SOME FAST ALGORITHMS ON GRAPHS AND TREES

Heather Donnell Booth
(Thesis)

CS-TR-296-90

January 1991

SOME FAST ALGORITHMS ON GRAPHS AND TREES

Heather Donnell Booth

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1991

Abstract

Presented here are three fast algorithms solving problems concerning trees or graphs.

We give a linear-time algorithm for finding a balanced decomposition of a k-ary tree where k is a constant not related to the input size. A different linear-time algorithm for finding a balanced decomposition of a binary tree was discovered previously by Guibas, Leven, Hershberger, Sharir, and Tarjan [26].

We also consider the problem of finding a minimum-cost maximum flow in a series-parallel network. The algorithm presented here runs in $O(m \log m)$ time on an m-edge series-parallel network and requires $O(m \log \log m)$ space. This is an improvement over the algorithm presented by Bein, Brucker, and Tamir [6], which runs in $O(m \log m + mn)$ time, where n is the number of vertices. In an effort to improve the space requirement we also present an algorithm which uses $O(m)$ space but runs in $O(m \log m \log \log m)$ time.

Finally, we consider the problem of finding all replacement edges for a minimum spanning tree of a planar graph. We present an algorithm for solving this problem which runs in linear time. This algorithm also performs sensitivity analysis for the minimum spanning tree, shortest path, and network flow problems.

The first two algorithms presented rely on the use of balanced binary trees for efficient representation of data. We give an overview of the relevant red-black tree and finger tree techniques in introductory chapter.

Acknowledgements

I am indebted to many folks for many things, from help with text-processing and dog-walking to financial support.

For help with getting things to work and/or fruitful theoretical discussions I would like to thank fellow students Jim Plank, Neal Young, Adam Buchsbaum, Norman Ramsey, and Richard Squier. Mark Greenstreet, Andrew Tolmach, and Mordecai Golin have helped a great deal by reading and commenting on parts of my thesis. I am greatly indebted to Jeffery Westbrook with whom I collaborated on the work presented in Chapter 5.

I am grateful to Sharon Rogers for help of all sorts and to Rebecca Davies for generosity with her considerable LaTeX knowledge.

I would like to thank my official thesis-readers, Bernard Chazelle and Danny Sleator for their time and attention.

It has been a privilege to work with my advisor, Robert E. Tarjan. I am grateful that I had the opportunity to learn from him. His guidance has been invaluable. My work has been supported by a grant from the Office of Naval Research, Number N000014-87-K-0467 and a grant from Dimacs, Number NSF-STC88-09648.

I would also like to thank former teachers Michael Murphy, Hoyle Pennington, William McKee, and former professor Joseph O'Rourke.

I am particularly grateful to my family for providing much-needed financial and emotional support.

Heather D. Booth

Table of Contents

1. Introduction

Presented here are three efficient algorithms each of which solves a problem concerning graphs or trees. Our solutions to the first two problems require the use of balanced trees. For each problem we require a way of representing a list which can support efficient implementation of search queries, data modification of a certain sort, and concatenation and splitting. Red-black trees [28], B-trees [4, 5], and splay trees [44] all achieve the running time desired. We have chosen to use red-black trees. For the result described in Chapter 4, finger seach trees are used instead of red-black trees to give a simpler implementation.

The finger search trees used here are a variant of those introduced by Tarjan and Van Wyk in [51] which are based on red-black trees. Comprehensive overviews of red-black trees and two types of finger trees are given in Chapter 2. We describe the finger search trees of Tarjan and Van Wyk and also a variant which appears to be slightly simpler. The searching methods and proofs given here can often be found elsewhere although not necessarily in great detail. The description given here aims to fill in the gaps and to collect in one place enough detail so that the reader can easily verify correctness.

In Chapter 3 we give an algorithm for finding the balanced decomposition of a tree of constant maximum degree in linear time. A decomposition is formed by repeatedly splitting the tree by removal of an edge. The decomposition is balanced if each edge removal yields two components which have roughly equal size. Previously, Guibas, Leven, Hershberger, Sharir, and Tarjan [26] presented an algorithm solving this problem for binary trees in linear time. Our algorithm uses rather different techniques.

Our algorithm is fairly simple. The key observation is that a good splitting edge can be found using the preorder numbering of the tree. We are able to find the splitting edge in logarithmic time by representing the tree as a preorder list [15] stored in a balanced tree.

Considered in Chapter 4 is another problem whose solution relies on the use of balanced trees. We present an $O(m \log m)$ algorithm for finding the min-cost maximum flow of an m-edge series-parallel network. Bein, Brucker, and Tamir [6] gave an algorithm which runs in $O(m \log m + mn)$ time on an m-edge n-vertex series-parallel network. Series-parallel graphs are a subclass of graphs useful for representing circuits or flow of control in a program.

Series-parallel graphs can be constructed by performing *series* or *parallel* composition operations. An m-edge series-parallel graph can be decomposed (and represented by a *decomposition tree*) in linear time by using the algorithm of Valdes, Tarjan, and Lawler [53]. The relevant flow information for such a graph can be captured in a list called the *flow list* [6]. We show how to compute this list by processing the decomposition tree bottom-up. Using balanced tree structures to represent the flow lists gives the running time mentioned above.

The key to the algorithm is a procedure for computing the flow list of a series-parallel network from the flow lists of its two components. This procedure, called composition, is similar to merging. An efficient merging procedure described by Brown and Tarjan [8] which can merge lists of size n and m, $n \leq m$, in time $O(n \log \frac{n+m}{n})$ time by using balanced trees to represent the lists can be modified to perform composition. We give a simpler method for composing (and merging) which is equally fast, but which uses finger search trees to represent the lists.

The final algorithm presented is unrelated to the other two; it does not use balanced search trees. This algorithm finds all replacement edges of a minimum spanning tree of a planar graph in linear time. In addition, it can be used to perform sensitivity analysis for minimum spanning trees, shortest path trees, and for a particular network flow problem in the restricted case of planar input graphs. This problem was motivated by work done by Eppstein [19] on finding the k smallest minimum spanning trees.

Unlike the other two algorithms, the algorithm presented in Chapter 5 relies almost completely on properties of the input. The solution is elegant and simple. Queues and lists are the only data structures required. The problem of finding the replacement edges of a minimum spanning tree in a general graph in linear time (or proving a lower bound) is open. The best known deterministic algorithm, due to Tarjan [46], runs in $O(m\alpha(m, n))$ time on a graph with m edges and n vertices. A linear time randomized algorithm has recently been discovered by Dixon, Rauch, and Tarjan [17]. The work presented in Chapter 5 is joint work with Jeffery Westbrook.

Some terms which are used throughout are defined below.

A graph G is specified by its vertex and edge set (V, E). An undirected edge e incident on vertices (or, equivalently, *nodes*) u and v in V is denoted by $\{v, u\}$ or $\{u, v\}$. A directed edge (or arc) from v to u is denoted by (v, u). If nodes u and v are connected by an edge in graph G then u and v are said to be *neighbors*.

An *unrooted* or *free* tree is a graph containing no cycles. That is, there is only one path between each pair of nodes. In a *rooted* tree, one vertex is designated to be the *root*; the edges are implicitly directed away from the root. For each node u in a *rooted* tree T, the *parent* of u, denoted $p(u)$, is the neighbor of u which is on the path from u to the root. The *children* of u are the remaining neighbors of u. The *subtree* rooted at u is the set of nodes reachable from u by a path not passing through $p(u)$. The nodes contained in the subtree rooted at u are called *descendants* of u and u is an *ancestor* of any node in its sbutree.

For a node in a graph or unrooted tree, the *degree* of the node is the number of neighbors. For a node in a rooted tree the degree is the number of children. A *binary tree* is a rooted tree in which every node has degree 0, 1, or 2. The nodes with degree 0 are called *leaves* or *external nodes* and the remaining nodes are called *internal* nodes. The *level* or *depth* of a node u in a rooted tree is the number of edges on the path from the root to u. Thus the root is at level 0. A single-node tree is said to have 1 level.

A *full* binary tree is a binary tree in which every node has degree 0 or 2. A *complete* binary tree is a rooted binary tree in which every non-leaf node has degree 2 and all leaves are at the same level. Such a tree has 2^k nodes at level k.

Introductory remarks and concluding remarks are given within each chapter.

2. An Overview of Red-Black and Finger Trees

2.1 Introduction

Balanced trees are a well-known solution to the problem of maintaining an ordered list of items so as to be able to quickly perform the operations *insert*, *delete*, and *access*. These operations can be executed in $O(\log n)$ time on a list of n elements if the list is represented by a balanced tree. Several types of balanced tree structures have been studied in the literature, including AVL-trees [2, 1], B-trees [4, 5], and red-black trees [28]. With both red-black and B-trees, the more complicated list operations *split* and *concatenate*, can also be performed in logarithmic time.

A more recently developed data structure is the *finger tree*, which performs better than balanced trees when the tree operations exhibit some locality of reference. In finger trees we access data through pointers to a fixed number of leaves ("fingers") rather than through the root. This makes the access time dependent on the distance to the fingers rather than on the number of items in the list.

Finger trees were introduced by Guibas, McCreight, Plass, and Roberts [27], as a variant of B-trees. Huddleston and Mehlhorn refined this work in [34], still using B-trees. Tsakalidis has presented a finger tree data structure based on AVL trees [52]. Kosaraju [37] presents a more general structure which also has similar properties. In the appendix of [51], Tarjan and Van Wyk give another, simpler, implementation of finger trees. They describe a finger data structure which is a modification of red-black trees, but other forms of balanced trees could be used as a basis for the structure.

The two problems presented in Chapters 3 and 4 rely on the use of red-black and finger trees respectively. In this chapter we give a fairly complete overview of red-black trees, of the finger trees introduced by Tarjan and Van Wyk, and of a variant of these which we use in Chapter 4. The material here is intended to be comprehensive and useful as an introduction to these two

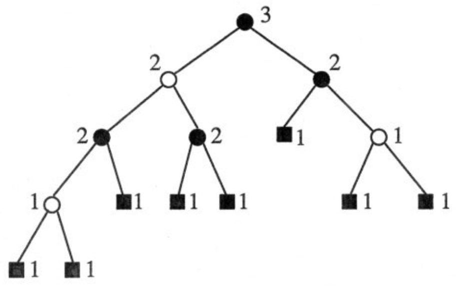


Figure 2.1: *A red-black tree. The darkened nodes are black nodes. The external nodes are denoted by squares. Shown with each node is its rank.*

types of data structures.

2.2 Red-Black Trees

A *red-black tree* is a full binary tree in which each node is assigned a color, either *red* or *black*. The leaves are called *external* nodes and the non-leaves are called *internal*. The node colors satisfy the following constraints:

(i) All external nodes are black.

(ii) All paths from the root to an external node contain the same number of black nodes.

(iii) Any red node with a parent has a black parent.

The *rank* of a node u is defined to be the number of black nodes along any path from u to an external node in its subtree. See Figure 2.1 for an example of a red-black tree with rank values shown. The *height* of node u is the maximum number of nodes along the longest path from u to one of its external descendants. A node with rank r has height between r and $2r$ and has between 2^{r-1} and 2^{2r-1} external descendants. Thus a red-black tree with n external nodes has depth $O(\log n)$. We will refer to the rank of the root of a tree as the rank of the tree.

For a node x we use $p(x)$ to denote the parent of x, $\mathit{left}(x)$ and $\mathit{right}(x)$ to denote the left and right children of x, and $\mathit{sib}(x)$ to denote the sibling of x.

There are two ways to represent a list using red-black trees. The items can be stored either in the internal nodes, so that order in the list corresponds to symmetric order in the tree, or in the external nodes, so that order in the list corresponds to left-to-right order among the external nodes. We will use the latter representation. We will also assume that the keys associated with list items are real numbers. Each red-black tree node will have pointers to its left and right children. The rank of each tree is stored with its root.

Given a list represented as a red-black tree T we may wish to access a particular item of the list, for example, the item with key k or perhaps the k^{th} item in the list. Performing any kind of access requires storing auxiliary data in the internal nodes. For example, in order to find the node with key k we store with each internal node u the maximum key associated with an external node in u's subtree. We will describe two broad types of search operations and how they are executed. Both types of search are used in the algorithm presented in Chapter 4. First we describe how the list operations *insert*, *delete*, *concatenate*, and *split* are implemented in a red-black tree.

Inserting into and deleting from red-black trees is accomplished by the addition and deletion of external nodes. Such updates require rebalancing the tree, which is done by recoloring nodes and performing *rotations*, which locally change the tree structure.

To insert a new item after a specified external node x, replace x by a new internal node having x and a new external node containing the new item as children. The new internal node is colored red. If its parent is also red, this violates the red constraint (iii). Such a violation is corrected by walking up the tree path from the violation, repeatedly applying the recoloring transformation of Figure 2.2(a) until it no longer applies, and then if necessary applying one of the transformations in Figures 2.2(b),(c), or (d). (Transformations (b) and (d) change the tree structure as well as the node colors; transformation (b) does one rotation; transformation (d), two.) Note that there are symmetric

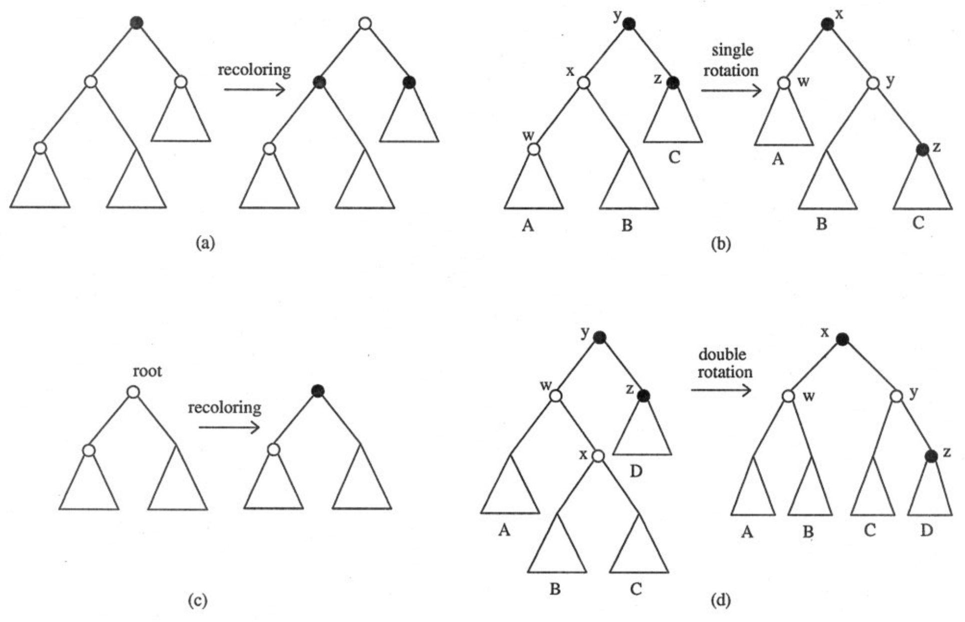


Figure 2.2: *Rebalancing transformations for insertions. There are symmetric cases not shown.*

cases which are not shown.

Deleting an item is similar but slightly more complicated. To delete an external node x with sibling y, replace $p(x)$ by y. If the deleted internal node was black, this violates the black constraint. That is, y is *short*, i.e., paths down from it contain one too few black nodes. The violation is corrected by walking up the tree path from the violation, repeatedly applying the transformation of Figure 2.3(a) until it no longer applies, followed if necessary by one application of Figure 2.3(b) and then possibly one application of Figure 2.3(c), (d), (e), or (f). Altogether at most three rotations are done.

The worst-case time needed to insert or delete an external node in a red-black tree is $O(\log n)$, but the amortized time[1] is only $O(1)$. To prove this, we define the *potential* of a red-black tree to be the number of black nodes with two black children plus twice the number of black nodes with two red children.

[1]By *amortized time* we mean roughly the time of an operation averaged over a worst-case sequence of operations. See [45].

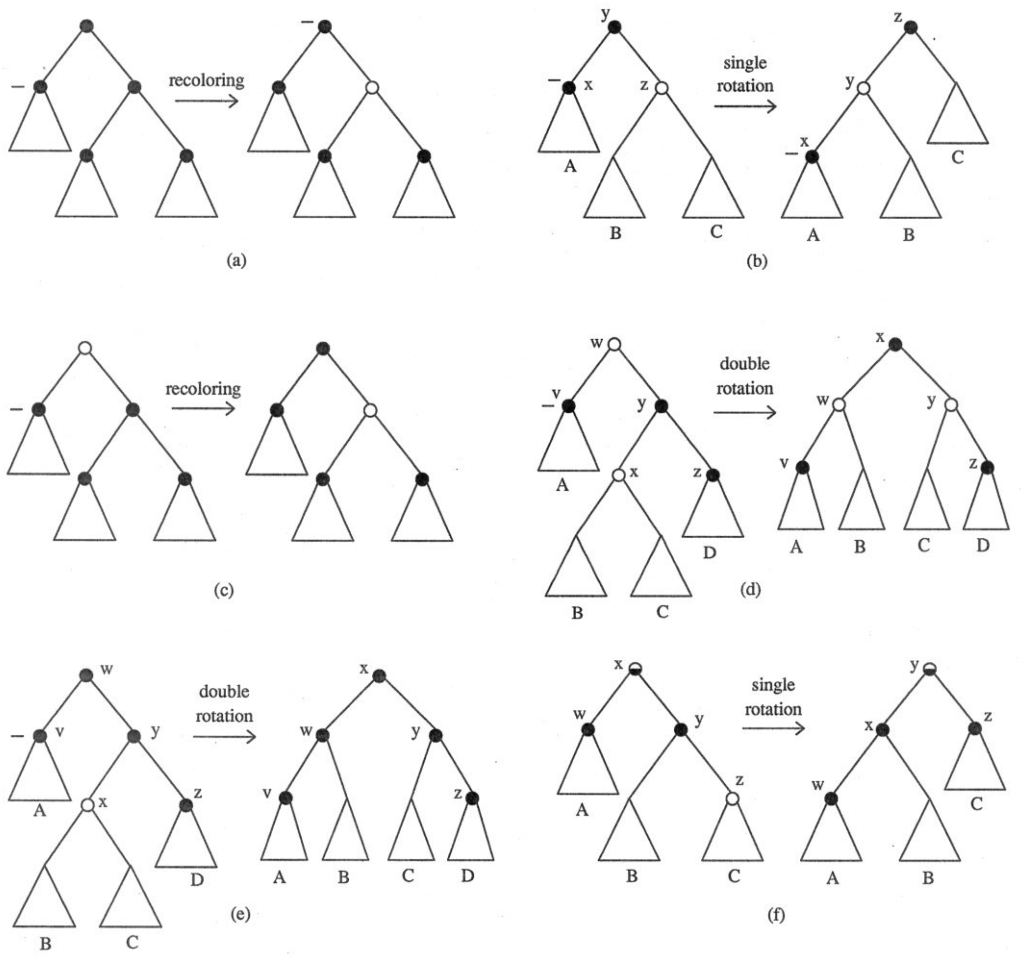


Figure 2.3: *Rebalancing transformations for deletions. The half-darkened nodes in (f) are nodes which can be either red or black, but they must both be the same color. A "-" by a node indicates that it is short.*

We define the *actual time* of an insertion or deletion to be one plus the number of transformations applied and the *amortized* time to be the actual time plus the net increase in potential caused by the tree restructuring. Each application of the (non-terminal) transformations of figures 2.2(a) or 2.3(a) reduces the potential by at least one. An application of any of the other transformations either decreases the potential or increases it by at most two. This implies that the amortized time of an insertion or deletion is $O(1)$, since there are at most a constant number of applications of the latter type of transformation. If we begin with an empty tree and perform a sequence of k external insertions and deletions, the total time for all the tree restructuring is $O(k)$, because the amortized time is $O(k)$, the actual time is the amortized time minus the net increase in potential over the sequence, and the net increase in potential is nonnegative (since the initial potential is zero and the final potential is nonnegative.)

