contained in the current restricted path. We also keep track of the smallest cost of any edge on the portion of the current restricted path that we have explored so far. At each level we add in $\Delta cost(w)$, and test if w, the ancestor of v at that level, has a sibling w_sib in the topology tree that is its parent in the induced tree. If so, we need to update the appropriate smallest costs so far, considering which of the clustering rules was used on w and w_sib. When we reach

the root of the topology tree, we have found the lowest cost edge on the path between v and the root.

We now give our procedure to find the minimum cost *curr_min* of an edge on the path from vertex v to the root, and the vertex *curr_vert* closest to the root such that the edge from this vertex to its parent achieves this minimum cost.

```
proc mincostvert(v)
    curr_min ← ∞        /* cost of min cost edge on path from v to root */
    curr_vert ← nil     /* lower endpoint of corresponding edge */
    temp_min ← 0        /* smallest cost of edge to which Δcost can be added */
    temp_vert ← v       /* lower endpoint of corresponding edge */
    w ← v
    while w is not the root of the topology tree do
        temp_min ← temp_min + Δcost(w)
        if w has a sibling w_sib in the topology tree
            and w_sib is the parent of w in the induced tree
        then
            if w_sib has precisely one child and is not the root
            then
                choosemin(temp_min, temp_vert, nodemin(w_sib), minvert(w_sib))
            else
                choosemin(curr_min, curr_vert, temp_min, temp_vert)
                temp_min ← nodemin(w_sib)
                temp_vert ← minvert(w_sib)
            endif
        endif
        w ← parent of w in the topology tree
    endwhile
```

Note that if *w_sib* is the root of some induced tree, then *temp_min* is reset to ∞, and all succeeding values of *temp_min* will be ∞, since ∞ plus some constant is ∞. Procedure *findmin* uses the following macro, which updates a minimum value, and the edge that realizes that minimum.

```
macro choosemin(some_min, some_vert, trial_value, trial_vert)
    if some_min ≥ trial_value
    then
```

```
        some_min ← trial_value
        some_vert ← trial_vert
    endif
```

The procedure to find $cost(v)$ is similar in structure, but simpler than, *mincostvert*.

To add a value x to every edge on the path from v to the root, we do the following. We search up in the topology tree from the leaf representing v. At each level we adjust the *nodemin* value. If the current restricted path is extended upwards, or the new restricted path is a subpath of a different restricted path, we add x to the $\Delta cost$ and *nodemin* fields.

We now give our procedure to add the value x to every edge on the path from vertex v to the root.

```
proc update(v, x)
    Δcost(v) ← Δcost(v) + x
    w ← v
    while w is not the root of the topology tree do
        Recompute nodemin(w) and minvert(w).
        if w has a sibling w_sib in the topology tree
            and w_sib is the parent of w in the induced tree
        then
            Δcost(w_sib) ← Δcost(w_sib) + x
            nodemin(w_sib) ← nodemin(w_sib) + x
        endif
        w ← parent of w in the topology tree
    endwhile
```

Note that at some point procedure *update* adds x to $\Delta cost$ of the root of some induced tree. This does not matter, since ∞ plus some constant will be ∞.

When we perform $link(v, w, x)$, we shall set $\Delta cost(v)$ to x. Note that when the corresponding two topology trees are merged, any nonleaf node u that has a nonzero $\Delta cost(u)$ and that gets handled in the merging, must have had its $\Delta cost$ value distributed to its children in the topology tree. We call this operation a *cleaning*. As

soon as it becomes apparent that a node will be handled, it and all of its uncleaned ancestors must be cleaned. Similarly, when a $cut(v)$ is performed, any nonleaf node that will be handled should have its $\Delta cost$ distributed downward. This cleaning will take $O(\log n)$ time overall, since ancestors that have been cleaned can be marked, so that they are not recleaned if they are ancestors of another node that through a change in its cluster must be cleaned.

Theorem 5.1 *Directed topology trees allow mincostvert, cost, update and root operations to be performed in $O(\log n)$ time, and parent in constant time.*

Proof. To find a parent in the original tree, just follow the *indparent* pointer. To find the root in the original tree, follow *topoparent* pointers to the root of the topology tree, and then follow *topchild* pointers down to a leaf in the topology tree. This will clearly take $O(\log n)$ time. The procedure *cost* just moves up the topology tree until it reaches the end of the current restricted path, adding in the $\Delta cost$ of the current node as it goes. This clearly takes constant time per level. Procedures *mincostvert* and *update* also take constant time per level. □

We next discuss how to adapt the maximum flow algorithm of Goldberg and Tarjan [GT] so that our directed topology trees can be used. We note that applying the graph transformation of [H] in the obvious fashion would increase the running time considerably. The transformation would increase the number of vertices to $\Theta(m)$, and thus the running time of the Goldberg-Tarjan algorithm to $O(m^2 \log m)$.

Instead, we do the following. When a $link(v, u, x)$ operation would add a third child to vertex u, we use a new operation *insert_link*. The operation first creates a new vertex u'. Then it performs a $cut(w)$, where w is $leftchild(u)$, followed by three links: $link(w, u', x')$, $link(v, u', x)$ and $link(u', u, \infty)$. The value of x' is the value associated with edge (w, u) once the ancestors of w have been cleaned. Rather than

reorganize the topology trees after each of the individual operations, the topology tree is reorganized in one upwards pass after the changes at the lowest level are made. Any new edge introduced from a new vertex u' to its parent u is treated as being of length 0, so that the valid labeling in the Goldberg-Tarjan algorithm is extended to have $d(u') = d(u)$. In addition we do not count u' in our count of vertices in a given link-cut tree. Whenever we perform a link, we charge the new vertex u' to the vertex v that is no longer a root. Thus the actual number of nodes in any of our link-cut trees is no greater than $2k$.

To perform a $cut(v)$ operation, we cut the edges into any new node v', and delete all such nodes. We also cut v from its parent. Note that the rebuilding of topology tree fragments should proceed only as all relevant edges have been cut from the fragment. Thus we do work proportional to $d \log k$, where d is the number of edges actually cut. Charging this cost to the d *link* operations that initially set up this structure show that the running time of the algorithm is not degraded.

Theorem 5.2 *The Goldberg-Tarjan algorithm for finding a maximum flow can be adapted to use directed topology trees, and still run in $O(mn \log(n^2/m))$ time, where m is the number of edges and n the number of vertices in the network.*

Proof. The proof follows from the above discussion. □

6 Implementation of dynamic expression trees

In this section we discuss dynamic expression trees based on topology trees. We first define an expression tree and the operations that act on it. We then specify what information to associate with each node in the topology tree. We then describe how the operations are implemented, and show that the time for each is $O(\log n)$.

We first define an expression tree. Let $(S, +, *, 0, 1)$ be a semiring. (S is a set of

elements closed under + and $*$, + and $*$ are associative, 0 is the identity for +, 1 is the identity for $*$, + is commutative, and $*$ distributes over +.) An *expression tree over* S is either a binary tree consisting of a single vertex with a value taken from S, or it is a tree consisting of a root with a label that is either + or $*$ and a left subtree and a right subtree that are both expression trees.

Consider a collection of expression trees for expressions over S. We handle the following operations.

makeleaftree(**value** x): Return an expression tree consisting of a single vertex with value x.

destroyleaftree(**vertex** v): If v is the root of an expression tree consisting of just a leaf, delete this tree.

construct(**operator** $\odot$,**vertex** u, v): If u and v are roots of expression trees, return an expression tree in which the root is labeled with $\odot$ and the left child of the root is u and the right child is v,.

destruct(**vertex** v): If v is the root of an expression tree not consisting of just a leaf, delete the root and return as two expression trees the subtree rooted at the left child of the root and the subtree rooted at the right child of the root.

evaluate(**vertex** v): Return the value of the subexpression associated with the subtree rooted at vertex v.

change(**vertex** v, **value** x): If v is a leaf in an expression tree, then reset the value of this leaf to be x.

swap(**vertex** u, v): If u is the root of an expression tree T_1, and v is a nonroot vertex in another expression tree T_2, then T_1 and T_2 are replaced by expression trees T_1' and T_2', where T_1' is the subtree of T_2 rooted at v , and T_2' is T_2 with the subtree rooted at v replaced by T_1.

This is the same set of operations as presented in [CT], except that their operations *graft* and *prune*, which are not quite symmetrical, are replaced by our single operation *swap*. It is not hard to verify that *swap* can be used to simulate *graft* and *prune*.

Let T be an expression tree for an expression E over S. We show how to store information about E in the nodes of the topology tree for T. We note that $*$ is not assumed to be commutative.

Let each instance of an operator or value in T label a leaf in the topology tree. There will be three types of nonleaf nodes in the topology tree. If a node v represents a cluster that has no children in the induced tree at a given level, then that node must be labeled with a value. If a node v represents a cluster that has 2 children in the induced tree at a given level, then that node must be labeled with an operator. Otherwise, the node represents a cluster that has one child in the induced tree at a given level. In this case the node is labeled by a linear form $A * X * B + C$, where A, B and C are constants and X represents the value of the child of the cluster in the induced tree, which is at that point unknown. We represent this form by a triple (A, B, C). We use the triple notation to represent single values too, representing a single value C by the form $(0, 0, C)$.

In clustering a node v with a child u, generate the label of the cluster as follows. If v is labeled with an operator, then u must be labeled by some triple $(0, 0, C)$. If the operator labeling v is a $+$, then label the parent cluster with $(1, 1, C)$, and otherwise label the parent cluster with $(1, C, 0)$. If v is not labeled with an operator, then it is labeled with a triple (A, B, C) and u is labeled with a triple (D, E, F). The parent cluster should be labeled with (G, H, J), where $G = A * D$, $H = E * B$, and $J = A * F * B + C$. If a node v is clustered by itself, then the label of its parent cluster will remain the same.