On a list represented by red-black trees we can also perform the more complicated operations *concatenate* and *split* in logarithmic time.

Concatenate is the simpler operation. Let T_1 and T_2 be two red-black trees with ranks r_1 and r_2 respectively. Let x_1 be the root of T_1 and x_2 the root of T_2. We will assume that $r_1 \leq r_2$ and that we wish to concatenate T_1 on the left of T_2; the other cases are analogous.

If $r_1 = r_2$, create new root i, having x_1 as left child and x_2 as right child. If both x_1 and x_2 are black, make i red and otherwise make it black. Note that although the balance constraints can be satisfied by making the new root black in either case, this would unnecessarily increase the rank of the resulting tree, which would adversely affect the splitting operation described below.

Otherwise $r_2 > r_1$. If x_1 is red, make it black. Let $r_1{}'$ denote the current rank of x_1; $r_1{}'$ is r_1 if x_1 was black to start with and is $r_1 + 1$ otherwise. Walk down the left path of T_2 until reaching first black node u having rank equal to $r_1{}'$. Create new node i having x_1 as left child and u as right child. Replace u by i and make i red. If this causes a violation to the red constraint (if the parent of i is red), then rebalance as for an insertion. Notice that both

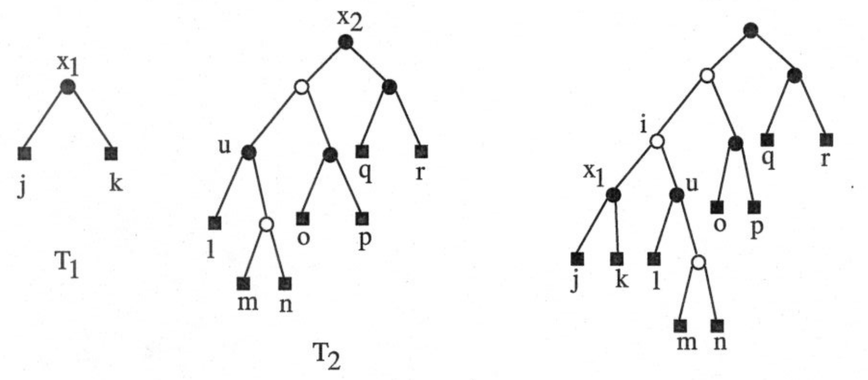


Figure 2.4: *Concatenating two red-black trees. Shown are the original trees, T_1 and T_2, and the result before rebalancing.*

children of i are black. This procedure (illustrated in Figure 2.4) requires $O(\max\{|r_1 - r_2|, 1\})$ time in the worst-case.

We will need the following property about the rank, r, of the tree resulting from such a concatenation: if $r_2 = r_1$ and both roots were originally black or if $r_2 > r_1$ and x_2 was black then r is r_2; otherwise r is r_2 or $r_2 + 1$. Further, if the rank of the resulting tree is $r_2 + 1$, then its root is black.

If the trees are concatenated by creating a new root with x_1 and x_2 as children, then an analysis of the possible cases above shows that the property holds. Otherwise the resulting tree is formed by modifying the structure of T_2, leaving the root intact, and then rebalancing as for an insertion if necessary. The rank of this tree is the same as the rank of T_2 unless rebalancing causes the rank of the root to increase. Inspection of the relevant rebalancing transformations (see Figure 2.2) reveals that this can occur only if x_2 was originally red. This implies that for $r_2 > r_1'$, the rank of the resulting tree is r_2 if x_2 was originally black, as claimed. Note that the rank of the root can increase by at most one; in all such cases the new root is black.

The inverse of concatenating is *splitting*. Splitting a tree T at node x yields trees T_l containing the items to the left of x and T_r containing the items to the right of x, in order. In the implementation described here x is not contained in either resulting tree. It is easy to modify this implementation so that x is

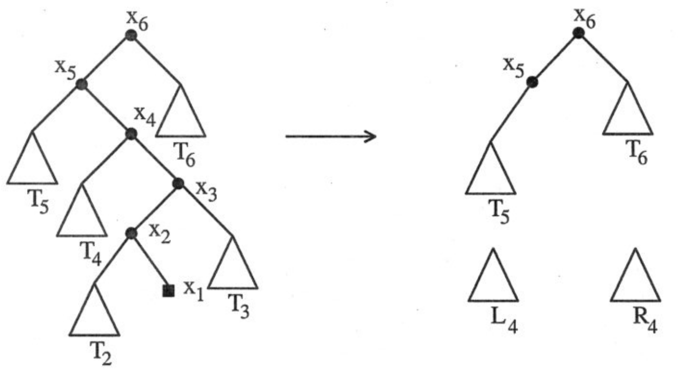


Figure 2.5: *Splitting a red-black tree. After four stages, L_4 is the result of concatenating T_2 and T_4 and R_4 is T_3.*

included either in T_l or in T_r, as desired. We assume that the path from x to the root of T is known. Splitting is accomplished by cutting off all subtrees of this path and then concatenating all the left ones in order to form T_l and then all right ones to form T_r.

More formally, let the nodes on the path be $x_1, \ldots x_l$ where $x_1 = x$ and x_l is the root. Let T_i, $2 \leq i \leq l$, be the subtree of x_i that does not contain x_{i-1}. Let L_i be the tree formed by concatenating those left subtrees of $x_2, \ldots, x_i$ that do not contain x_1 and let R_i be defined analogously for the right subtrees. This is illustrated in Figure 2.5.

Splitting is accomplished as follows. Initially L_1 and R_1 are empty. At time i, compute L_i and R_i. If x_{i-1} is a left child, then $L_i = L_{i-1}$ and R_i is formed by concatenating R_{i-1} and T_i. If x_{i-1} is a right child, then $R_i = R_{i-1}$ and L_i is formed by concatenating T_i and L_{i-1}. The final trees L_l and R_l are T_l and T_r.

We will show that the time required for splitting an n-node tree is $O(\log n)$. The cost is essentially a telescoping series, but the analysis is involved. Let $cost_i$ denote the time required for executing the first i stages as described above. Note that $cost_i = cost_{i-1} + 1+$ the cost of performing the i^{th} concatenation. Recall that the worst-case time to concatenate two trees of rank r_1 and r_2 is $O(\max\{|r_1 - r_2|, 1\})$. For simplicity we will assume here that the

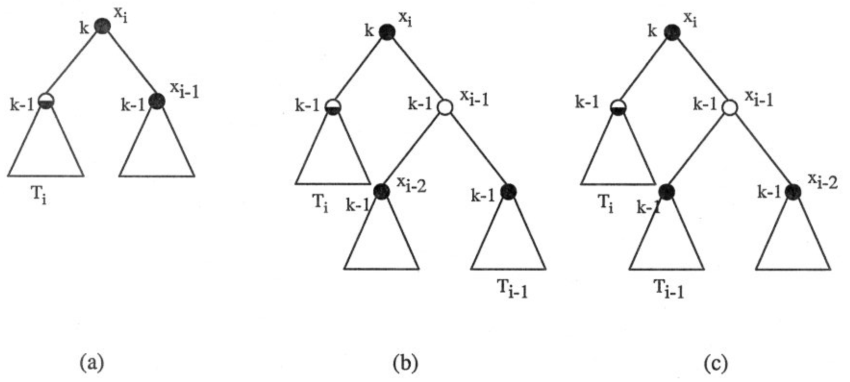


Figure 2.6: *The cases of the proof of Lemma 2.1. The half-darkened nodes may be either black or red.*

constant is 1; this affects only the constant of the running time of the splitting algorithm.

Lemma 2.1 *The time required to split an n-node red-black tree at an external node is $O(\log n)$.*

Proof: We show by induction on the black nodes of the path that the following invariant is maintained: at the i^{th} stage, if x_i is black, then $r(L_i) \leq r(x_i)$, $r(R_i) \leq r(x_i)$, and $cost_i \leq 4r(x_i) + r(L_i) + r(R_i)$. When $i = l$ this implies that $cost_l$, which is the time required to execute the entire splitting process, is $O(\log n)$, since the ranks of x_l, L_l and R_l are all $O(\log n)$.

When $i = 1$, the invariant is satisfied since $r(x_1) = 1$, L_1 and R_1 are empty and have rank 0 and $cost_1 = 0$.

We will show that the invariant holds at stage i if x_i is black. Assume that the invariant holds for all black nodes x_j such that $j < i$. Assume also that x_{i-1} is a right child; the other case is symmetric. Let k denote the rank of x_i. Both children of x_i have rank $k - 1$.

Case 1 : x_{i-1} is black. (See Figure 2.6(a).)
By the inductive hypothesis, the invariant holds for x_{i-1}. This implies that the ranks of L_{i-1} and R_{i-1} are at most $k - 1$ and that $cost_{i-1}$ is at most $4r(x_{i-1}) + r(L_{i-1}) + r(R_{i-1})$.

The invariant holds with respect to R_i since $R_i = R_{i-1}$ which has rank at most $k-1$. Next consider L_i, which is formed by concatenating T_i and L_{i-1}. Since both these trees have rank at most $k-1$, L_i has rank at most k. Finally, consider $cost_i$, which is $cost_{i-1} + 1 + \max\{|r(T_i) - r(L_{i-1})|, 1\}$. Using the facts that $r(T_i) \geq r(L_{i-1})$ and that for all i the rank of L_i (R_i) is no less than the rank of L_{i-1} (R_{i-1}) gives: $cost_i \leq 4k - 4 + r(R_i) + 1 + \max\{r(T_i), r(L_{i-1}) + 1\}$. Substituting $r(L_i) + 1$ for the last term gives the desired result.

Case 2 : x_{i-1} is red.
There are two sub-cases depending on whether x_{i-2} is a left or a right child. In both cases, both children of x_{i-1} are black. The children of x_i and of x_{i-1} all have rank $k-1$. Since the invariant holds for x_{i-2} we know that the ranks of L_{i-2} and R_{i-2} are at most $k-1$ and that $cost_{i-2} \leq 4(k-1) + r(L_{i-2}) + r(R_{i-2})$.

Case 2a : x_{i-2} is a left child (see Figure 2.6(b)).
On the $(i-1)^{st}$ step, R_{i-1} is formed by concatenating R_{i-2} and T_{i-1}. The rank of R_{i-1} is at most k since the ranks of the two constituent trees are at most $k-1$. The rank of R_i is also at most k, since $R_i = R_{i-1}$. On the i^{th} step, L_i is formed by concatenating T_i and L_{i-1}, which is the same as L_{i-2}. Again, both constituent trees have rank at most $k-1$, making the rank of L_i at most k.

The cost at stage i is at most $cost_{i-2} + 1 + \max\{|r(T_i) - r(L_{i-1})|, 1\} + 1 + \max\{|r(T_{i-1}) - r(R_{i-2})|, 1\}$. The techniques used in Case 1 above can be used here to show that $cost_i \leq 4k + r(L_i) + r(R_i)$, which completes the proof that the invariant is satisfied in this case.

Case 2b : x_{i-2} is a right child (see Figure 2.6(c)).
Here R_i is the same as R_{i-2} and has rank at most $k-1$ which is less than $r(x_i)$. Consider L_i. In the $(i-1)^{st}$ stage L_{i-1} is formed by concatenating T_{i-1} and L_{i-2}. The resulting tree has rank k or $k-1$. By the concatenation property given previously, the rank of L_{i-1} will be $k-1$ or k; if it is k then the root of L_{i-1} must be black. If L_{i-1} has rank $k-1$ then $r(L_i) \leq k$ since T_i has rank $k-1$. If L_{i-1} has rank k, then we are concatenating a tree with rank k and

black root with a tree of rank $k-1$. The resulting tree, L_i, has rank k.

The cost at time i is

$$cost_{i-2} + 2 + \max\{|r(T_{i-1} - r(L_{i-2})|, 1\} + \max\{|r(T_i) - r(L_{i-1})|, 1\}.$$

The reasoning used in the previous cases can be used to show that this is at most $4k - 1 + r(R_i) + r(L_i) + \max\{|r(L_{i-1}) - r(T_i)|, 1\}$. Since the value of the last term is at most 1, $cost_i$ also satisfies the invariant. ∎

Searching in a list represented by a balanced tree requires logarithmic time. Recall that an n-element list is represented by a red-black tree with n external nodes, in which each item and its key is associated with an external node such that order in the list corresponds to left-to-right order among the external nodes.

There are two general types of search operations which we will describe here. First we describe, for given value k, how to find the leftmost item with key at least k. This does not assume any ordering among the keys.

To perform this type of search store with each internal node u the value $maxkey(u)$ which is the maximum key of an external descendant of u. The desired item is found by searching down the path from the root. If u is the current node on the path and v is its left child, then go left if $maxkey(v) \geq k$ and go right otherwise. The external node reached is the leftmost external node with key at least k.

It is easy to modify this in order to find the rightmost item with key at least k. By storing *minkey* values instead of *maxkey* we can find the leftmost (rightmost) item with key at most k.

We can also search on *cumulative* key values, if the keys are all non-negative. That is, given value k, find the leftmost item x such that the sum of the keys of items up to and including x is at least k. For example, if all keys are 1, this would find the k^{th} item.

For external node x let $cumkey(x)$ denote the sum of keys of external nodes up to and including x. One way to perform the above search would be to consider the *cumkey* values as the keys and to perform the search given above.

In order to achieve this we would have to store with each node u the value $maxcumkey(u)$, which is the maximum *cumkey* value of an external descendant of u. Note that this equivalent to the *maxcumkey* value of the rightmost external descendant of u. If data is stored in this manner, then operations modify the tree (ie, insertions and deletions) may potentially invalidate many *maxcumkey* values. For example, deletion of external node x invalidates the *cumkey* data of all external nodes succeeding x which is likely to invalidate the *maxcumkey* data of all ancestors of these nodes.

Instead we store with each internal node u the value $totkey(u)$ which is the sum of the keys of all the external descendants of u. From these values we can compute *maxcumkey* values as they are needed. Again the search traverses the path from the root to the desired node. Let u be the current location of the search and let v be the left child of u and w the right child. Initially u is the root and $maxcumkey(u) = totkey(u)$. At a general step compute $maxcumkey(v) = maxcumkey(u) - totkey(w)$. If this value is greater than or equal to k go left; otherwise go right and set $maxcumkey(w) = maxcumkey(u)$.

In accessing an item both of the search operations described above traverse the path from the root to the desired item. This requires $O(\log n)$ time in an n node red-black tree.

It is also necessary to describe how to compute and maintain the auxiliary data. Computing the auxiliary data initially requires $O(n)$ time for an n-node tree since both *maxkey* and *totkey* have recursive definitions and can be computed bottom-up. Note that tree restructuring such can invalidate some auxiliary data values. These values can be updated quickly as long as the data is defined recursively; that is, the value associated with a node can be computed from the values associated with its children.

Insertion or deletion at an external node x may invalidate the auxiliary data of the ancestors of x only. Concatenating may invalidate the data of all nodes along a path from the new node (i in Figure 2.4) to the root. This data can be recomputed by processing the affected path bottom-up. This will take at most $O(\log n)$ time after an insertion into or a deletion from an n-node tree, making the amortized (as well as the worst-case) time $O(\log n)$. Updating data after a concatenation adds only a constant factor to the running time.

The auxiliary data is also affected by rotations occurring during rebalancing. A rotation affects the data values of only a constant number of nodes. It is easy to see that this data can be recomputed (bottom-up) in constant time per rotation.

Another useful operation that can be executed is *add value.* Given a value k and a node u we can add the value k to the keys of all items in u's subtree in constant time. This operation is compatible only with the search on *maxkey* values. That is, we cannot perform *add value* operations and also search on cumulative keys in logarithmic time. We will assume from now on that each item has two keys, *key1* and *key2.* The *maxkey* values are defined relative to *key1* values and *totkey* and *cumkey* relative to *key2.* Add value operations affect *key1* values only.

In order to support *add value* we store the *maxkey* data in difference form. With each non-root node x we define $\Delta maxkey(u) = maxkey(u) - maxkey(p(u))$. If u is the root, $\Delta maxkey(u) = maxkey(u)$. This is illustrated in Figure 2.7. Note that for a node u, $maxkey(u) = \sum\{\Delta maxkey(v) | v$ is an ancestor of $u\}$. If $\Delta maxkey$ values are stored and we are given the *maxkey* value of a node u then we can compute the *maxkey* values for the parent or children of u in constant time.

In order to perform *add value*(k, u) to add the value k to the keys of all external descendants of u, simply add k to $\Delta maxkey(u)$. It is also possible to add a value to the keys of all external nodes to the left of a given external node x. This requires time proportional to the depth of the tree (assuming that the path from the root to x is given) and is accomplished by traversing this path, at each node u adding k to $\Delta maxkey(sib(u))$ if $sib(u)$ is a left child. An analogous procedure can be used to add value to all items contained in external nodes to the right of x. Combining the two procedures allows us also to add a value to the keys of all items between two specified external nodes.

If $\Delta maxkey$ values are stored instead of *maxkey* values, then extra work

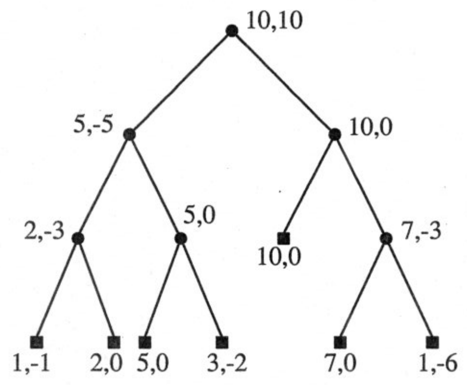


Figure 2.7: *Data required for performing an* add value *operation. Shown with each node u is* maxkey(u), Δmaxkey(u). *For each external node u,* maxkey(u) = key(u).

is required in the splitting operation. In order to split at node x we need to compute $maxkey(x)$ and to maintain the *maxkey* value of the current node as the path up from x is traversed. When a subtree of this path is cut off, the $\Delta maxkey$ value of the root of the subtree is replaced by the *maxkey* value of the root. Taken by itself, each subtree hanging off the path now has correct $\Delta maxkey$ values and the splitting algorithm can proceed as described above. This extra computation adds only a constant factor to the running time.