In the case in which the ring is commutative, the form is somewhat simpler: $A * X + B$. In that case triples are replaced by pairs. If v is labeled with a $+$, then label the parent cluster with $(1, C)$. If v is labeled with a $*$, then label the parent cluster with $(C, 0)$. If v is labeled with a pair (A, B), then u must be labeled with a

pair (C, D). The parent cluster should be labeled with (E, F), where $E = A * C$ and $F = A * D + B$.

We now discuss how to perform each operation on the directed topology tree. Operations *makeleaftree* and *destroyleaftree* can each be performed in a straightforward fashion. Operation *construct* is implemented by creating a new node and setting the adjacency between it and the nodes that will be its left and right children, and then rebuilding the topology tree bottom-up in a fashion similar performing a *link* operation. Operation *destruct* is implemented by reclaiming the root of the expression tree and setting the adjacency between it and its children, and then rebuilding the topology tree bottom-up in a fashion similar performing a *cut* operation.

Operation $evaluate(v)$ could be implemented by performing a *swap* with an arbitrary leaf tree, reading off the value at the root, and then swapping back again, but here is a faster method. If v is a leaf in the induced tree (i.e., in the expression tree itself), return its value. Otherwise, start with the leaf v in the topology tree, and note the operator $\odot$ labeling it. Then proceed upward through the topology tree, looking for the lowest two ancestors w' and w'' of v such that each of w' and w'' has two children in the topology tree, $topchild(w')$ is on the path from w' to v in the topology tree, and similarly for w''. Let c' be the value represented at w' and c'' the value represented at w''. If $botchild(w'')$ is the right child of $topchild(w'')$ in the induced tree, then return the value $c' \odot c''$, else return $c'' \odot c'$. The operation $change(v, x)$ requires that the path in the topology tree from the leaf representing vertex v to the root be traversed, and the triple (or pair) at each node visited be recomputed. The $swap(u, v)$ operation is implemented by setting the adjacency between nodes u, v and $indparent(v)$, and then rebuilding the topology tree bottom-up in a fashion similar performing a *link* or *cut* operation.

Theorem 6.1 *Using directed topology trees, each of the expression tree operations*

can be performed in $O(\log n)$ time, where n is the size of the corresponding expression trees.

Proof. The operations *makeleaftree* and *destroyleaftree* each take just constant time. The remaining operations take time proportional to the time to traverse a path in the topology tree and/or the time to perform an operation similar to a *link* or *cut*. Thus the total time for any operation is $O(\log n)$. □

In Figure 6.1, an expression tree is given for an expression over a commutative semiring, along with a multi-level partition. A topology tree for the expression tree is given in Figure 6.2. Whenever a node v has two children u and w in the topology tree, where u is the parent of w in the induced spanning tree, u is shown as the right child of v. Instead of labeling nodes with the pair notation, for clarity we omit the multiplicative factors of 1 and additive factors of 0. Also, we use a question mark to hold the place of an unknown. Thus $(9, 0)$, which represents "$9 * X + 0$" is shown as "$9 * ?$", and $(1, 8)$, which represents "$1 * X + 8$" is shown as "$? + 8$".

To *evaluate* the subexpression $8 + 6 * 9$, we are passed a pointer to the leaf in the topology tree that represents the $+$ sign of this subexpression. Searching up the topology tree, we find w' to be the node labeled "$? + 8$", whose *botchild* is labeled with value 8, and w'' to be the node labeled "186", whose *botchild* is labeled with value 54. Since *botchild*(w'') is the right child of *topchild*(w'') (See Figure 6.1.), the value returned should be $8 + 54$. (Of course the order of operands doesn't matter in this case, since the operators are both commutative.)

7 Experimental results

In this section we discuss some experimental comparisons between directed topology trees and splay trees, when both are used to implement link-cut trees. To generate reasonable sequences of instructions, we have considered the sequence of operations

generated as a result of running the Goldberg-Tarjan maximum flow algorithm [GT]. The goal of these tests has not been to identify the relative performance of the data structures with high precision. Rather it is to see if one data structure clearly dominates the other. No extensive search was made to determine if certain types of networks are more amenable to one approach or the other. No such trends have been suggested by the tests that we have made.

The motivation for testing both data structures within the framework of the Goldberg-Tarjan algorithm is that it provides a more "real-to-life" sequence of instructions than a more ad hoc generation method. It is well-known that splay trees do better when there is a substantial locality of reference, so it was felt to be important to have the sequence of operations generated by an application identified by the original authors. Of course, we understand that other maximum-flow algorithms may be faster in practice than the Goldberg-Tarjan algorithm.

The splay trees, the topology trees, and the Goldberg-Tarjan algorithm were coded by an undergraduate. The version of splaying employed is the standard 3-pass version. The code was written in C, using the macro facility liberally. Some care was taken to ensure that approximately the same number of function calls occurred in competing executions. The -O option was used for the optimization of the C code. From a viewpoint of fairness, a serious review of the code was made to ensure that the "deck hadn't been stacked" in favor of topology trees.

The code has been tested on graphs generated by the network generator NETGEN [KNS] as corrected by [BCJL]. The instances of the maxflow problem generated by NETGEN each have 1000 nodes. For each value of average node degree $20d$, for $d = 1, 2, \cdots, 9$, two instances were generated. Consequently, the parameter $k = n^2/m$, the bound on the size of individual trees in the Goldberg-Tarjan algorithm, ranged from 11 to 100.

The tests were run on a SPARC station IPC. (This is a 16 megabyte machine rated at 16.6 mips.) The execution times of various functions in the code were tabulated using the unix function *prof.* It is understood that interrupts may affect the exact timings during a run; however each example was run twice, with variation between runs being at most about 3%.

Timings are given for a collection of graphs in Table 7.1. Given in seconds are both the total execution time and time spent in performing link-cut tree operations. The time for performing the link-cut tree operations was larger for topology trees by a factor that ranged from around 1.40 for graphs with average degree 10 to around 1.56 for graphs with average degree 90. The ratio for tree operations seems to be worse for smaller size trees, reflecting perhaps the fact that the ratio $(\log(2k))/(\log k)$ is larger for small k. In any event, the topology trees do well considering their greater size. The total time was larger for topology trees by a factor that ranged from around 1.26 for graphs with average degree 10 to around 1.04 for graphs with average degree 90. The ratio for total time should decrease as tree size decreases, since there is more work between trees.

Acknowledgement.

I would like to acknowledge the programming assistance of Sean Vyain and Sean Ahern.

References

[ADKP] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 14:287–302, 1989.

[BST] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. on Computing*, 14:545–568, 1985.

[BCJL] R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladanyi, 1992. personal communication.

[CT] R. F. Cohen and R. Tamassia. Dynamic trees and their applications. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 52–61, 1991.

[CV] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[EITTWY] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. of Algorithms*, 13:33–54, 1992.

[F1] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. on Computing*, 14:781–798, 1985.

[F2] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, pages 632–641, 1991.

[F3] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees. CSD-TR-91-048, Purdue University, Department of Computer Science, revised February 1992.

[GT] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940, 1988.

[G] M. T. Goodrich. Planar separators and parallel polygon triangulation. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 507–516, 1992.

[H] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Massachusetts, 1969.

[JM] D. S. Johnson and C. C. McGeoch. Dimacs implementation challenge workshop algorithms for network flows and matching. Technical Report 92–4, DIMACS Center for Discrete Mathematics and Theoretical Computer Science, 1992.

[KNS] D. Klingman, A. Napier, and J. Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow problems. *Management Sci.*, 20:814–821, 1974.

[MR] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. *J. Comput. System Sci.*, to appear.

[ST1] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[ST2] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, 1985.