Maintaining $\Delta maxkey$ values is harder than maintaining *maxkey* or *totkey*. Again values may be invalidated by structural changes and by rotations. First consider the changes caused by modifying the tree. Modifying the contents of the subtree rooted at node x may invalidate the *maxkey* values of the nodes on the path from x to the root and the $\Delta maxkey$ values of these nodes and their siblings. These values can be recomputed by traversing the path from x to the root as long as the old and new *maxkey* values for x are known. Let *maxkey* and $\Delta maxkey$ denote the old values and $maxkey'$ and $\Delta maxkey'$ the new ones. At each node x we compute $\Delta maxkey'(x)$ and $\Delta maxkey'(sib(x))$, where $sib(x)$ denotes the sibling of x, as follows:

$$maxkey(p(x)) = maxkey(x) - \Delta maxkey(x)$$

$$\begin{aligned}
&maxkey(sib(x)) = maxkey(p(x)) + \Delta maxkey(sib(x))\\
&maxkey'(p(x)) = \max\{maxkey'(x), maxkey(sib(x))\}\\
&\Delta maxkey'(x) = maxkey'(x) - maxkey'(p(x))\\
&\Delta maxkey'(sib(x)) = maxkey(sib(x)) - maxkey'(p(x)).
\end{aligned}$$

Now it is possible to process $p(x)$. This procedure is used for updating $\Delta maxkey$ data after an insertion, deletion, or concatenation. The time required is the length of the path traversed, and is proportional to the worst-case time required for the operation itself.

Auxiliary values may also be invalidated by rotations. To recompute these values, we maintain the *maxkey* value of the current node while rebalancing. After a rotation, this value is used to compute first the old *maxkey* values from which we compute the new *maxkey* values and new $\Delta maxkey$ values, which are then stored. This requires constant time per rotation.

To summarize, we have described a data structure that is capable of supporting any sequence of insertions, deletions, concatenations, splittings, and accesses of the form described above. Except for concatenation, all operations require $O(\log n)$ time on an n-node tree. Concatenation of two trees of ranks r_1 and r_2 requires $O(\max\{|r_1 - r_2|, 1\})$ time. In $O(\log n)$ time we can access the leftmost external node with *key* at least k or the leftmost with *cumkey* at least k and can also add a constant to the values of all external nodes in the tree, or to the values of all external nodes between two chosen ones.

2.3 Finger Trees

In this section we describe the finger tree data structure presented by Tarjan and Van Wyk in [51]. We will refer to these trees simply as finger trees. Also presented is a cleaner variant of this data structure which has slightly worse access time.

In a regular red-black tree, the *left path* is the path from the root to the leftmost external node and the *right path* is the path from the root to the

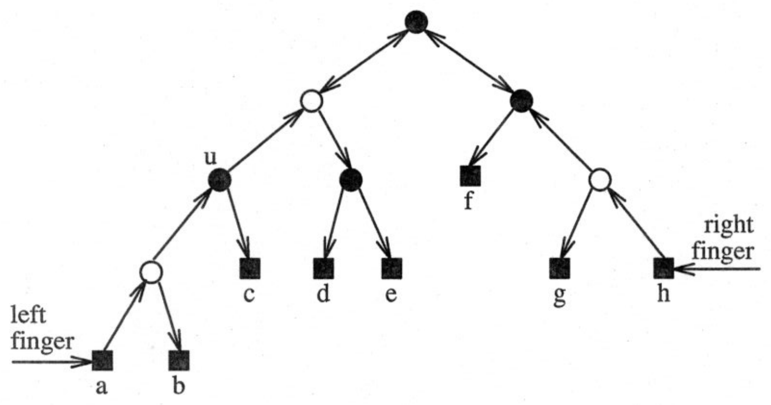


Figure 2.8: *A finger tree.*

rightmost external node. To convert a red-black tree into a finger tree we reverse the pointers along the left and right paths and add pointers from the root to its two children.

For non-root nodes on the left path in the finger tree the left pointer now points to the parent in the red-black tree. For non-root nodes on the right path the right pointer points to the parent. The pointers from the root are unchanged. The tree is accessed via pointers to the leftmost and rightmost external nodes. These are called the *left finger* and *right finger*, respectively. See Figure 2.8 for an example.

For such a finger tree with n external nodes, inserting, deleting, or splitting at the d^{th} node requires $O(1 + \log(\min\{d, n - d\}))$ time, as does accessing the d^{th} node. Concatenating requires time proportional to the rank of the smaller tree. All these time bounds are in the amortized sense.

The nodes *reachable* from a node u in the finger tree are those nodes connected to u by a path of pointers that does not pass through the root. It will later be clear why we exclude paths which pass through the root. The *subtree* of u is the set of nodes which would be reachable if the pointers were not reversed. For example, in Figure 2.8, c, d, and e are the external nodes reachable from u and a, b, and c are the external nodes contained in the subtree of u. A non-root node is called a *path* node if it is on the left or right path and a *non-path* node otherwise. The *non-path subtree* of a node u on the left or right

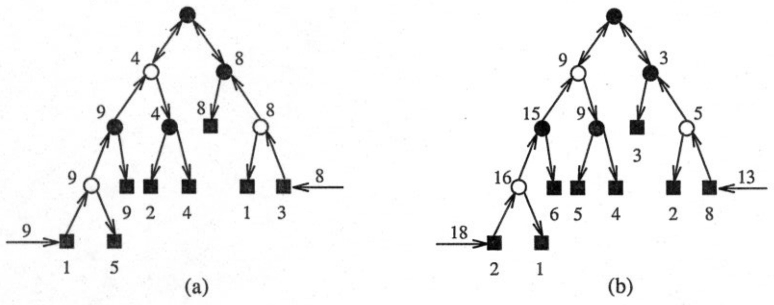


Figure 2.9: *Figure (a) illustrates a finger tree shown with* key1 *and* maxkey *values. In Figure (b) the same tree with* key2 *and* totkey *values shown.*

path is the subtree rooted at the non-path child of u.

A list of items is again represented by associating the items with the external nodes, in order. We assume that each item has two associated real values, *key1* and *key2*. For each non-root node u in the tree let $maxkey(u)$ be $\max\{key1(x)|x$ is an external node reachable from $u\}$, and let $totkey(u)$ be $\sum\{key2(x)|x$ is an external node reachable from $u\}$. Note that these values are defined in terms of the reversed pointers along the left and right paths. In order to perform the search operations described previously we store these values with each node except for the root and with each finger. Figure 2.9 contains an example illustrating these values.

A node is accessed by traversing the path from one of the fingers. There are two such paths; one passes through the root and one does not. A node is always accessed by the path which does not pass through the root, which is called the *access path.* Although this path may be longer than the other one, it will be longer by at most a factor of 2. The finger from which the access path starts is called the *nearer* finger. The advantage of finger search trees over red-black trees is that the time required to access a node is proportional to the distance from a finger rather than on the distance to the root; this distance is $O(1 + \log(\min\{d, n - d\}))$ if the node is the d^{th} and the tree has n external nodes altogether, as shown below.

Lemma 2.2 *The length of the access path to the d^{th} external node is $O(1 + \log(\min\{d, n - d\}))$.*

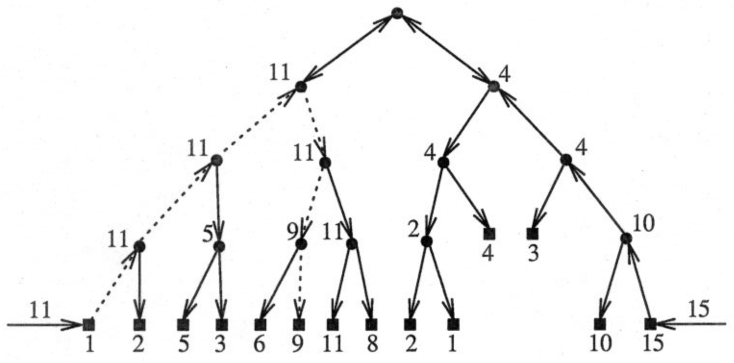
Proof: Let x be the d^{th} external node in an n-node tree. Assume x is in the left subtree; the other case is symmetric. We will show that the length of the left access path is $O(1 + \log d)$ and that this is also $O(1 + \log(n - d))$. The case when x is in the right subtree is analogous.

Let u be the left path node whose non-path subtree contains x. The left access path consists of the path nodes from the left finger up to u and the path from u down to x. Its length is $O(rank(u))$. If $d = 1$ then the path has length one. Otherwise let v be the predecessor of u on the left path (ie, u is the left child of v). Since v has fewer than d external descendants, we know that v has rank at most $\log d + 1$. This implies that the rank of u is at most $\log d + 2$ and thus that the length of the left access path is $O(1 + \log d)$.

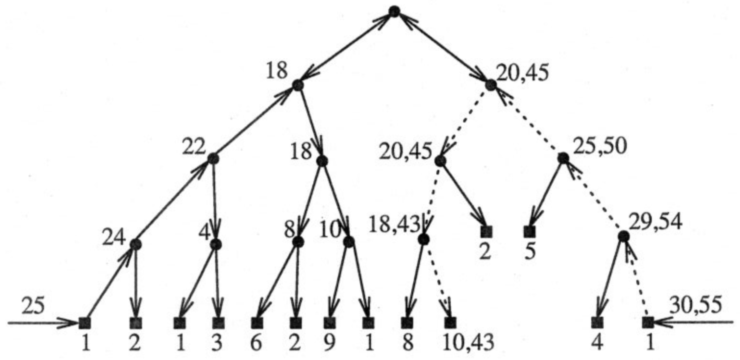
Given that x is in the left subtree we can also show that this value is $O(1 + \log(n - d))$. Let z be the right child of the root. The number of external descendants of z is at most $n - d$. Thus the rank of z is at most $\log(n - d) + 1$. The rank of z is at least $r - 1$, where r is the rank of the root of the tree. This is at least $(\log n - 1)/2$, which implies that $\log n \leq 2\log(n - d) + 3$. Since $d \leq n$, this implies that that $1 + \log d$ is $O(1 + \log(n - d))$, as claimed. ∎

Both types of queries described in the previous section can be performed on finger trees as well. Both searches proceed by finding the node on the left or right path whose non-path subtree contains the item and then searching down in the indicated non-path subtree (which is a regular red-black tree) for the desired external node.

First we describe how to find the leftmost node with key at least k. Detecting whether such a key exists can be done in constant time; a tree contains an item with $key1 \geq k$ if and only if one of the fingers has $maxkey$ value at least k. If the maxkey value of the left finger is at least k (the item is reachable from the left finger) then walk up the left path until reaching the first node u with non-path child v such that $maxkey(v) \geq k$. Then search down in v's subtree for the leftmost external node with key at least k. Otherwise, walk up



(a) k = 8



(b) k = 40

Figure 2.10: *Searching in finger trees. A tree with* maxkey *values is shown in Figure (a). The dashed edges comprise the access path for a search on* maxkey *with $k = 8$. The same tree is shown in Figure (b) with the* totkey *value associated with each node. Also shown are the* maxcumkey *values maintained along the access path (which is dashed) during the search for the leftmost node with* cumkey $\geq k$*, for $k = 40$.*

the right path until reaching the last node u such that $maxkey(u) \geq k$. Then search down in u's non-path subtree for the leftmost external node with key at least k. This is illustrated in Figure 2.10 (a).

Finding the leftmost external node x such that the sum of keys up to and including x is at least k is similar. Such a node exists if and only if the sum of the *totkey* values of the left and right fingers is at least k. As in Section 2.2, the search is guided by *maxcumkey* values, where *maxcumkey*(u) is the maximum *cumkey* value of an external node in u's subtree. This may not be the same as the maximum *cumkey* value of an external node reachable from u.

If the *totkey* value of the left finger is at least k, set $maxcumkey = 0$ and walk up the left path, maintaining *maxcumkey*, until $maxcumkey \geq k$. To update this value when going from node u to its successor on the left path, add $totkey(v)$ where v is the right child of u. Next search down in the non-path subtree as in a regular red-black tree. If the item is not in the left subtree, search from the right finger as follows. Set *maxcumkey* to be the sum of the *totkey* values of the left and right fingers. Walk up the right path maintaining *maxcumkey* until reaching the first node u whose successor has *maxcumkey* value less than k. To update *maxcumkey* when going from a node v to its successor, subtract the *totkey* value of v's left child. Then search for the leftmost node with *cumkey* at least k in the left subtree of u. See Figure 2.10(b) for an example.

In both cases, accessing a node requires time proportional to the length of the access path, which is $O(1 + \log \min\{d, n - d\})$ for the d^{th} external node in an n-node tree.

As described in [51], it is also possible to access a node by searching simultaneously from both fingers. Here a node is always accessed via the access path (i.e., the path from the "nearer" finger), even though the other path may be shorter by as much as a constant factor. Which method one should use depends on the application. If after accessing an external node x we wish to insert after x or to delete x, we need to know the access path in order to update the auxiliary data. Similarly, splitting at x requires knowledge of the access path, as we will see. For our purposes it is simpler to search as described above.

Performing insertions and deletions in finger trees is almost the same as in regular red-black trees. Consider inserting after or before an external node x or deleting x. It is again necessary to know the access path to x. If x and its parent are both non-path nodes, then the restructuring is the same as before. Otherwise the reversed pointers have to be taken into account. Both of these operations potentially invalidate the data of all nodes on the access path to x. Updating is done as in regular trees by traversing the access path backwards and recomputing the auxiliary data for each node as it is accessed.

Updating the auxiliary data requires time proportional to the length of the access path, which is $O(1 + \log(\min\{d, n - d\}))$ if x is the d^{th} node in an n-node tree. Constant time is required for the restructuring in the worst case and constant amortized time for the rebalancing. This implies that insertion at or deletion of the d^{th} node requires $O(1 + \log(\min\{d, n - d\}))$ amortized time.

Concatenating is a little more complicated than in the regular case. Let T_1 and T_2 be two finger trees with roots x_1 and x_2, and ranks r_1 and r_2. Assume that $r_1 \leq r_2$ and that T_1 is to be concatenated to the left of T_2. The other cases are analogous.

If $r_1 = r_2$, walk up the left paths of T_1 and T_2 until reaching the roots. Create new node i having x_1 and x_2 as children and make x_1 and x_2 point to i. Make i red if both x_1 and x_2 are black and red otherwise.

If $r_1 < r_2$, then the procedure is as follows. Delete the pointers from x_1 to its children. Walk up the left path of T_1 to the root, x_1. If x_1 is red, make it black. Let r_1' denote the resulting rank. Walk up from the left finger of T_2 until reaching a black node u such that $rank(u) = r_1'$. Let v be the left child of u (the successor on the left path). Create new node i. Make v and u the left and right children of i, respectively and make i the left child of x_1 (see Figure 2.11). Make i red and rebalance using the transformations given

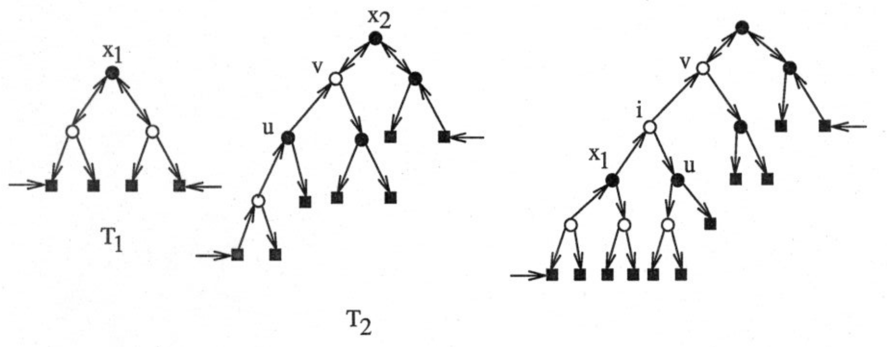


Figure 2.11: *Concatenating finger trees. Shown are the trees to be concatenated, T_1 and T_2, and the resulting tree T before rebalancing.*

in Figure 2.2 if this causes a violation of the red constraint with v (i.e., if v is red).

The tree formed is now a valid red-black tree but is not a proper finger tree since the pointers on the right path of T_1 and on the path from the old left finger of T_2 to u (or x_2 if the ranks are equal) point up instead of down. To fix this, traverse both these paths reversing the pointers and recomputing auxiliary data values. The left finger for the resulting tree is the left finger of T_1 and the right finger is the right finger of T_2. Finally, recompute the auxiliary data for nodes on the path from the left finger to i.

The time required for finding u and updating pointers and data is $O(r_1)$, which is $O(1 + \log d)$ if the tree had d external nodes, and the time required for rebalancing is $O(1)$ in the amortized sense.

Splitting is more complicated still. Let T be a finger tree and let x be the d^{th} external node of T. Splitting at x yields finger trees T_1 containing the external nodes preceding x and T_2 containing the nodes succeeding x. We assume that x is in the left subtree; the other case is symmetric. Let u be the left path node whose non-path subtree contains x.

Let T_u be the subtree rooted at u, let v be the successor of u on the left path and let u' be the non-path child of v (see Figure 2.12). Delete T_u from T leaving a fragment, T', of the original tree. Make T_u a regular red-black

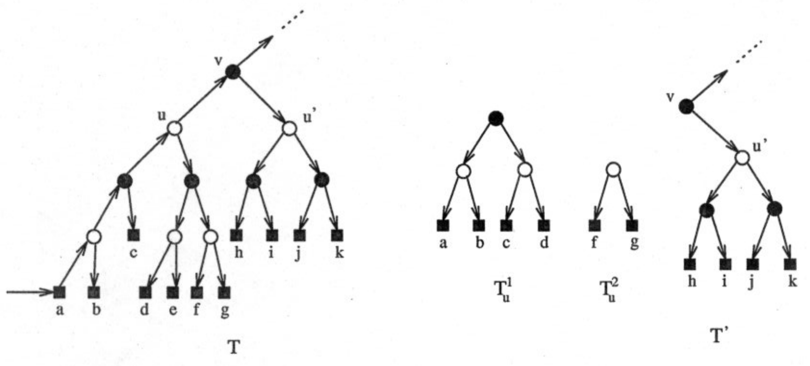


Figure 2.12: *Splitting finger tree T at node e. Shown at right are the trees formed during the splitting process; the trees T_u^1 and T_u^2 left after splitting T_u at e and the fragment of T left after deleting T_u.*

tree by reversing the pointers along the left path and recomputing auxiliary data for nodes on this path. Use the red-black tree splitting operation to split T_u at x into T_u^1 and T_u^2. These trees are regular red-black trees; make them finger trees by reversing the pointers on the left and right paths and again recomputing the auxiliary data of these nodes. The desired tree T_1 is T_u^1.

To form T_2 we need to combine the nodes of T_u^2 and T'. Make T' a valid finger tree as follows. Reverse the pointers along the path from u' to its leftmost descendant, which is now pointed to by a left finger. Replace v by u' and its subtree. If v was black this causes a violation of the black constraint (u' is short). If so, rebalance from u' as for a deletion, updating auxiliary data as required. The auxiliary data along the path from the left finger to u' is now invalid (since the pointers have been reversed) and must be recomputed. Now use the finger tree concatenation operation to concatenate T' and T_u^2. The result is T_2.