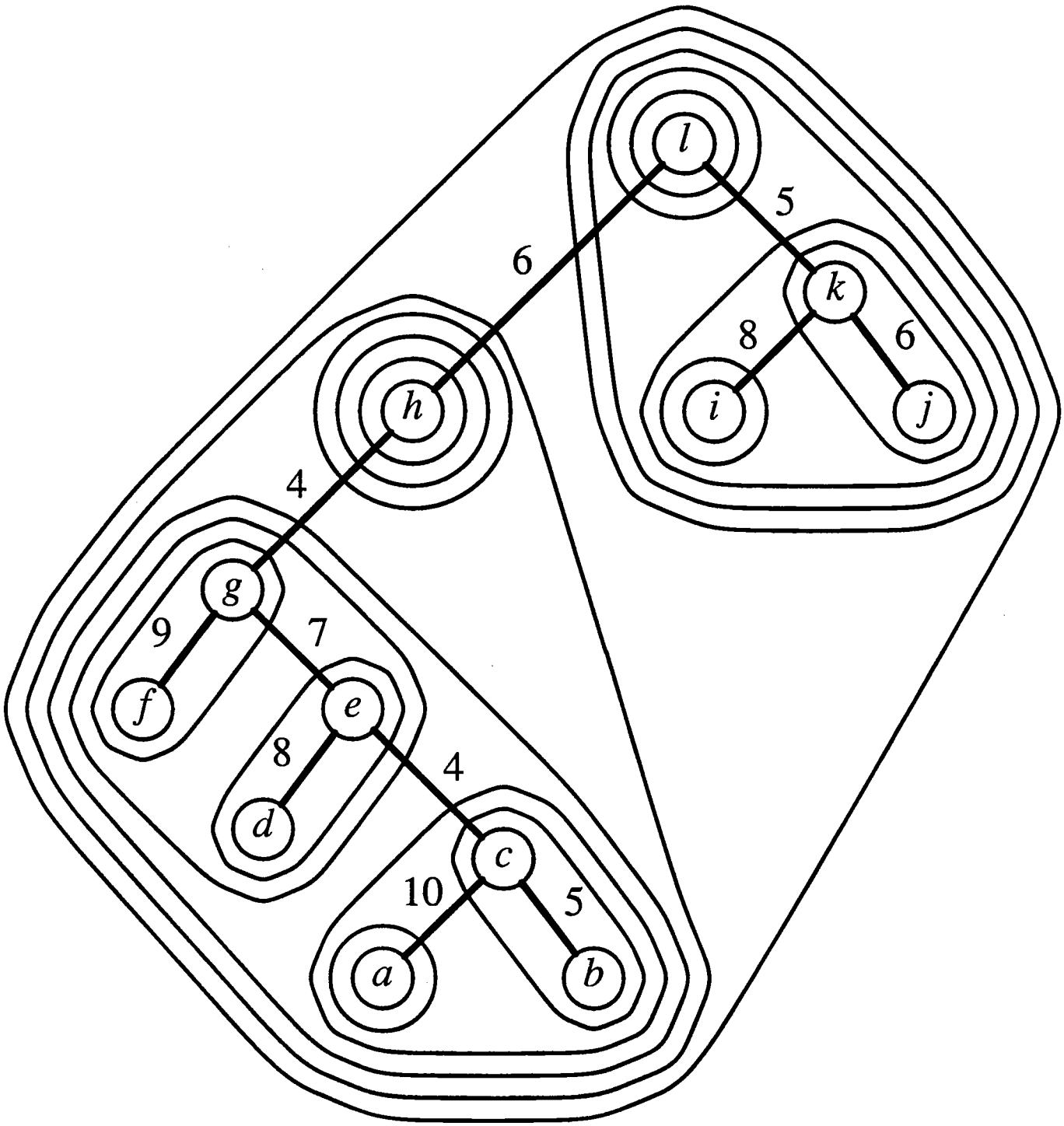

Figure 4.1. Binary tree with edge costs, vertex names, and multi-level partition.

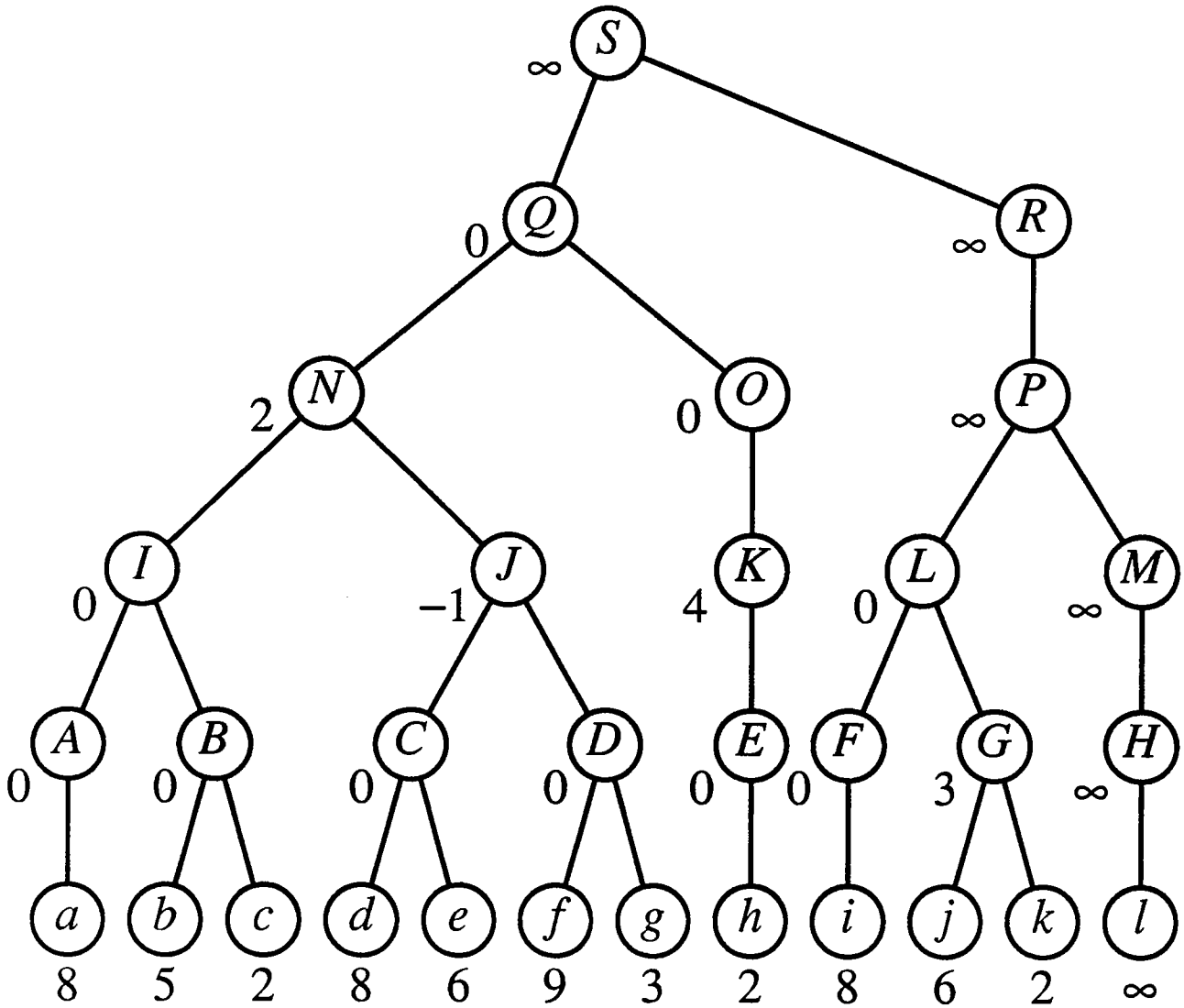

Figure 4.2. Topology tree with Δ*costs* for the binary tree in Figure 4.1.

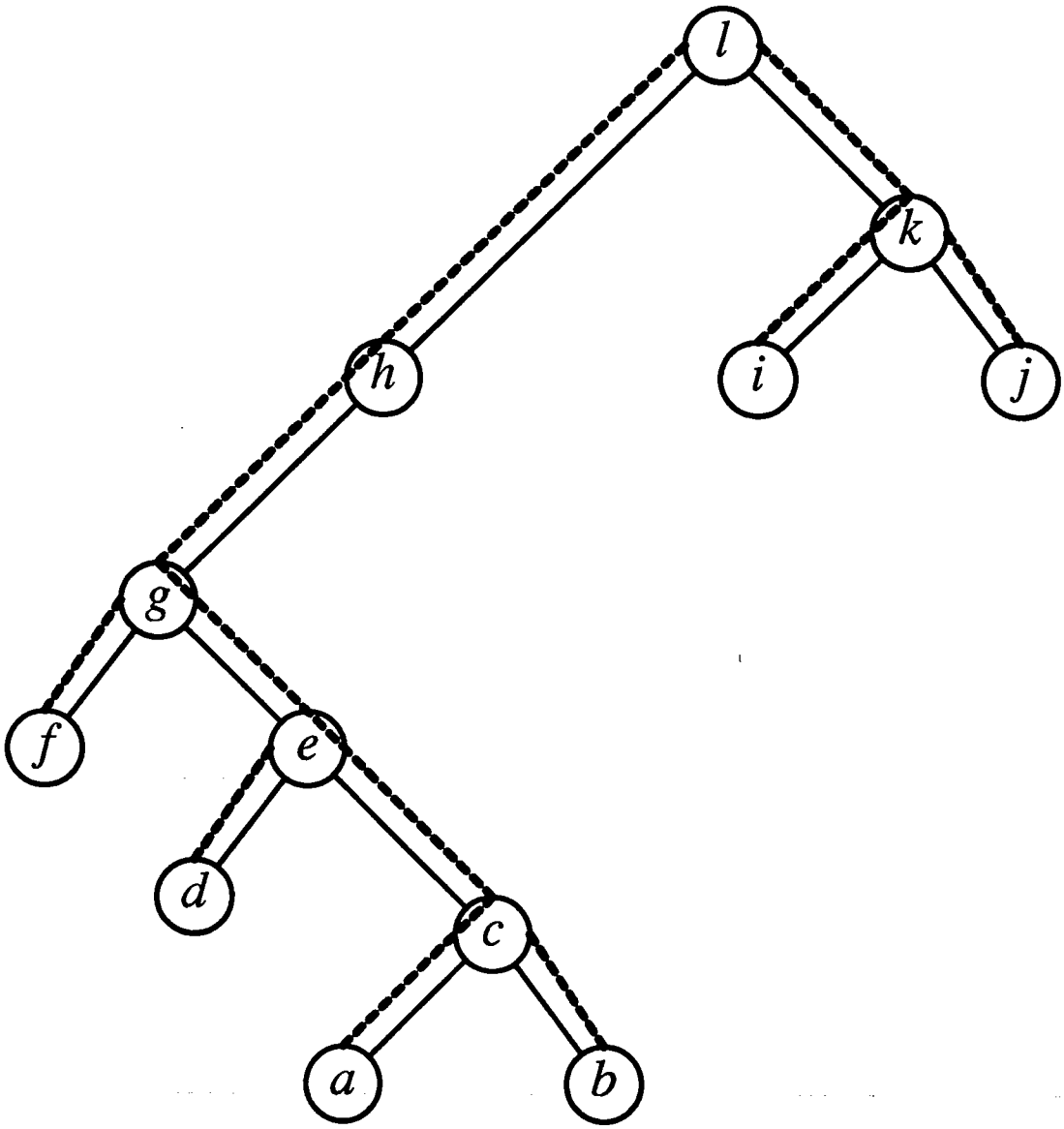

Figure 5.1. Maximal restricted paths for the binary tree in Figure 4.1.