Note that u, u', and v all have heights proportional to $O(1+\log d)$ which is also $O(1+\log(\min\{d, n-d\}))$. The work involved in finding u and deleting and creating fingers in the various trees requires time proportional to the height of u. Splitting T_u requires $O(1+\log d)$ time. Rebalancing after deleting v requires $O(1)$ amortized time. Concatenating T' and T_u' requires $O(1+\log d)$

time also. Thus the amortized time for splitting T at the d^{th} node is $O(1 + \log(\min\{d, n-d\}))$.

To restore the balance condition after a tree restructuring operation (eg, insertion, deletion, or concatenation), as for regular red-black trees, we use the rebalancing transformations of Figures 2.2 and 2.3. These operations are applied to the path from the violation up to the root, ignoring the reversed pointers. In order to traverse this path without storing parent pointers, the portion (if any) that is not on the left or right path must be stored. Note that in a regular red-black tree without parent pointers the entire path from the root must be stored in a similar manner.

Auxiliary data must again be updated after rotations and restructuring operations. For a node x, the auxiliary value stored at x becomes invalid only if the set of nodes reachable from x changes. First consider how rotations affect the data.

For rotations involving only non-path nodes, the effect and subsequent recomputation is the same as in regular red-black trees; if path nodes are involved the reversed pointers must be taken into account. A rotation at the root is more complicated. Such an operation changes the set of reachable nodes for all nodes on both the right and left paths. To restore these auxiliary values requires traversing both paths top-down, recomputing the data at each node. This requires $O(\log n)$ time in an n-node tree. However, the fact that the rebalancing procedure reached the root implies that $O(\log n)$ work was already done, either in rebalancing (in an insertion or deletion) or in restructuring and rebalancing (in a concatenation). This implies that rebalancing still requires only $O(1)$ time in the amortized sense.

Finger search trees can also support add value operations. Again, the *maxkey* values must be stored in difference form as described in Section 2.2 for red-black trees. Maintaining this data with the reversed pointers, in particular, recomputing auxiliary data when reversing the pointers along a path, is somewhat involved. We describe here a simpler, but slightly less powerful, type of finger search tree in which data is stored only with the nodes not on

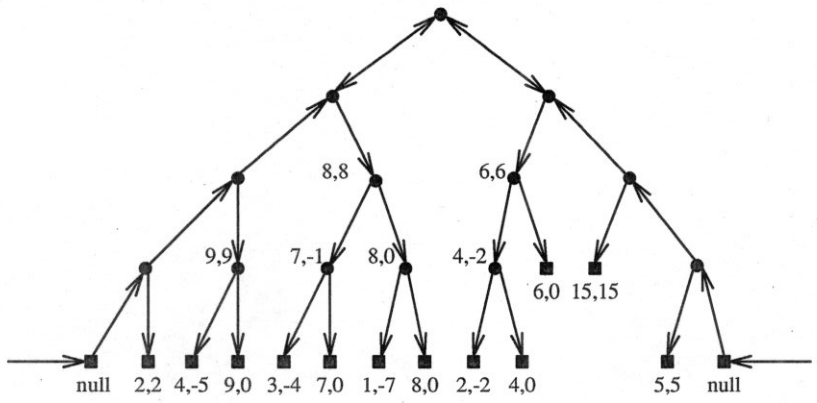


Figure 2.13: *An example of a finger tree with no path data. Shown with each node x are the values* maxkey(x) *and* Δmaxkey(x) *(*maxkey *values are not stored).*

the left or right paths. This type of finger tree is used for the application described in Chapter 4.

In order to represent a list having n items, create a finger search tree having $n+2$ external nodes; the leftmost and rightmost nodes represent null items. The remaining external nodes are associated with list items in order, as before. For both null external nodes set *key1* to minus infinity and *key2* to 0. Store auxiliary data only with the non-path nodes. For each non-path node y store $totkey(y)$ and $\Delta maxkey(y)$. The latter value is $maxkey(y) - maxkey(p(y))$ if $p(y)$ is not a path node and is $maxkey(y)$ otherwise. An example illustrating the data is shown in Figure 2.13.

Notice that the non-path subtrees are regular red-black trees with *maxkey* stored in difference form.

Using this data structure we can perform the same operations. With respect to searching it is similar to a finger tree with only a left finger. Accessing an item requires $O(1+\log d)$ time if the item found is the d^{th}. Again, insertion, deletion, and splitting require $O(1+\log(\min\{d, n-d\}))$ time at the d^{th} external node in an n-node tree. Concatenation requires time proportional to the rank of the smaller tree. Adding a constant to the values of all external nodes (or to the values of a consecutive subsection of the external nodes) requires

$O(\log n)$ time in an n-node tree.

Implementation of insertion, deletion, concatenation, and splitting is the almost exactly the same as for regular finger trees. The only difference lies in the data updating and searching procedures.

Searching is slower since it is no longer possible to detect in constant time whether the desired item is reachable from the left finger. To access an item, first search up the left path until finding a subtree containing the item or until reaching the root. If no subtree is found, search down the right path (which first requires walking up the entire path) until reaching a subtree containing the desired item or until reaching the right finger. If we reach the right finger, then the item does not exist. Otherwise, search for the item in the found subtree as in a regular red-black tree.

In the search for the leftmost item with key at least k, we find the leftmost path node whose non-path child u has $\Delta maxkey \geq k$ and then search down within u's subtree as described in Section 2.2.

To find the leftmost item with cumulative value at least k we again maintain the value *maxcumkey*. To update this value when going from a node u on the left path to its left child, v (if v is not the root), add $totkey(right(v))$. If $maxcumkey(u) < k$ for all nodes u on the left path, then the search continues from u, the right child of the root, starting with *maxcumkey* incremented by $totkey(left(u))$. To update *maxcumkey* when going from node on the right path to its predecessor v, add $totkey(left(v))$. The leftmost node with $maxcumkey \geq k$ is contained in the non-path subtree of the leftmost path node having $maxcumkey \geq k$. The procedure for searching within this subtree was described in Section 2.2.

If the desired item is not found, then both search procedures described above will take $O(\log n)$ time in an n-node tree. If the desired item is the d^{th}, then the search will require $O(1 + \log d)$ time. Lemma 2.2 implies that the length of the access path for the d^{th} item is $O(1 + \log d)$ if it is in the left subtree. If the item is in the right subtree the algorithm requires $O(\log n)$ time, but using the reasoning used in Lemma 2.2 we can show that this is also

$O(1 + \log d)$ in this case.

To add a constant k to the *key1* values of all external nodes, add k to the Δ*maxkey* values of all non-path children of path nodes. Given two external nodes x and y and their access paths, we can also add k to the values of all external nodes between x and y. Both these operations require $O(\log n)$ time on an n-node tree. The first operation is slower than for the Tarjan and Van Wyk finger trees in which adding a constant to all nodes of the tree requires constant time. However, by storing a *cost offset* value with the root of each modified finger tree, we can perform *add value* operations in $O(1)$ amortized time. Adding a constant k to this value increases by k the costs associated with all leaves [43]. Such a scheme entails more work in maintaining data after a concatenation or a split operation, but this adds only a constant factor to the running time.

It is easier to maintain the auxiliary data in this modified finger tree. Since no data is associated with the path nodes, the reversed pointers do not need to be taken into account. In contrast, with the finger trees described previously, rotating at the root does not require updating data of nodes on the left and right paths.

Since the subtree containing a particular external node is a regular red-black tree, updating data after an insertion or deletion is handled exactly as described in Section 2.2. Creating or destroying a finger also invalidates data. Updating data after such an operation is straightforward. If we wish to add a finger by reversing all the pointers along the left or right path of a tree, first traverse the path once to compute *maxkey* values for the path nodes and their siblings. For each sibling replace Δ*maxkey* by *maxkey*. To destroy a finger again traverse the path once computing and storing *maxkey* values (with respect to downward pointers) in place of Δ*maxkey* values for the nodes on the path and their siblings. Then traverse the path once more to compute new Δ*maxkey* values for all these nodes. Store *maxkey* with the root. In addition to reversing the pointers and updating data, when deleting a finger we must also delete the corresponding special external node and when adding a finger we must insert a null external node.

2.4 Remarks

Finger trees have a reputation for being complicated and impractical, hard to understand and hard to implement. If one considers only the operations insert, delete, and access, then the finger tree implementations given here are not significantly more complex than those of regular balanced trees. Concatenating and splitting are more complex and allowing add value operations adds still more details.

In this variant the path nodes serve only to link together a bunch of red-black subtrees and to impose regulations on their sizes so that the search time is optimal. A natural question is whether this observation can be generalized to create a simpler data structure with the same kind of performance. It is not necessary to think of the set of subtrees plus the linking paths as a tree itself, especially since the linking nodes already have to be treated differently from the subtree nodes. However, treating the whole structure as a tree allows us to use standard tree rebalancing operations (i.e., rotations) in order to maintain the balance condition in the entire structure. Splay trees, a form of self-adjusting search tree introduced by Sleator and Tarjan [44], are more elegant and easier to implement. It is not known if they perform as well, however.

3. Finding a Balanced Decomposition of a Tree

3.1 Introduction

In this chapter we describe how to find a balanced decomposition of a tree in linear time. The balanced decomposition can be defined for both *rooted* and *unrooted* (or *free*) trees. A free tree is a connected graph that has no cycles. A rooted tree [36] is a free tree with one vertex designated as the root.

A *decomposition* of a tree, either rooted or free, is formed by breaking the tree into two or more pieces and then repeatedly decomposing each piece until it can no longer be split. Breaking the tree can be accomplished by removing any non-empty set of vertices and/or edges. For our purposes we consider only decompositions in which each splitting operation consists of removing a single edge, thereby breaking the tree into two pieces. In this case the decomposition process ends when each piece is a single-node tree.

This process can be be represented by a rooted tree called the *decomposition tree*. The decomposition tree (here) is binary; each node of the decomposition tree represents a component resulting at some point in the decomposition process. Such a component is called a *fragment* of the input tree. An example is given in Figure 3.1. For a tree T, the *size* is the number of nodes in T and is denoted $|T|$.

Let T be a tree we wish to decompose. A decomposition is called *balanced* if there exists a constant c not dependent on the size of the tree such that $0 < c \leq 1/2$ and for each fragment F formed during the decomposition of T, the two fragments left after splitting F both have size at least $c|F|$. The decomposition process terminates after $O(\log n)$ stages in a balanced decomposition. This implies that the decomposition tree has depth $O(\log n)$.

In [26], Guibas, Leven, Hershberger, Sharir, and Tarjan give an algorithm for finding a balanced decomposition of a binary tree in linear time, which is optimal. As used here, a *k-ary* tree is a rooted tree in which each node has at most k children. Here we give an algorithm for finding a balanced

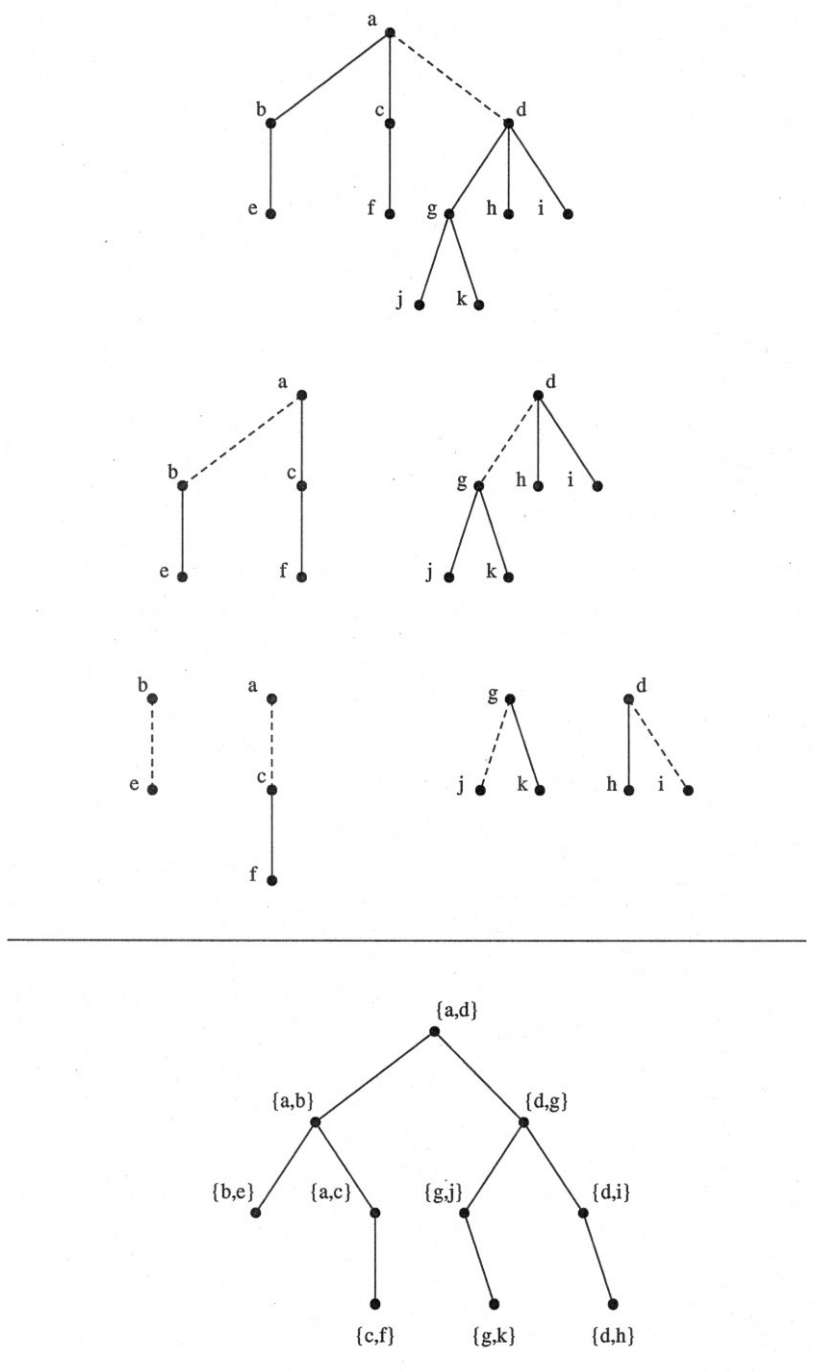


Figure 3.1: *A tree, a partial decomposition of the tree, and the corresponding decomposition tree.*

decomposition of a k-ary tree in linear time, if k is a constant independent of tree size. Since an unrooted tree with maximum degree $k+1$ can be converted into a k-ary tree by choosing any node with degree 1 to be the root, this method will also work for unrooted trees. The method is quite different from that of Guibas, et. al.. Cole and Vishkin [14] have given a parallel algorithm for a similar tree decomposition problem.

This chapter contains three additional sections. In Section 3.2 we prove the existence of a balanced decomposition for a k-ary tree and show how to find one in $O(n \log n)$ time. In Section 3.3 we present a linear-time algorithm. Section 3.4 contains concluding remarks.

3.2 Existence

Every k-ary n-node tree has a balanced decomposition which can be found in $O(n \log n)$ time. To show that a balanced decomposition exists we need to show that there exists a constant $0 < c \leq 1/2$ such that every k-ary n-node tree has an edge whose removal leaves two pieces each of size at least cn. For this proof and the remainder of the chapter we assume that the input tree is rooted.

To denote an edge from node u to node v in a tree we use the notation $\{u, v\}$. For a node u in tree T, $nd(u)$ denotes the number of descendants of u in T. If u is not a leaf, the *largest child* of u is the child of u with the largest number of descendants. Observe that for a node u with at most k children, the largest child of u has at least $\lceil \frac{nd(u)-1}{k} \rceil$ descendants.

Lemma 3.1 *If T is a k-ary n-node tree with $k \geq 2$ and $n \geq 4$, then there exists an edge of T whose removal leaves two pieces both having size at least $\lceil \frac{n}{2k} \rceil$.*

Proof: Consider the following procedure. Walk down from the root of the tree, taking at each node x the edge from x to its largest child, until reaching a node v with $nd(v) < \lceil (1 - \frac{1}{2k})n \rceil$. Let u be the parent of v. We can show that the edge $\{u, v\}$ satisfies the claim. Since $nd(u) \geq \lceil (1 - \frac{1}{2k})n \rceil$ and v is the largest child of u, $nd(v) \geq \lceil \frac{nd(u)-1}{k} \rceil$, which is at least $\lceil \frac{n}{2k} \rceil$ for $n \geq \frac{2k}{k-1}$, which is less than or equal to 4 for $k \geq 2$. We also need to show that $n - nd(v) \geq \lceil \frac{n}{2k} \rceil$. This follows from the fact that $nd(v)$ is an integer strictly less than $\lceil (1 - \frac{1}{2k})n \rceil$. ∎

This lemma implies that there exists a balanced decomposition of T with constant $c = \frac{1}{2k}$. The depth of the decomposition is dependent on the constant c. The depth is less than or equal to $d \log n$ where d is the smallest integer such that $(1-c)^d n \leq 1$. This implies that $d \leq \frac{1}{\log \frac{1}{1-c}}$. This discussion proves the following theorem:

Theorem 3.1 *For an n-node k-ary tree T there exists a balanced decomposition of T with depth at most $\frac{1}{\log \frac{1}{1-\frac{1}{2k}}} \log n$.*

For example, for an n-node binary tree ($k = 2$), the depth is at most $\frac{1}{\log 4/3} \log n$, which is approximately $2.4 \log n$.

In the splitting procedure given in Lemma 3.1, the threshold $(1 - \frac{1}{2k})$ can be replaced by $1 - \frac{1}{k+2}$, yielding a split in which both pieces have size at least $\frac{n}{k+2}$. This method will not always yield a decomposition with splitting constant less than or equal to $\frac{1}{k+1}$ since there exists a family of trees in which removal of any edge yields a piece of size less than $\frac{n}{k+1}$, where n is the size of the tree. For example, consider the family of k-ary trees with parameter d such that each tree consists of a central node connected to the roots of $k+1$ complete k-ary trees with d levels. This tree will have $\frac{(k^d-1)(k+1)}{k-1} + 1$ nodes. Removal of any edge yields a piece of size at most $\frac{k^d-1}{k-1}$, which is less than $n/(k+1)$. Figure 3.2 shows the structure of such a tree when $k = 2$.

Note that a tree in which the maximum degree is not a constant may not have a balanced decomposition. For example, the "star" tree consisting of n nodes such that one node is adjacent to all the others (shown in Figure 3.3) has no decomposition with depth less than $n-1$. (Note that splitting by removing a vertex instead of an edge can yield a decomposition with constant depth.)

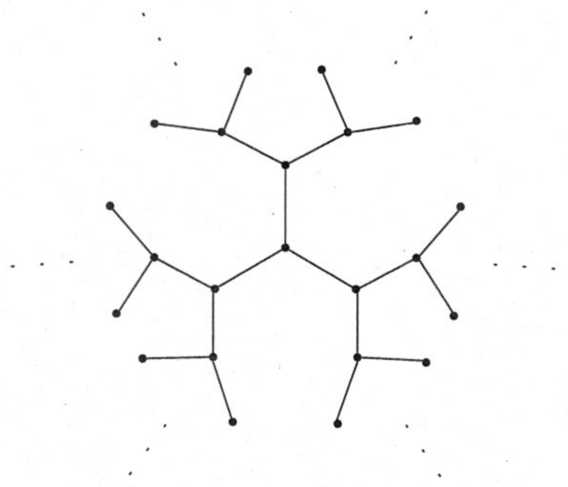


Figure 3.2: *This figure shows the structure of a family of binary trees having the following property. If T is a tree in this family and has d levels (and hence $n = 3(2^d - 1) + 1$ nodes), then removal of any edge in T leaves a piece having size less than $\frac{n}{3}$.*

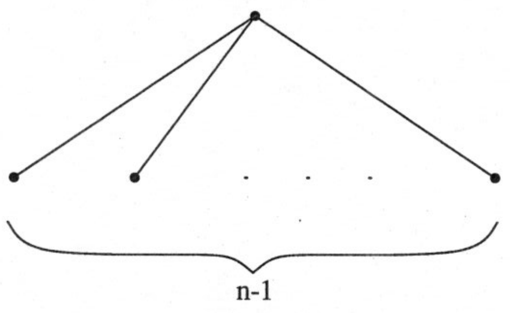


Figure 3.3: *A tree which has no balanced decomposition. The decomposition tree is the same - a chain of $n - 1$ nodes - for all possible decompositions (all possible orderings of the edges).*

Let T be an n-node k-ary tree. An edge e in T is called a *splitting edge* (for T) if and only if removal of e yields two pieces both having size at least $\lceil \frac{n}{2k} \rceil$. The proof of Lemma 3.1 gives a procedure for finding the splitting edge of T in linear time, which leads to the following algorithm for finding a balanced decomposition.

Algorithm 1 *Compute $nd(u)$ values for all nodes u in T, find the splitting edge for T, and recursively decompose the two pieces formed by removing this edge.*

Lemma 3.2 *Algorithm 1 runs in $O(n \log n)$ time.*

Proof: The running time of this algorithm satisfies the following recurrence:

$$T(n) = \max_{\substack{n_1, n_2 \geq cn \\ n_1 + n_2 = n}} \{T(n_1) + T(n_2) + O(n)\}, \tag{3.1}$$

for $c = \frac{1}{2k}$. It is well-known that this has solution bounded by $O(n \log n)$. ∎

Algorithm 1 gives a simple method for finding a balanced decomposition which splits an n-node tree in $O(n)$ time. This method relies on values, the nd values, which cannot be easily recomputed after splitting. In order to get a faster decomposition algorithm, we need a procedure for finding the splitting edge that can be executed recursively. That is, the data used to find the splitting edge should either be easy to recompute after splitting or should need no recomputation at all.

3.3 A linear-time algorithm

Our algorithm has the same form as Algorithm 1. After preprocessing the input tree T, we execute the same recursive step: find the splitting edge, remove it, and recurse on the two fragments formed. Our algorithm cannot match the splitting constant of the previous section. There we defined a splitting edge to be one whose removal left pieces both having size at least $\lceil \frac{n}{2k} \rceil$; here this quantity must be decreased to the *floor* of $\frac{n}{2k}$ rather than the ceiling.

The basis of our algorithm is a procedure for finding a splitting edge of an n-node tree in $O(\log n)$ time with linear preprocessing. Thus the running time of the algorithm satisfies the recurrence:

$$T(n) = \max_{\substack{n_1, n_2 \geq cn \\ n_1 + n_2 = n}} \{T(n_1) + T(n_2) + O(\log n)\} \tag{3.2}$$

where c is the splitting constant; $0 < c \leq \frac{1}{2}$. The solution of this recurrence is $O(n)$. This can be proved by induction.

Proof: Assume that $T(n) \leq an - 2\log n$ for a constant a, which will be derived. We will assume that the log term in the running time recurrence has a constant of 1; this affects only the constant of the result. It suffices to show for all n_1 and n_2 such that $n_1 \geq cn$, $n_2 \geq cn$, and $n_1 + n_2 = n$, that

$$an_1 - 2\log n_1 + an_2 - 2\log n_2 + \log n \leq an - 2\log n.$$

This is true for all n such that $3\log n \leq 2\log n_1 + 2\log n_2$. Since n_1 and n_2 must both be greater than or equal to cn, the right hand side is at least $4\log cn$, which is at least $3\log n$ for $n \geq c^{-4}$.

To complete the inductive proof it is necessary to show that $T(n)$ is at most $an - 2\log n$ for $n < c^{-4}$. It can be shown for all n that $T(n) \leq a'n\log n$ for some constant $a' > 0$. This implies that the bound on $T(n)$ holds if there exists a constant $a > 0$ such that:

$$(a' + 2)n\log n \leq an.$$

For the values we are concerned with, $\log n \leq 4\log(1/c)$. Thus choosing $a = 4(a' + 2)\log(1/c)$ gives the desired result. ∎

In order to find the splitting edge in logarithmic time we use a *preorder numbering* of the tree.

A *preorder numbering* [1] of a rooted tree T is computed by traversing the tree depth-first, numbering each node as it is first accessed. A depth-first traversal of a rooted tree T first accesses the root of T and then recursively traverses each subtree. Computing the preorder numbering of an n-node tree

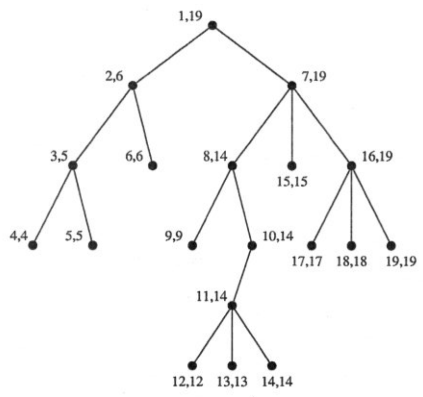


Figure 3.4: *A rooted tree with* preorder *and* range *values shown with each node.*

can be done in linear time [48]. For node u we use $pre(u)$ to denote the preorder number. In addition to $pre(u)$ we need the value $range(u)$. For node u, $range(u)$ is the largest preorder number of a node in u's subtree. Figure 3.4 illustrates these values.

Preorder and range values have the following properties. The descendants of a node u are all numbered consecutively, from $pre(u)$ to $range(u)$. If u and v are two nodes in a tree T such that u is an ancestor of v, then $pre(u) \leq pre(v)$ and $range(u) \geq range(v)$. Further, $pre(u) \leq pre(v)$ if and only if u is on the path from v to the root (ie, is an ancestor of v) or is contained in a subtree to the left of the path from v to the root.

The *nearest common ancestor* of two nodes u and v is the deepest node that is an ancestor of both u and v.

The following observation is the basis of our algorithm.

Lemma 3.3 *Let T be a k-ary tree, $k \geq 2$, with n nodes. If x and y are the nodes with preorder numbers $\lceil \frac{n}{4} \rceil$ and $\lceil \frac{3n}{4} \rceil$ respectively, z is the nearest common ancestor of x and y, and u is the largest child of z, then u has between $\lfloor \frac{n}{2k} \rfloor$ and $\lfloor \frac{3n}{4} \rfloor$ descendants.*

Proof: Let $S_1 = \{v \in T | pre(v) \leq \lceil \frac{n}{4} \rceil\}$ and let $S_2 = \{v \in T | pre(v) \geq \lceil \frac{3n}{4} \rceil\}$. The subtree of z contains all nodes with preorder numbers between $pre(x)$

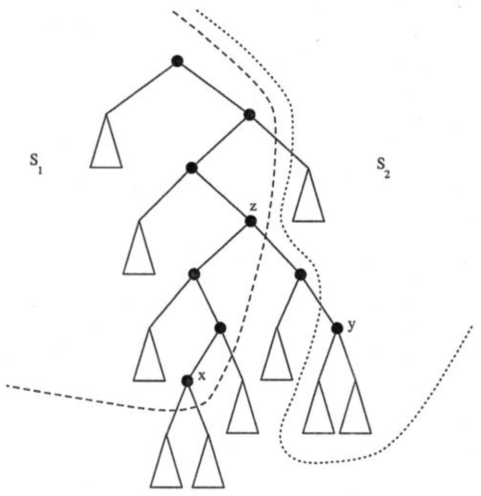


Figure 3.5: *The triangles indicate subtrees. The subtrees and nodes to the left of the dashed line are contained in the set S_1 and those to the right of the dotted line are contained in S_2.*

and $pre(y)$, that is, all nodes not in either S_1 or S_2 (Figure 3.5 illustrates these definitions). This subtree also contains two distinct nodes, z and y, that are contained in either S_1 or S_2. Since $S_1 \cap S_2 = \emptyset$, this implies that $nd(z) \geq n - |S_1| - |S_2| + 2$, which is at least $\lfloor \frac{n}{2} \rfloor + 1$. The largest child u of z has at least $\lceil \frac{nd(z)-1}{k} \rceil$ descendants, which implies that $nd(u) \geq \lfloor \frac{n}{2k} \rfloor$.

Finally, we can see that $nd(u) \leq \lfloor \frac{3n}{4} \rfloor$ by observing the following. If the subtree of a node v contains nodes from both S_1 and S_2, then it must be an ancestor of both x and y and hence an ancestor of z. This implies that a child of z may not contain nodes from both sets, which in turn implies that $nd(u) \leq \max\{n - |S_1|, n - |S_2|\}$, which is at most $\lfloor \frac{3n}{4} \rfloor$. ∎

Corollary 3.2 *Edge (z, u) defined in Lemma 3.3 is a splitting edge for T.*

Proof: A splitting edge is one whose removal leaves two pieces both having size at least $\lfloor \frac{n}{2k} \rfloor$. Removal of (z, u) leaves two trees, the subtree rooted at u and the tree containing the remaining nodes of T. By Lemma 3.3, the former has at least $\lfloor \frac{n}{2k} \rfloor$ nodes and the latter has at least $\lceil \frac{n}{4} \rceil$. This implies that both fragments have size at least $\lfloor \frac{n}{2k} \rfloor$ for $k \geq 2$. ∎

Replacing $\lceil \frac{n}{4} \rceil$ and $\lceil \frac{3n}{4} \rceil$ in the lemma given above by $\lceil \frac{n}{k+2} \rceil$ and $\lceil \frac{n(k+1)}{k+2} \rceil$ yields a better split; removal of the edge found by this variation yields two pieces each having size at least $\lfloor \frac{n}{k+2} \rfloor$.

Lemma 2.2 suggests the following method for splitting an n-node tree T:

(i) find the nodes x and y with preorder numbers $\lceil \frac{n}{4} \rceil$ and $\lceil \frac{3n}{4} \rceil$, respectively,

(ii) find z, the nearest common ancestor of x and y in T,

(iii) find the largest descendant u of z in T, and

(iv) remove edge $\{z, u\}$.

This can easily be executed in linear time. To improve the running time from $O(n)$ to $O(\log n)$, we represent each fragment as a list, recasting the above procedure in terms of some simple list operations. Later we show how to represent the list as a balanced binary tree so that the list operations and, hence, the splitting procedure, can be executed in $O(\log n)$ time. In order to transform the splitting problem into a list problem, each tree is represented by a list called its *preorder list.* Such a representation has been used in many previous settings; for example, see the paper by Dial, Glover, Karney, and Klingman [16].

For the input tree T, the preorder list L_T contains the nodes of T in preorder order; stored with each node u is the *range* value for u in T. For a fragment F of T, the preorder list L_F contains the nodes of F in preorder order, but with each node u is stored the *range* value of u with respect to the input tree T. This value is denoted by $range_T(u)$. Figure 3.6 gives an example of a tree, a fragment, and its corresponding preorder list.

The preorder list gives a good deal of information about the tree structure, enough so that the operations required for finding the splitting edge can be executed using simple list queries. First we describe how to execute these operations - finding the node with a given preorder number, finding the nearest common ancestor of two nodes, and computing the number of descendants of a node - in the input tree T.

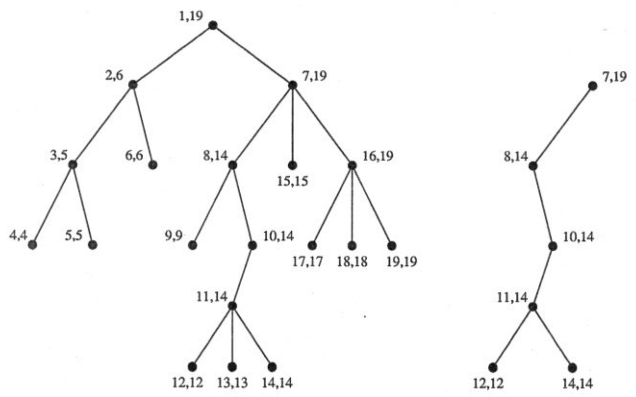


Figure 3.6: *Shown are a tree T and a fragment of T; shown with each node u is the value* $\mathrm{range}_T(u)$. *The preorder list L_F for fragment F is* $(7,19)(8,14)(10,14)(11,14)(12,14)(14,14)$.

Since nodes are ordered in the preorder list by increasing preorder number, finding the node with a given preorder number is equivalent to *searching on position* in L_T. Given integer j, the node with preorder number j is the j^{th} item in L_T.

To find the nearest common ancestor we need to use the following property: if x and y are two nodes in T with $pre(x) \leq pre(y)$, then the nearest common ancestor of x and y is the common ancestor with highest preorder number; a node z is a common ancestor, an ancestor of both x and y, if and only if $pre(z) \leq pre(x)$ and $range(z) \geq range(y)$. Thus, in L_T, the nearest common ancestor of x and y is the rightmost node to the left of x whose *range* value is at least $range(y)$. We call this query *search to the left*: given a list with associated keys, a key r and a node x, find the rightmost node to the left of x with key at least r.

The number of descendants of a node in T can be determined without searching. Since the descendants of a node are all numbered consecutively, from $pre(u)$ to $range(u)$, $nd(u) = range(u) - pre(u) + 1$.

For a fragment of T the latter two operations are more involved because the values stored are the *range* values with respect to T rather than those computed for the fragment itself.

Finding the nearest common ancestor of two nodes in a fragment is the same as in the input tree. For a node u in F let $pre_F(u)$ and $pre_T(u)$ denote the preorder numbers of u with respect to F and T respectively. Consider nodes x and y in F such that $pre_F(x) \leq pre_F(y)$. Any node z which is an ancestor of both x and y in F is also an ancestor of both in T and thus must satisfy: $pre_T(z) \leq pre_T(x)$ and $range_T(z) \geq range_T(y)$. The first condition is equivalent to $pre_F(z) \leq pre_F(x)$. The nearest common ancestor in F is the common ancestor in F with highest preorder number. This implies that the nearest common ancestor of x and y is the rightmost node to the left of x in L_F having $range_T \geq range_T(y)$.

To compute the number of descendants of a node in a fragment we again use the fact that the descendants of a node all occur consecutively in the preorder list. Let u be a node in F. The *rightmost descendant* of u is the descendant of u in F with highest preorder number. Let u' be the rightmost descendant of u and let $pos(x)$ denote the position of node x in F. Then $nd_F(u) = pos(u') - pos(u) + 1$.

To find u', we use the following facts. For all nodes v between u and u' in L_F, $range_T(v) \leq range_T(u)$. For nodes w following u', $range_T(w) > range_T(u)$. This implies that u' is the predecessor of the leftmost node to the right of u that has $range_T > range_T(u)$. Finding u' thus requires the list operations *search to the right* and *predecessor*. Determining $nd_F(u)$ also requires the list operation *position*, which returns the position of a node in its list.

In order to remove the splitting edge so that the splitting procedure can be recursively applied to the two pieces, we need to compute the preorder lists of the resulting pieces. Let F be a fragment with splitting edge $\{z, u\}$ where z is the parent of u. Removal of this edge leaves two pieces, the subtree rooted at u and the tree containing the remaining nodes. We call these new fragments A and B, respectively. The preorder list, L_A, for A contains all nodes between u and u', where u' is the rightmost descendant of u; L_B contains the remaining nodes in the same order as in L_F. To construct L_A and L_B (given u'), split L_F before u and after u'. The middle piece is L_A; concatenating the first and third pieces yields L_B.

By representing each fragment as a list, we have converted the splitting edge problem into the following list problem. Given an ordered list of items in which each item has an associated real-valued key, we wish to represent the list so as to be able to execute the following operations quickly:

search on position: given integer j, $1 \leq j \leq n$, return the j^{th} item,

position: given item x, return its position,

search to the left: given value r and item x, return the rightmost item to the left of x having key at least r,

search to the right: given value r and item x, return the leftmost item to the right of x having key at least r,

predecessor : given node x, return its predecessor in the list,

split after: given list L and item x, return two lists, one containing the items up to and including x and the other containing the remaining items,

split before : given list L and item x, return two lists, one containing the items up to but not including x and the other containing the remaining items, and

concatenate: given two such lists L_1 and L_2, concatenate L_2 to the back of L_1,

The keys associated with items in our list are the $range_T$ values.

We can now specify the recursive step of the algorithm in terms of the list operations given above. Let F be an n-node fragment of the input tree T. The following procedure splits F, producing lists L_A and L_B representing the new fragments A and B:

```
$x = search\ on\ position(L_F, \lceil \frac{n}{4} \rceil)$
$y = search\ on\ position(L_F, \lceil \frac{3n}{4} \rceil)$
```

```
$z$ = $search\ to\ the\ left(L_F,\ x,\ range_T(y))$
For each child $v$ of $z$ in $F$, determine $nd_F(v)$ as follows:
    $r$ = $search\ to\ the\ right(L_F,\ v,\ range_T(v))$
    $v'$ = $predecessor(r)$
    $nd_F(v)$ = $position(v')$ - $position(v) + 1$
Let $u$ be the child of $z$ with largest $nd_F$ value
Let $u'$ be the rightmost descendant of $u$
/* Split $F$ as follows: */
remove $\{z, u\}$ from $F$
let $A$ be the subtree rooted at $u$
let $B$ be portion of $F$ left after removing $A$
/* Split $L_F$ as follows: */
$split\ before(L_F,\ u,\ L_1,\ L_1')$
$split\ after(L_1',\ u',\ L_2,\ L_3)$
$L_A = L_2$
$L_B = concatenate(L_1,\ L_3)$
```

The procedure for splitting an n-node fragment runs in $O(g(n))$ time where $g(n)$ is the maximum time required for executing the list operations given above on an n-item list. With a simple linked-list representation, $g(n) = O(n)$. This time can be improved to $O(\log n)$ by representing the list by a balanced tree.

There several well-known types of balanced tree data structures which can support the operations mentioned above in logarithmic time in the worst-case. Two of these are (a, b) [40] and red-black [28] trees. In addition, the splay tree data structure introduced by Sleator and Tarjan [44] can be used to execute these operations in amortized logarithmic time. For this algorithm, any of the above data structures can be used.

Red-black trees [28] are described in detail in Chapter 2 and we develop that method here. The non-leaf nodes are called *internal* nodes and the leaf nodes are called *external*. An n-item list L is represented by a red-black tree

T_L with n external nodes by associating each item of L with an external node of the tree so that left-to-right order in L corresponds to left-to-right order among the external nodes of T_L.