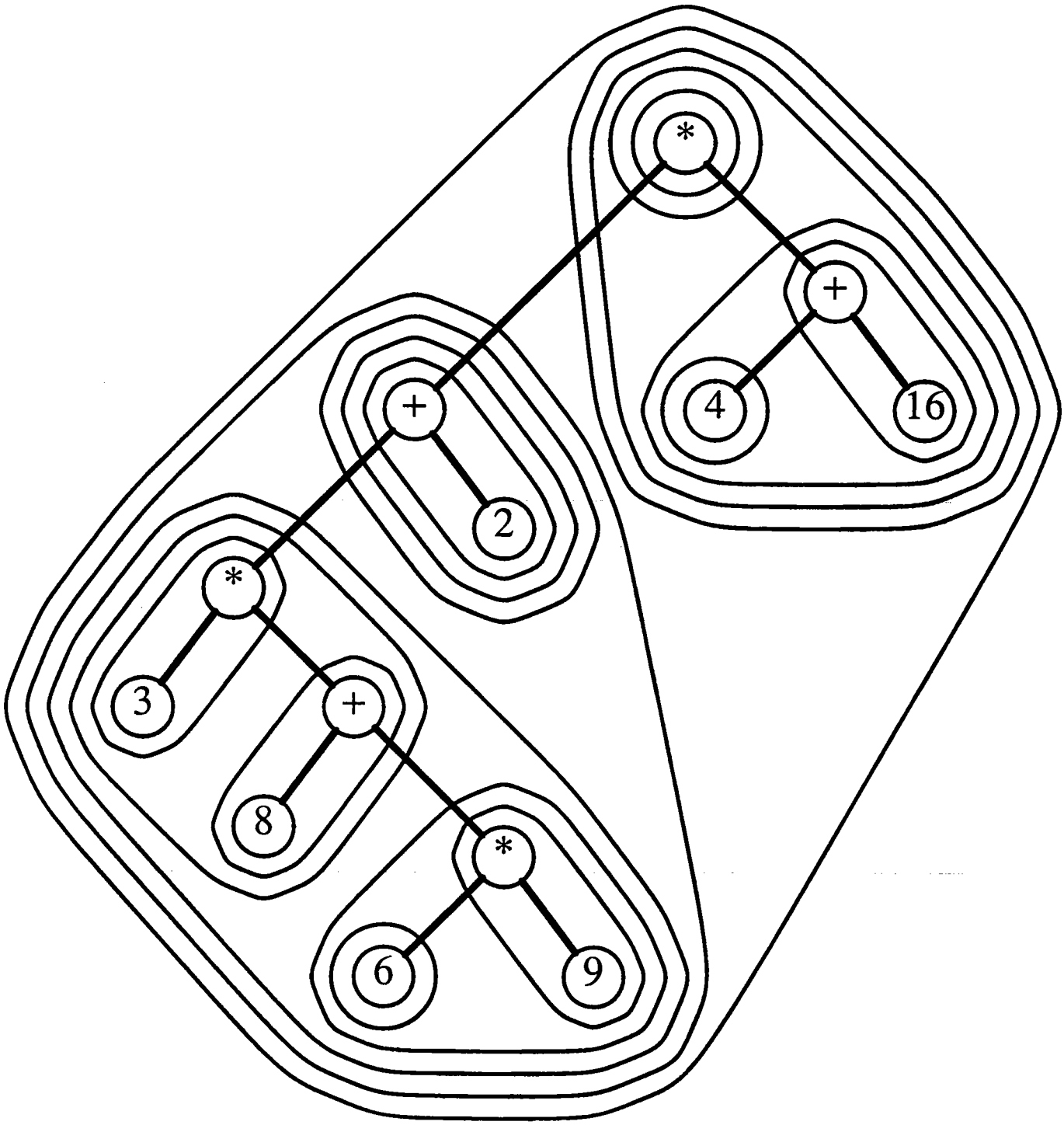

Figure 6.1. Expression tree and a multi-level partition

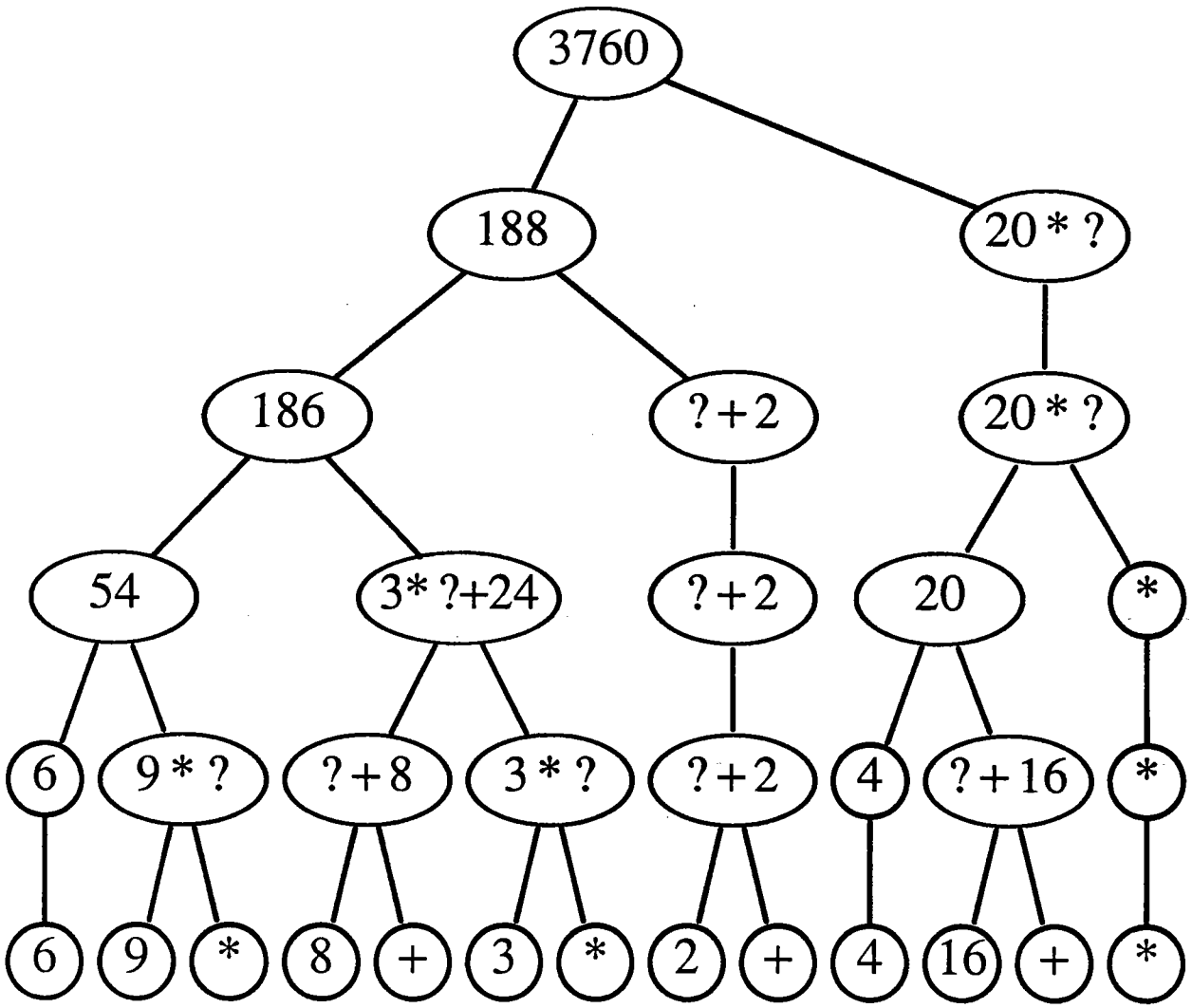

Figure 6.2. Topology tree for the expression tree in Figure 6.1.

generator	number of vertices	number of edges	splay tree total	splay tree link-cut	topology tree total	topology tree link-cut
NETGEN	1,000	10,000	252	106	316	146
NETGEN	1,000	10,000	254	106	322	149
NETGEN	1,000	20,000	397	110	460	156
NETGEN	1,000	20,000	393	109	461	154
NETGEN	1,000	30,000	536	114	600	166
NETGEN	1,000	30,000	535	114	599	164
NETGEN	1,000	40,000	672	115	724	163
NETGEN	1,000	40,000	678	120	742	179
NETGEN	1,000	50,000	817	121	879	186
NETGEN	1,000	50,000	818	120	873	183
NETGEN	1,000	60,000	950	117	989	173
NETGEN	1,000	60,000	956	126	1023	200
NETGEN	1,000	70,000	1088	124	1147	191
NETGEN	1,000	70,000	1115	140	1196	237
NETGEN	1,000	80,000	1240	124	1271	187
NETGEN	1,000	80,000	1251	134	1310	224
NETGEN	1,000	90,000	1379	120	1411	185

Table 7.1. Results of experiments