In order to perform the desired query operations, the following two auxiliary data values are stored with each internal node u:

$$\begin{aligned} &\mathit{size}(u) = |\{x|x \text{ is an external descendant of } u\}| \text{ and} \\ &\mathit{maxkey}(u) = \max\{\mathit{key}(x)|x \text{ is an external descendant of } u\}, \end{aligned}$$

where $\mathit{key}(x)$ is $\mathit{range}_T(x)$.

Given a list L, the tree T_L representing L can be constructed in linear time. Constructing the tree structure is easy. The auxiliary data values can be computed by traversing the tree bottom-up.

For the implementation described in Chapter 2, *maxkey* and *cumkey* are stored. The latter value is equivalent to *size* if each item of the list is thought of as having a second key of value 1. Searching on position is thus equivalent to searching on *cumkey* which was described previously. Given below are descriptions of the other query operations.

To determine the position of a node x, walk up the path from x to the root summing the values $\mathit{size}(u)$ for the nodes u such that u is a sibling of a node on the path and is also a left child.

Performing *search to the left* entails finding the rightmost external node with key at least k such that the node is also to the left of a given node x (given the path from x to the root). To do this, walk up from x until reaching the first node with left child u such that u is not on the path to x and $\mathit{maxkey}(u) \geq k$. Then search down in u's subtree for the rightmost node with $\mathit{key} \geq k$. This search requires time proportional to the length of the path from x to the desired node which is $O(\log n)$. Performing *search to the right* is analogous.

To return the predecessor of an external node x in logarithmic time, walk up from x until reaching the first node that is a right child then walk down from its sibling by following right pointers until reaching an external node.

All these query operations run in time proportional to the depth of the tree, which is $O(\log n)$.

Splitting and concatenating were also described in Chapter 2 along with the procedures for updating any auxiliary data invalidated by the tree restructuring. Both run in $O(\log n)$ time on an n-node tree. As described, the operation *split*(u) yields two lists, one containing the items preceding u and one containing the items succeeding u. It is easy to modify this operation so that u is included at the end of the former list or at the beginning of the latter as desired. This does not affect the running time.

By representing the preorder list as a red-black tree, the list operations necessary for computing the splitting edge can be executed in $O(\log n)$ time if the list has n items. This implies that an n-node fragment can be split in $O(\log n)$ time. To preprocess the input tree T, compute the preorder and range numbers for T, construct the preorder list, and finally construct the red-black tree representing the preorder list. This can be done in linear time. Thus the running time of the splitting algorithm satisfies Recurrence 3.2. This constitutes a proof of the following theorem.

Theorem 3.3 *The algorithm given above for finding a balanced decomposition of an n-node k-ary tree runs in $O(n)$ time.*

3.4 Remarks

Crucial to finding a balanced decomposition is a method for splitting a tree into two roughly equal pieces. The splitting method presented here is fast and relatively simple.

Splitting a tree is an operation that may be useful for solving other problems. The *evolutionary tree* problem, studied by Kannan, Lawler, and Warnow [35], is such a problem. An evolutionary tree is a tree repesenting ancestor-descendant relations between biological species in which the leaves correspond to distinct species. An internal node is an ancestor of the species contained in its subtree. Given a set of n species and the ability to perform queries of a certain sort, we wish to be able to determine the structure of the tree while minimizing the number of queries and the asymptotic time required for computation. Kannan, et. al. considered the following query: given three species, return the pair that has the deepest nearest common ancestor.

One of the algorithms they presented relies on a procedure for splitting the tree in a balanced manner by removing a vertex. For splitting they use a variant of Algorithm 1, which requires linear time. Using our splitting method, modified to find a splitting vertex instead of edge, yields an algorithm which requires $n \log_{\frac{3}{2}} n$ queries and runs in $O(n \log^2 n)$ time. This bound is equivalent to one of the bounds presented in their paper.

Another open question which is also related to the evolutionary tree problem is the *dynamic* balanced decomposition problem. That is, given a balanced decomposition of an n-node tree T, we wish to be able to perform an insertion to or a deletion from T so that a balanced decomposition of the resulting tree can be computed quickly. Such an algorithm could be used to solve the evolutionary tree problem; if the decomposition can be updated in $O(\log n)$ time after an insertion or a deletion, then the evolutionary tree can be computed in $O(n \log n)$ time with $n \log_{\frac{3}{2}} n$ queries.

Finally, a related problem is finding a parallel balanced decomposition algorithm.

4. Finding the Minimum-Cost Maximum Flow of a Series-Parallel Network

4.1 Introduction

In this chapter we give a fast algorithm for computing a minimum-cost maximum flow in a series-parallel network. On an m-edge network, the algorithm runs in $O(m \log m)$ time. The space needed is $O(m)$ if only the cost of the minimum-cost flow is desired, or $O(m \log m)$ if the entire flow is needed. This space bound can be reduced to $O(m \log \log m)$ without increasing the running time, or to $O(m)$ by increasing the running time to $O(m \log m \log \log m)$. The idea behind the algorithm is to represent a set of augmenting paths by a balanced search tree.

Let $G = (V, E)$ be a directed (multi)graph with vertex set V of size n and edge set E of size m. Each edge $e \in E$ has a *source* $s(e) \in V$ and a *sink* $t(e) \in V$; e is directed from $s(e)$ to $t(e)$. A graph G is called (*two-terminal*) *series-parallel* with *source* s and *sink* t if it can be built by means of the following three rules (see Figure 4.1):

Base graph. Any graph of the form $G = (\{s,t\},\{e\})$ with $s(e) = s$ and $t(e) = t$ is series-parallel with source s and sink t.

Let $G_1 = (V_1, E_1)$ be series-parallel, with source s_1 and sink t_1, and let $G_2 = (V_2, E_2)$ be series-parallel with source s_2 and sink t_2.

Parallel composition. The graph G formed from G_1 and G_2 by identifying s_1 and s_2 and identifying t_1 and t_2 is series-parallel, with source $s_1 = s_2$ and sink $t_1 = t_2$.

Series composition. The graph G_s formed from G_1 and G_2 by identifying t_1 and s_2 is series-parallel, with source s_1 and sink t_2.

A series-parallel graph can be represented by a *decomposition tree*, which is a binary tree in which the leaves represent edges of the graph and each internal node represents either a series or a parallel composition of the graphs

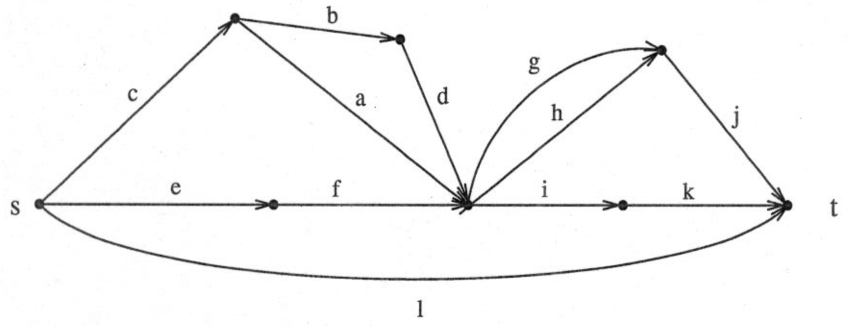


Figure 4.1: *A series-parallel graph.*

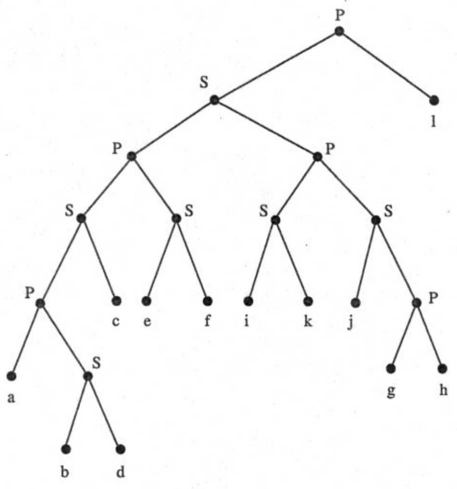


Figure 4.2: *The decomposition tree of the graph given in Figure 4.1.*

represented by its subtrees. (See Figure 4.2.) One can test an arbitrary graph to determine if it is series parallel, and if so compute a decomposition tree, in $O(n+m)$ time [53].

A *network* is a directed graph $G=(V,E)$ with two distinguished vertices, a source s and a sink t, and three real-valued functions on the edges, a *lower bound* l, a *capacity* u, and a *cost* c. A *flow* on a network is a real-valued function f on the edges satisfying the following constraints:

(1) (capacity constraints) $l(e) \le f(e) \le u(e)$ for all $e \in E$;

(2) (conservation constraints) $\sum\limits_{\substack{e \in E \\ t(e)=v}} f(e) = \sum\limits_{\substack{e \in E \\ s(e)=v}} f(e)$ for all $v \in V - \{s,t\}$.

The *value* of a flow f is $\sum\limits_{\substack{e \in E \\ t(e)=t}} f(e)$; the *cost* of f is $\sum\limits_{e \in E} c(e)f(e)$. A flow is *maximum* if it has maximum possible value and *minimum* if it has minimum possible value. A flow is *minimum-cost* if it has minimum cost among all flows with the same value. The *minimum-cost flow problem* is that of computing a maximum flow of minimum cost in a given network.

The minimum-cost flow problem can be solved in polynomial time; many algorithms are known, (see for example [18]). The best currently known bound as a function of n and m is $O((m + n \log n)m \log n)$ [41]. In the special case that the capacities and/or costs are integers of moderate size, slightly better bounds are known; see [3]

Bein, Brucker, and Tamir [6] considered the question of whether the minimum-cost flow problem becomes easier on networks of special structure, in particular on series-parallel networks. For the special case of zero lower bounds, they observed that a simple greedy strategy works, and they obtained a time bound of $O(nm + m \log m)$ for each of two algorithms, one based on the greedy strategy and one a composition algorithm using a decomposition tree.

In this chapter, we develop an algorithm for finding a minimum-cost flow in a series-parallel network in $O(m \log m)$ time. The algorithm requires $O(m)$ space if only the cost of a minimum-cost flow is desired or $O(m \log \log m)$ if the flow on each edge is needed. The latter bound can be improved to $O(m)$; however, the running time increases to $O(m \log m \log \log m)$. Finding a linear-space algorithm with $O(m \log m)$ running time is an open problem. Our algorithm is based on the ideas of Bein, Brucker, and Tamir; its efficiency comes from the use of balanced search trees to represent sequences of augmenting paths.

This chapter consists of four sections in addition to this introduction. Section 4.2 describes the idea of the algorithm. Section 4.3 discusses the implementation and efficiency analysis. Section 4.4 extends the algorithm to construct an entire minimum-cost flow rather than just computing its cost.

Section 4.5 contains some concluding remarks.

4.2 A Composition Algorithm

Our algorithm builds a representation of minimum-cost flows for all possible flow values by processing a decomposition tree for the network bottom-up. This information can be stored as a list which we call the *flow list* associated with the network. Associated with each node of a decomposition tree is the flow list corresponding to the series-parallel graph it represents.

The algorithm first computes the decomposition tree by using the algorithm of Valdes, Tarjan, and Lawler [53]. Next we compute the flow lists associated with each node of the decomposition tree by processing it bottom-up. This step relies on the observation that the flow list of a node in the tree can be computed from the flow lists of its two children; we call this procedure *composition.* Finally, the algorithm computes the flow assignment by processing the tree top-down, using the flow lists to determine at each stage how to distribute the flow.

As a preliminary observation, note that we can compute the minimum and maximum flow values, denoted by *minval* and *maxval*, respectively, in $O(m)$ time by processing a decomposition tree bottom-up using the following equations:

Base graph. Let $G = (\{s, t\}, \{e\})$ with $s(e) = s$ and $t(e) = t$. Then $minval(G) = l(e)$, $maxval(G) = u(e)$.

Let G_1 and G_2 be series-parallel. Let their parallel composition be G_p and their series composition G_s.

Parallel composition. $minval(G_p) = minval(G_1) + minval(G_2)$; $maxval(G_p) = maxval(G_1) + maxval(G_2)$.

Series composition. $minval(G_s) = \max\{minval(G_1), minval(G_2)\}$; $maxval(G_s) = \min\{maxval(G_1), maxval(G_2)\}$.
(If $minval(G_s) > maxval(G_s)$, no flow value is feasible.)

Whereas the value of a maximum flow can be computed knowing only the maximum flow values of the constituent graphs, this is not true of the minimum cost of a maximum flow. To compute the latter value, we need in general to know the costs of minimum cost flows of all possible values in the constituent graphs.

To represent the minimum cost of each possible flow value, we shall use a list we call a *flow list*. This is a list of pairs $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_k, c_k)$ with the following properties:

(i) $u_i > 0$ for $1 \le i \le k$.

(ii) $c_i \le c_{i+1}$ for $1 \le i < k$.

(iii) l_0 is the value of a minimum flow and $u_0 = l_0 + \sum_{i=1}^{k} u_i$ is the value of a maximum flow;

(iv) for any flow value x, $l_0 \le x \le u_0$, where $x = l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j$ with $0 < \alpha \le 1$, the cost of a minimum-cost flow of value x is $c_0 + \sum_{i=1}^{j-1} c_i u_i + \alpha u_j c_j$.

Note that the first pair of a flow list plays a special role; we call it *special* and the remaining pairs *normal*. We call the first component of a pair the *capacity* and the second component the *cost*. In a flow list, all the normal pairs are ordered by cost, i.e., $c_i \le c_{i+1}$ for $1 \le i < k$. Normal pairs of equal cost can be combined by adding their capacities, thereby shortening the list. We call a flow list in which all the normal pairs have distinct costs *reduced*. Figure 4.3 shows two networks and their flow lists. Roughly speaking, the special pair gives the minimum flow value, l_0, and the minimum cost of l_0 units of flow; the next u_1 units of flow can be obtained at a cost of c_1 per unit; the next u_2 units of flow can be obtained at a cost of c_2 per unit, and so on.

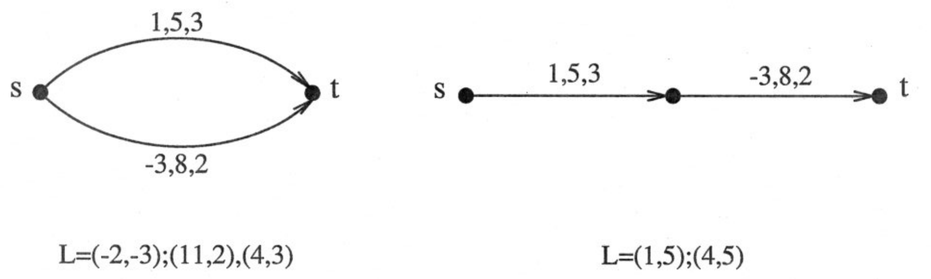


Figure 4.3: *Two simple series-parallel networks and their flow lists. Shown with each edge is a triple giving the lower bound, capacity, and cost.*

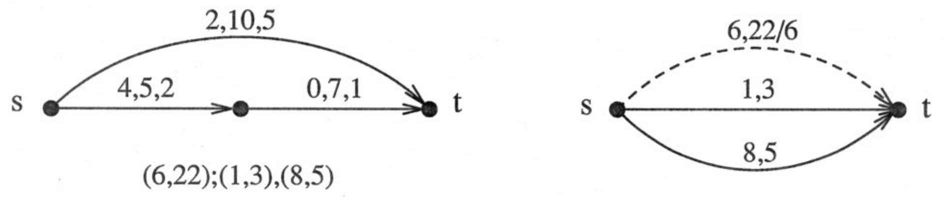


Figure 4.4: *Shown at the left is a series-parallel network and its flow list. At the right is the simplest network with the same flow list. The dashed edge represents the special pair* (l_0, c_0). *The remaining edges represent normal capacity, cost pairs.*

A flow list can be regarded as corresponding to a series-parallel network consisting of a source s, a sink t, and an edge with source s and sink t for each pair. The edge e_0 for pair (l_0, c_0) has $l(e_0) = u(e_0) = l_0$ and $c(e_0) = c_0/l_0$; for $1 \leq i \leq k$, the edge e_i for pair (u_i, c_i) has $l(e_i) = 0$, $u(e_i) = u_i$, and $c(e_i) = c_i$. See Figure 4.4 for an example. This is the simplest flow network for which the given flow list is valid, with the possible exception of networks formed by combining edges of the same cost by adding their lower bounds and their capacities. There is a unique simplest graph representing a reduced flow list. *Remark.* In our formulation of the minimum-cost flow problem, we have made no assumption about the signs of the lower bounds, the capacities, or the flows. A negative flow on an edge e can be regarded as a positive flow from $t(e)$ to $s(e)$.

A flow list for an arbitrary network can be constructed by first computing a minimum flow of minimum cost using any minimum-cost flow algorithm. This gives the first pair (l_0, c_0). Each subsequent pair is computed by finding a minimum-cost augmenting path in the residual network for the current flow, augmenting along this path, and forming a pair consisting of the capacity of the augmenting path and its cost per unit of flow. Ford and Fulkerson [21] and many other books discuss augmenting paths and minimum-cost augmentation.

For general networks, the length of even a reduced flow list can be exponential in n. For a series-parallel network, however, a reduced flow list has size $O(m)$, and constructing such a flow list provides a way to find a minimum-cost flow in such a network.

To construct a flow list for a series-parallel network, it is more efficient not to use the minimum-cost augmenting path method directly but to build the list by processing a decomposition tree bottom-up. The following rules define this computation:

Base graph. Let $G = (\{s,t\}, \{e\})$ with $s(e) = s$, $t(e) = t$. The flow list for G is $(l(e), c(e)l(e))$; $(u(e) - l(e),\ c(e))$ if $u(e) > l(e)$ or just $(l(e),\ c(e)l(e))$ if $u(e) = l(e)$. If $u(e) < l(e)$ then there is no valid flow.

Let G_1 and G_2 be series-parallel graphs with flow lists $L_1 = (l_0^1, c_0^1)$; L_1' and $L_2 = (l_0^2, c_0^2)$; L_2' respectively.

Parallel composition. For G_p, the flow list is $L_p = (l_0^1 + l_0^2, c_0^1 + c_0^2)$; L_p', where L_p' is formed by merging lists L_1' and L_2', ordering the pairs in nondecreasing order by cost, and combining pairs of equal cost by adding their capacities.

Series composition. For G_s, the flow list $L_s = (l_0^s, c_0^s)$; L_s' is constructed as follows:

Step 1. (Compute the special pair.) If $l_0^1 = l_0^2$ then $l_0 = l_0^1$ (or l_0^2) and $c_0 = c_0^1 + c_0^2$. Otherwise assume $l_0^1 > l_0^2$ (the remaining case is symmetric). Let $L_2' = (u_1^2, c_1^2), \ldots, (u_k^2, c_k^2)$. Determine j and α such that $1 \le j \le k$, $0 < \alpha \le 1$, and $l_0^1 - l_0^2 = \sum_{i=1}^{j-1} u_i^2 + \alpha u_j^2$. (If there are no such j and α, G_s has no feasible flow.)

Set $l_0^s = l_0^1$ and $c_0^s = c_0^1 + c_0^2 + \sum_{i=1}^{j-1} c_i^2 u_i^2 + \alpha c_j^2 u_j^2$. Replace L_2' by $((1-\alpha)u_j^2, c_j^2)$, $(u_{j+1}^2, c_{j+1}^2), \ldots, (u_k^2, c_k^2)$, eliminating the first pair if $\alpha = 1$.

Step 2. (Compute the remaining pairs.) Initialize L_s' to be the empty list and repeat the following step until either L_1' or L_2' (or both) is empty:

Add-insert. Remove the first pair (u^1, c^1) from L_1' and the first pair (u^2, c^2) from L_2'. Add the pair $(\min\{u^1, u^2\}, c^1 + c^2)$ to the back of L_s'. If $u^1 > u^2$, add the pair $(u^1 - u^2, c^1)$ to the front of L_1'; if $u^1 < u^2$, add the pair $(u^2 - u^1, c^2)$ to the front of L_2'.

Figure 4.5 illustrates the composition procedure.

This method is essentially the second algorithm of Bein, Brucker, and Tamir [6], modified to handle lower bounds. Its correctness is easy to verify by induction on m; we omit the details, which merely extend those in [6].

We conclude this section by recasting the parallel and series composition steps so that they more closely match the implementation to be developed in Section 4.3.

For pair $x = (u, c)$ in flow list L, we use $cap(x)$ and $cost(x)$ to denote the capacity, u, and cost, c, of x; we define the following value:

$$capsum(x) = \sum\{u | (u, c) \text{ is } x \text{ or is a pair preceding } x \text{ in } L\}.$$

Parallel composition. Computation of the special pair is the same as before. Assume $|L_1'| \leq |L_2'|$; the other case is symmetric. Let $L_1' = (u_1^1, c_1^1), \ldots, (u_j^1, c_j^1)$. Initially L_p' is the empty set. Repeat the following step for i from 1 through j:

Insert: If $L_2' = \emptyset$, add pair (u_i^1, c_i^1) to the back of L_p'. Otherwise find the first element x in L_2' such that $cost(x) \geq c_i^1$. Split L_2' at x: let A be the list containing all elements preceding x and let L_2' contain all elements succeeding x. If $cost(x) = c_i^1$, set $cap(x) = cap(x) + u_i^1$ and add x to the back of A. Otherwise, add the element (u_i^1, c_i^1) to the back of A. Concatenate A to the back of L_p'.

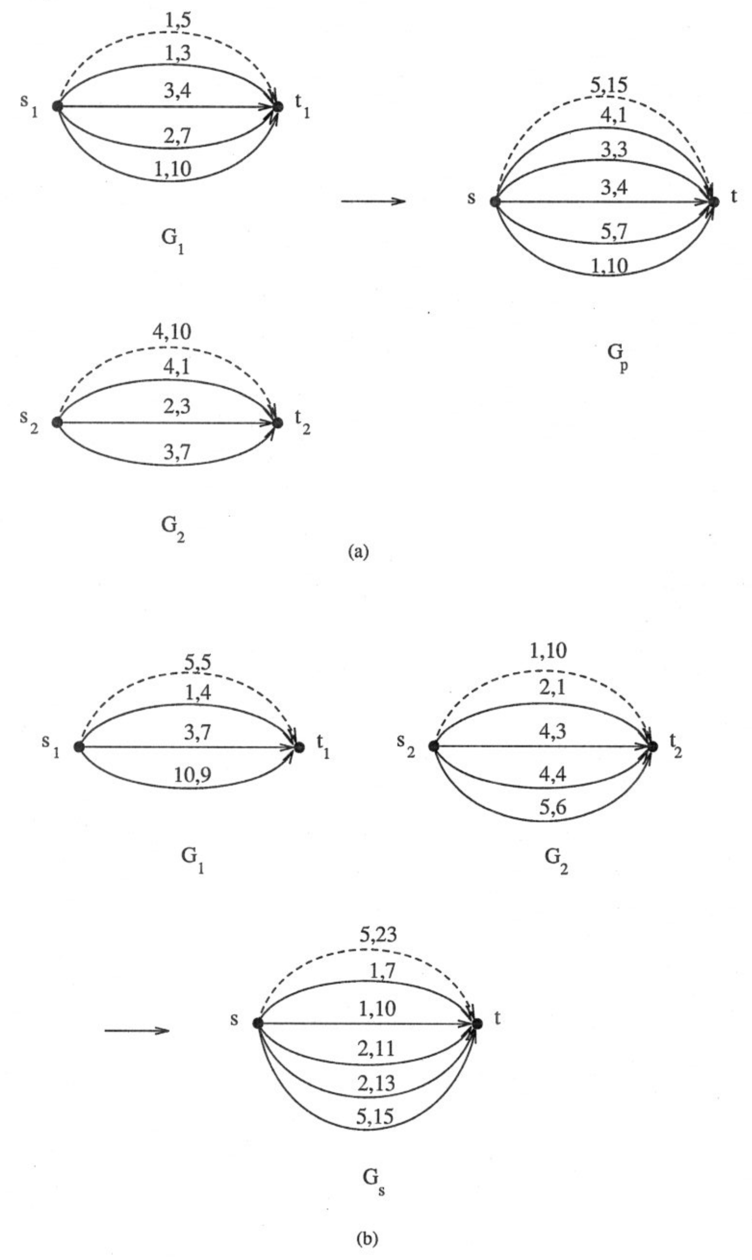


Figure 4.5: *Shown at the top is an example of the parallel composition procedure. The graphs shown are the simplest graphs corresponding to a given flow list. A dashed edges represent a special pair* (l_0, c_0). *The remaining edges represent normal capacity, cost pairs. At the bottom is an example of a series composition.*

Series composition. Assume $|L_1'| \leq |L_2'|$; the other case is symmetric.

Step 1. Compute the special pair.

If $l_0^1 = l_0^2$, set $l_0^s = l_0^1$ and $c_0^s = c_0^1 + c_0^2$.

If $l_0^1 < l_0^2$, set $l_0^s = l_0^2$, and initialize c_0^s to c_0^1, U to l_0^1, and i to 1. Repeat the *delete* step given below until $U \geq l_0^2$:

Delete: Replace U by $U + u_i^1$. If $U \leq l_0^2$, replace c_0^s by $c_0^s + u_i^1 c_i^1$, and increment i.

If $U > l_0^2$, replace c_0^s by $c_0^s + (l_0^2 - U + u_i^1)c_i^1$ and u_i^1 by $U - l_0^2$. Delete the first $i-1$ elements of L_1'.

If $l_0^1 > l_0^2$, perform the procedure given in the previous case, reversing the roles of L_1' and L_2'.

Step 2. Let $L_1' = (u_1^1, c_1^1), \ldots, (u_j^1, c_j^1)$. Repeat the following step for i ranging from 1 to j or until $L_2' = \emptyset$:

Add-insert: Find the first element x in L_2' such that $capsum(x) \geq u_i^1$. Split L_2' at x into two lists, letting A be the list containing all pairs preceding x and L_2' all the pairs succeeding x. If $capsum(x) = u_i^1$, add x to the back of A. Otherwise add the new pair $(cap(x) - capsum(x) + u_i^1, cost(x))$ to the back of A and the element $(capsum(x) - u_i^1, cost(x))$ to the front of L_2'. Increment the cost of each element of A by c_i^1 and concatenate A to the back of L_s'. Replace i by $i+1$.

To perform the above procedures we need a representation for the flow lists that supports the following list operations:

concatenate$(L_1,\ L_2)$: return the list formed by concatenating L_2 to the back of L_1.

split$(L_1,\ x,\ A,\ B)$: for x an element of L_1, construct lists A and B such that A contains the portion of L_1 preceding x and B contains the portion of L_1 succeeding x.

add cost(L_1, c) : add cost c to the costs of all items in L_1.

search cost(L_1, c) : return the item x which is the leftmost item in L_1 with cost at least c; also return *cost(x)*.

search capsum(L_1, U) : return the item x which is the leftmost item in L_1 such that *capsum*(x) is at least U; also return *capsum(x)*,

where L_1 and L_2 are lists of capacity, cost pairs, ordered by non-decreasing cost.

Pseudo-code implementing the composition procedures using the list operations given above is given in Appendix 4.1. As described in the next section, the list operations can be implemented quickly by using *finger search trees* to represent the lists.

4.3 Representation of Flow Lists as Finger Search Trees

A straightforward implementation of the algorithm of Section 4.2, using linked lists to represent the flow lists, computes a flow list for an m-edge series-parallel network in $O(m^2)$ time. We obtain an $O(m\log m)$ time bound by using finger search trees to represent the flow lists. The basis of our method is the result of Brown and Tarjan [8] that two sorted lists represented as binary trees can be merged fast, specifically in $O(m_1\log\left(\frac{m_1+m_2}{m_1}\right))$ time, where m_1 and m_2 are the sizes of the two lists with $m_1 \leq m_2$. The similarity between merging and composition of flow lists is sufficiently high that the merging algorithm can be extended to do both series and parallel composition of flow lists. Representing the flow lists as finger search trees gives a cleaner algorithm with the same running time. The result is an algorithm for flow list computation (excluding preprocessing) that has a running time bound satisfying the following recurrence:

$$T(m) = \max_{\substack{1\leq m_1\leq m_2\leq m-1 \\ m_1+m_2=m}} \left\{T(m_1) + T(m_2) + O(m_1 \log \tfrac{m}{m_1})\right\} \qquad (4.3)$$

Preprocessing can be done in linear time. We can show that this recurrence has solution bounded by $O(m \log m)$. Assuming that the constant of the last term in the recurrence is 1 changes the solution by a constant factor. For the inductive hypothesis, assume for all values $1, \ldots, m-1$, that $T(m) \leq cm \log m + m$ for some positive constant c. To complete the proof, it is sufficient to show that:

$$cm_1 \log m_1 + c(m - m_1) \log(m - m_1) + m_1 \log(\frac{m}{m_1}) \leq cm \log m$$

for all $1 \leq m_1 \leq m/2$ for some positive constant c. Setting c to 1 and expanding the last term gives: $(m - m_1) \log(m - m_1) + m_1 \log m$, which is at most $m \log m$, as claimed.

Finger search trees are an extension of balanced binary trees in which nodes are accessed via pointers to external nodes, called *fingers*, rather than through the root. This has the advantage that access time is dependent on the distance from the nearest finger rather than on the depth of the tree.

Used here is the second type of finger search tree described in Chapter 2. Recall that such a structure is created from a red-black tree by reversing the pointers along the left and right paths and adding pointers to the leftmost and rightmost external nodes.

To represent a flow list with m normal pairs, create a finger tree and associate each pair with an external node so that left-to-right order among external nodes corresponds to increasing cost of the pairs. The special pair in each list is stored separately. The keys associated with an external node (or equivalently a pair) z are denoted by u_z and c_z and are the capacity and cost of z, respectively.

In order to perform *search cost*, *search capsum*, and *add cost* operations, we store some auxiliary data with each internal node. For each node y let

$$\begin{aligned} &\textit{totcap}(y) = \Sigma\{u_z | z \text{ is an external descendant of } y\} \text{ and} \\ &\textit{maxcost}(y) = \max\{c_z | z \text{ is an external descendant of } y\}. \end{aligned}$$

Observe that if y is an external node, *totcap*$(y) = u_y$ and *maxcost*$(y) = c_y$.

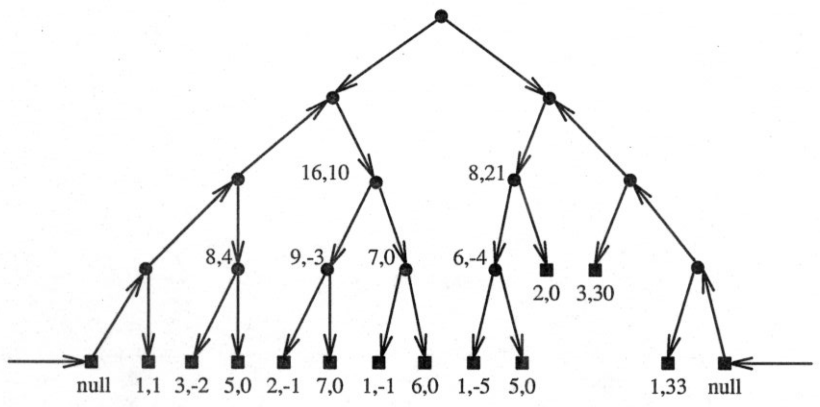


Figure 4.6: *The finger tree representation of the flow list L, where L' is* $(1,1),(3,2),(5,4),(2,6),(7,7),(1,9),(6,10),(1,12),(5,17),(2,21),(3,30),(1,33)$. *Shown with each non-path node are its* totcap *and* Δmaxcost *values.*

A node is called a *path* node if it is on the left or right path and is called a *non-path* node otherwise. Auxiliary values are stored with non-path nodes only. With each such node y store $totcap(y)$ and the additional value $\Delta maxcost(y)$ which is $maxcost(y) - maxcost(p(y))$ if $p(y)$ is not a path node and is $maxcost(y)$ otherwise.

For any non-path node y, $maxcost(y)$ is the sum of the $\Delta maxcost$ values associated with all non-path ancestors of y. Shown in Figure 4.6 are a flow list and its finger tree representation.

The list operations desired are a subset of the operations described in Chapter 2; *search cost* and *search capsum* are equivalent to the two search procedures described there. For a flow list with n elements, performing *add cost* requires $O(1+\log n)$ time. Concatenating an n-item flow list with a longer one requires $O(1+\log n)$ time. If x is the d^{th} element, then the time required for splitting at x is $O(1+\log(\min\{d, n-d\}))$ and for accessing x via either search is $O(1+\log d)$.

This leads to an analysis of the time required to perform a series or parallel composition which will complete the analysis of the running time of the entire algorithm.

Lemma 4.1 *Let L_1 and L_2 be two flow lists with m_1 and m_2 pairs respectively, $m_1 \leq m_2$. The time required for computing L'_s or L'_p is $O(m_1 \log\left(\frac{m_1+m_2}{m_1}\right))$.*

Proof: Both composition procedures consist of at most m_1 iterations of a general step. Consider the cost of the i^{th} iteration. In both composition procedures each iteration begins by accessing a pair in L'_2; let x_i be the pair found in the i^{th} iteration and let d_i be its position in L'_2 at this stage. For $i > 1$, d_i is one greater than the distance between x_{i-1} and x_i in L'_2 before the procedure starts (note that all d_i's are at least 1). This implies that $\sum_{i=1}^{m_1} d_i \leq m_1 + m_2$.

In the i^{th} iteration, some subset of the following operations is executed: access x_i, split L'_2 at x_i, add cost to A, the portion of L'_2 preceding x_i, concatenate A to a larger list, and concatenate a single element to a large list. Note that A has length d_i. Each of these operations requires at most $O(1 + \log d_i)$ time if the flow lists are represented as described above. If we ignore the constant factor, the time required to perform the parallel or series composition, excluding computation of the special pairs, is at most $\sum_{i=1}^{m_1}(1 + \log d_i)$. Concavity of the log function implies: $\frac{1}{m_1}\sum_{i=1}^{m_1} \log d_i \leq \log \frac{m_1+m_2}{m_1}$, which implies that time required for composition is $O(m_1 \log \frac{m_1+m_2}{m_1})$, as claimed. ▮

Theorem 4.1 *Computing the flow list of an m-edge series-parallel graph requires $O(m \log m)$ time and $O(m)$ space.*

Proof: By the above lemma, the time required for computing the non-special pairs satisfies the Recurrence 4.3 given at the beginning of this section, which has solution bounded above by $O(m \log m)$.

The time required for computation of the special pairs must be considered separately. For parallel composition the special pair is computed in constant time. For series composition the time required is proportional to the number of pairs deleted. Although this value can be large for a particular composition step, we can show that the number of pairs deleted overall is at most m.

Consider an m-edge graph G with flow list L. The following fact is the key. If L is formed by composing lists L_1 and L_2, then the size of L is at most the sum of the sizes of L_1 and L_2 minus the number of pairs deleted during the composition of L_1 and L_2. This implies that the length of a flow list for an m-edge graph, which is non-negative, is at most m. We can show by induction that the size of L is at most m minus the number of pairs deleted during the entire flow-list computation, which implies that the number of pairs deleted during computation of the special pairs is at most m.

The algorithm can be implemented in $O(m)$ space. Constructing and storing the decomposition tree can be done in $O(m)$ space. Recall that the decomposition tree is processed bottom-up. The general step consists of processing node x by performing a series or parallel composition on the flow lists L_1 and L_2 associated with the children of x to compute the flow list L associated with x. If L_1 and L_2 are then discarded, then the space required is $O(m)$. We can prove this by induction, using the fact that $|L| \leq |L_1| + |L_2|$. ∎

4.4 Finding the Flow Assignment

From the flow list and a valid flow value k, we can compute in $O(\log m)$ time the cost of the min-cost flow of value k as follows. Given flow list $(l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_l, c_l)$, if $1 \leq j \leq l$ and $0 < \alpha \leq 1$ are such that $l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j = k$ then the cost of the min-cost flow of capacity k is $c_0 + \sum_{i=1}^{j-1} u_i c_i + \alpha u_j c_j$. In this section we show how to compute the min-cost flow assignment.

The algorithm given in [6] computes the flow assignment in $O(nm + m \log m)$ time and $O(m)$ space. Any algorithm that uses flow lists will require at least $\Omega(m \log m)$ time and any algorithm will require $\Omega(m)$ space. The algorithms presented here do not simultaneously achieve these bounds. We present a simple algorithm that runs in $O(m \log m)$ time and requires $O(m \log m)$ space and two more complicated algorithms, each of which is off by a factor of $O(\log \log m)$. The first requires $O(m \log m \log \log m)$ time and $O(m)$ space and the second requires $O(m \log m)$ time and $O(m \log \log m)$ space.

This section contains 4 subsections. Section 4.4.1 contains the basic idea used in computing the flow assignment from the flow lists. Section 4.4.2 contains another idea that leads to two simple, more efficient algorithms. Finally, in Sections 4.4.3 and 4.4.4, we present the two algorithms which are within a factor of $O(\log\log m)$ of optimal.

4.4.1 Distributing Flow

The algorithms presented here all rely on the following observation. Let G be a series-parallel network formed by composing series-parallel networks G_1 and G_2. Given flow value k, flow list L for G, and the flow list for one of G_1 or G_2, say L_1, we can compute flow values k_1 and k_2 such that some min-cost flow of value k on G has values k_1 and k_2 on G_1 and G_2, respectively.

If $G = G_s$ it is clear that k_1 and k_2 must both be k since all flow through G must pass through both G_1 and G_2. The other case, $G = G_p$, is more complicated. In any valid flow, k_1 and k_2 must be valid flow values for G_1 and G_2 respectively and must sum to k. Among the pairs of values satisfying these contraints we desire the one with minimum cost. This can be computed from the flow lists.

Recall that the smallest graph represented by a particular flow list consists of a source and sink with one edge for each pair of the flow list. It is simpler to consider the problem of distributing flow in the parallel case on these graphs. Let G', G'_1 and G'_2 be the smallest graphs whose flow lists are L, L_1 and L_2, respectively. First determine the cost of the min-cost flow with capacity k and the values j and α as defined above. A min-cost flow having capacity k will saturate the first $j-1$ edges of G' and thus also saturates the corresponding edges of G'_1 and G'_2 which are all the edges with cost strictly less than c_j. The remaining αu_j units of flow go through the edge or edges in the constituent graphs with cost equal to c_j.

We call this procedure the *distribution procedure.* It is described more formally below.

Distribution procedure: If x represents a series composition, then $k_1 = k_2 = k$. If x represents a parallel composition, execute the following procedure. Let

$L = (10,17); (1,1)(2,4)(6,5)(2,6)(3,9)$
$L_1 = (5,8); (1,1)(3,5)(2,6)$
$L_2 = (5,9); (2,4)(3,5)(3,9)$
$k = 20$

$j = 4 \; \alpha = 1/2$
$j' = 3$
$k_1 = 5 + 1 + 3 + \min\{1/2 * 2, 2\} = 10$
$k_2 = 10$

Figure 4.7: *An example execution of the distribution procedure.*

$L = (l_0, c_0); (u_1, c_1), (u_2, c_2), \ldots, (u_l, c_l)$ be the flow list associated with G and $L_1 = (l_0^1, c_0^1); (u_1^1, c_1^1), (u_2^1, c_2^1), \ldots, (u_r^1, c_r^1)$ be the flow list associated with G_1. The following procedure gives k_1 and k_2. Determine $1 \leq j \leq l$ and $0 < \alpha \leq 1$ such that $k = l_0 + \sum_{i=1}^{j-1} u_i + \alpha u_j$. Determine the largest index j' such that $c_{j'}^1 \leq c_j$. Then if $c_{j'}^1 = c_j$ and $u_{j'}^1 \leq u_j$, set $k_1 = l_0^1 + \sum_{i=1}^{j'-1} u_i^1 + \min\{\alpha u_j, u_{j'}^1\}$. Otherwise set $k_1 = l_0^1 + \sum_{i=1}^{j'} u_i^1$. In either case set $k_2 = k - k_1$.

An execution of this procedure is illustrated in Figure 4.7.

Executing the distribution procedure on a list with m elements requires $O(\log m)$ time if the list is represented as described in Section 4.3.

The flow assignment can be computed by processing the decomposition tree top-down, using this procedure to compute the flow list associated with each node. More specifically, first run the flow list algorithm and store each flow list with its corresponding node in the decomposition tree. Then process the tree top-down, performing the distribution procedure at each node. The flow values computed for the leaves give the flow function. An example is shown in Figure 4.8.

The flow lists can be computed in $O(m \log m)$ time if no intermediate flow lists are stored. However, storing all flow lists may require $O(m^2)$ time and

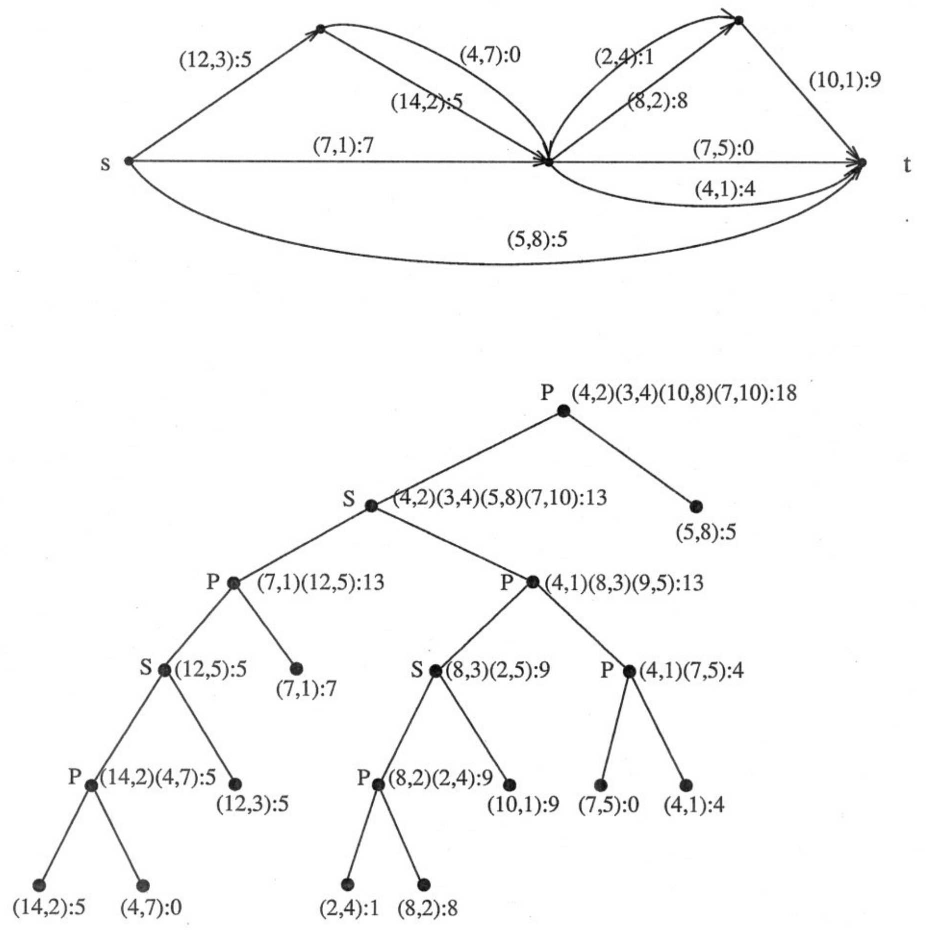


Figure 4.8: *A series-parallel graph and its decomposition tree, without lower bounds on the edge capacities. Shown with each node of the tree are its flow list and the computed flow value when 18 units of flow through the graph are desired. Shown with each edge of the network are its capacity, cost, and computed flow value.*

space. The running time of the distribution algorithm satisfies the following recurrence:

$$T(m) = \max_{\substack{1 \le m_1 \le m_2 \le m-1 \\ m_1+m_2=m}} \{T(m_1) + T(m_2) + O(\log m)\} \tag{4.4}$$

This has solution bounded above by $O(m \log m)$. The space required for a particular decomposition tree is bounded above by the sum of the number of external descendants of each node. For an m-node decomposition tree this sum is $\Theta(m^2)$. The worst case occurs when T is a chain of length $m-1$. Thus the flow assignment can be computed in $O(m^2)$ time and $O(m^2)$ space.

4.4.2 Uncomposition

The space can be reduced to $O(m \log m)$ by observing that we do not need to store every flow list. We can get away with storing the flow list associated with only one of each pair of siblings in the tree if the flow list associated with the other member of each pair can be recomputed. Performing the distribution procedure at node x requires only the flow list for x, the flow list of one child of x, and the flow value for x. From this plus a small amount of auxiliary information, we can compute the flow list for the other child and can then recursively process the children. This procedure is the inverse of composition and is called *uncomposition*. Let L be the flow list associated with a node x and let L_1 and L_2 be the flow lists associated with its children. Assume that L and L_1 are known. We will show how to compute L_2.

For the parallel case uncomposition is easy. The special pair, (l_0^2, c_0^2) is $(l_0 - l_0^1, c_0 - c_0^1)$; L_2' is computed by unmerging L' and L_1' with respect to cost. In the series case the computation is more complicated. If $l_0^1 < l_0$, then $l_0^2 = l_0$ and c_0^2 is found by performing a calculation very similar to the special pair calculation used in composing. If $l_0^1 = l_0$, then l_0^2 cannot be determined. It is in this case that auxiliary information is needed. In order to uncompose, we store the special pair of L_2 and the pairs of L_2' deleted in composing L_1 and L_2 to compute L. We call this list L_2^{aux}. Note that it is already computed during the flow list algorithm. After determining the special pair, L_2' is computed by unmerging with respect to capacity sums and subtracting costs (the inverse of the series composition procedure).

If L has size m and L_1 has size m_1 then uncomposing L and L_1 requires $O(m_1 \log(m/m_1))$ time. Below we give a detailed description of the uncomposition procedures.

Parallel uncomposition. The special pair of L_2 is $(l_0 - l_0^1, c_0 - c_0^1)$. Form L_2' as follows. Let $L_1' = (u_1^1, c_1^1), \ldots, (u_j^1, c_j^1)$. Initially L_2' is the empty list. Repeat the following step for i from 1 through j:

Delete: Find the element x in L' such that $cost(x) = c_i^1$. Split L' at x: let A be the list containing all elements preceding x and let L' contain all elements succeeding x. If $cap(x) > u_i^1$, set $cap(x) = cap(x) - u_i^1$ and add x to the back of A. Join A to the back of L_2'.

Series uncomposition.

Step 1. Compute the special pair. Initialize L_2' to the empty set.

If $l_0 > l_0^1$, set $l_0^2 = l_0$ and initialize j to 1, U to l_0^1, and c_0^2 to $c_0 - c_0^1$.

Repeat the following step until $U \geq l_0$:

Delete. Replace U by $U + u_j^1$. If $U \leq l_0$ replace c_0^2 by $c_0^2 - u_j^1 c_j^1$, and increment j.

If $U > l_0$, replace (u_j^1, c_j^1) by the pairs $(u_j^1 - U + l_0, c_j^1)$ and $(U - l_0, c_j^1)$. Set $c_0^2 = c_0^2 - (u_j^1 - U + l_0)c_j^1$ and delete the first i elements of L_1'.

If $l_0 = l_0^1$, set L_2 to L_2^{aux}.

Step 2. Let $L_1' = (u_1^1, c_1^1), \ldots, (u_j^1, c_j^1)$. Repeat the following step for i ranging from 1 to j:

Subtract-delete: Find the first element x in L' such that $capsum(x) = u_i^1$. Split L' at x into two lists, letting A be the list containing all pairs preceding x and L' all the pairs succeeding x. Add x to the back of A. Subtract c_j from the costs of all pairs in A. Let v be the last pair in L_2'. Let y be the first element of

A. If $cost(y) = cost(v)$, delete y from A and replace $cap(v)$ by $cap(v)+cap(y)$. Join A to the back of L_2'.

Note that in the series case the unknown list may not be completely reconstructed. If the total capacity of L_2 was greater than the total capacity of L_1, the extra capacity can never be used in the composite graph represented by flow list L and the portion of L_2' corresponding to this capacity is lost.

These procedures can be implemented using the list operations given in Section 4.2. The analysis of the running time is almost identical to the analysis of the composition algorithm. Uncomposing lists L and L_1 of sizes m and m_1 requires $O(m_1 \log(m/m_1))$ time.

This suggests an algorithm for finding the flow assignment that requires less space. For each decomposition tree node x we define the *size* of x, denoted $s(x)$, to be the number of external descendants of x. For each pair of siblings u and v in the decomposition tree, we call one *small* and one *large* depending on the sizes of the nodes. If the nodes have different sizes then the node with smaller size is called small. Otherwise one node is arbitrarily chosen to be designated small and the other large.

Algorithm 2 *Let x be a decomposition tree node with small child u and large child v. Let L, L_1, and L_2 be the flow lists for x, u, and v respectively and L_2^{aux} the auxiliary list for v.*

1. *Run the flow list algorithm on T, storing the auxiliary lists of the large nodes and the flow lists associated with the small nodes.*
2. *Process T top-down, processing node x as follows. At node x, given its flow value k: distribute to compute k_1 and k_2, the flow values for u and v, uncompose to compute L_2, and discard L.*
3. *For each edge e, the flow value $f(e)$ is the flow value associated with the leaf representing e in T.*

Lemma 4.2 *Algorithm 2 runs in $O(m \log m)$ space and time on an m-edge series-parallel network.*

Proof: Storage is required for the flow lists associated with small nodes and also for the auxiliary lists. The space required for storing the flow lists of the small nodes satisfies the following recurrence:

$$S(m) = \max_{\substack{1 \leq m_1 \leq m_2 \leq m-1 \\ m_1 + m_2 = m}} \{S(m_1) + S(m_2) + O(m_1)\} \tag{4.5}$$

This has solution bounded by $O(m \log m)$. The auxiliary lists contain the pairs deleted in computation of the special pairs. We showed in Theorem 4.1 that these pairs have total size at most $O(m)$.

Space is also required to hold the flow lists of the nodes on the frontier of the computation - that is, nodes that have been processed but whose children have not. The nodes in this set are unrelated (i.e., no node in this set is an ancestor of another). This implies that the sum of the sizes of the associated flow lists is at most $O(m)$.

Step 1 requires executing the flow list algorithm, which runs in $O(m \log m)$ time. Distributing flow in Step 2 requires $O(m \log m)$ time (as shown in Section 4.4.1). The time required for uncomposition satisfies recurrence (4.1). This implies that the running time is $O(m \log m)$. ∎

To conclude this section we present an alternative algorithm for computing the flow assignment, one that requires only $O(m)$ space. Again we process the decomposition tree top-down, computing the flow list associated with node x by running the flow list algorithm on the graph represented by x. This requires $O(m)$ space since at each stage only the flow lists of a set of unrelated nodes are stored, as above, but is very slow. Use of the uncomposition procedures to compute the flow lists of the large nodes reduces the running time a bit. This algorithm is described in detail below.

Algorithm 3 *Given as input are the decomposition tree T for a series-parallel network and the desired flow value. Let x be a decomposition tree node with small child u and large child v. Let L, L_1, and L_2 be the flow lists associated with x, u, and v, respectively.*

1. *Run the flow list algorithm on T, storing the auxiliary lists of the large nodes.*

2. *Process T top-down. For node x which has been processed but whose children have not, process the children of x as follows: run the flow list algorithm on the graph represented by u to compute L_1, uncompose to compute L_2, and distribute to compute the flow values for u and v.*

Lemma 4.3 *Algorithm 3 runs in $O(m)$ space and $O(m\log^2 m)$ time.*

Proof: The auxiliary lists are stored during the entire algorithm and take up only $O(m)$ space. Temporarily stored during each stage are the flow lists associated with the nodes that have been processed but whose children have not. These nodes are all unrelated; thus the sum of the sizes of their flow lists is at most $O(m)$. This implies that Algorithm 2 requires $O(m)$ space.

The running time is harder to analyze. Computation of the auxiliary lists, distribution and uncomposition altogether require $O(m \log m)$ time. The calls to the flow list algorithm cause the composition algorithm to be executed a variable number of times per node. The number of times we perform the composition procedure at a node x is equal to the number of small ancestors of x. For each node x let $l(x)$ denote the number of small ancestors of x. This value can be defined recursively as follows: $l(x) = 1$ if x is the root of T and otherwise, $l(x)$ is $l(p(x))$ if x is large and $l(p(x)) + 1$ if x is small. We call $l(x)$ the *label* of x. The time required for composition can be expressed as a function of the number of leaves of the decomposition tree and the label $l(x)$ associated with the root as follows:

$$F(m,i) = \max_{\substack{1 \le m_1 \le m_2 \le m-1 \\ m_1+m_2=m}} \left\{ F(m_1, i+1) + F(m_2, i) + O(im_1 \log\left(\frac{m}{m_1}\right)) \right\}$$

We will show by induction that $F(m,i) \le 2m\log^2 m + im\log m$.

Assuming that the constant factor of the last term is 1 only affects the constant of the result. It is sufficient to show for all values of m_1 and m_2 summing to m, $1 \le m_1 \le m/2$, that:

$$2m_1 \log^2 m_1 + (i+1)m_1 \log m_1 + 2m_2 \log^2 m_2 + im_2 \log m_2 + im_1 \log(\frac{m}{m_1})$$

is less than or equal to $2m\log^2 m + im\log m$. It suffices to show that

$$2m_1\log^2 m_1 + m_1\log m_1 + 2m_2\log^2 m_2 \leq 2m\log^2 m$$

and that

$$i(m_1\log m + m_2\log m_2) \leq im\log m.$$

For the first claim, the left hand side is less than or equal to

$$2m_1\log^2(m/2) + m_1\log(m/2) + 2m_2\log^2 m,$$

which is at most

$$2m\log^2 m - 3m_1\log m + m_1 \leq 2m\log^2 m.$$

The second claim is obvious.

The running time of our algorithm is bounded above by $F(m,1)$, which is $O(m\log^2 m)$. ∎

4.4.3 An Optimal Time, Almost Optimal Space Algorithm

Finally we present two algorithms which are off by a factor of $O(\log\log m)$ from optimal, one in space and one in time. The improvement is achieved by splitting the decomposition tree into pieces and processing the pieces separately. We will first describe the basic ideas which are common to both algorithms. The first algorithm requires $O(m\log m)$ time and $O(m\log\log m)$ space and the second requires in $O(m\log m\log\log m)$ time and $O(m)$ space.

First we need a few definitions. We classify each node of the tree as a *bottom* or a *top* node according to its size, which is the number of leaves in its subtree. Node x is a top node if $s(x) \geq m/\log m$ and is a bottom node otherwise. The set of nodes x such that x is a bottom node and its parent is a top node are called *border* nodes.

The set of top nodes is a subtree of T which has the root of T as its root. We use T' to denote this subtree and refer to it as the *top subtree* of T. The set of border nodes is made up of the children of the leaves of T' plus all

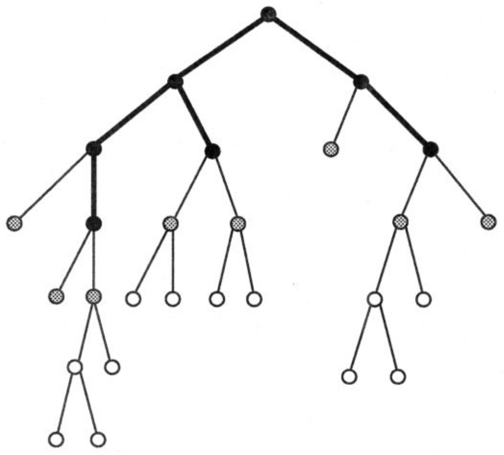


Figure 4.9: *A decomposition tree with $m = 16$ leaves. Nodes in the top tree are black and the border nodes are gray. The darker edges are the edges in T'.*

nodes that are not contained in T' and whose siblings are. The bottom nodes are all contained in the subtrees rooted at the border nodes. We will call these the *border subtrees.* The border subtrees contain all the leaves of the decomposition tree. This implies that the sum of the sizes of the border nodes is m. Figure 4.9 illustrates these definitions.

In both algorithms the tree is processed in three stages. First compute the flow lists for the border nodes by processing the small subtrees bottom-up; second, process T' to compute the flow values of the border nodes; and third, process the small subtrees to compute the flow values of the leaves.

The first and third stages can be executed in $O(m \log m)$ time on an m-edge network. The two algorithms presented here differ only in implementation of the second stage. We will first describe and analyze the first and third stages.

Stage 1: *For each border node x_i run the flow list algorithm, storing no intermediate flow lists. Store the flow list for each x_i.*

Lemma 4.4 *Performing Stage 1 on an m-edge network requires $O(m \log m)$ time and $O(m)$ space.*

Proof: Let s_i denote the size of x_i. The time required for computing the flow list of an m-edge graph is $O(m \log m)$. This implies that the time required for computing the flow list for x_i is $O(s_i \log s_i)$ since the graph associated with x_i has s_i edges. The total time required is proportional to $\sum_i s_i \log s_i$, which is $O(m \log m)$. The space required is the sum of the s_i's which is $O(m)$. ∎

In the third stage we are given the flow lists and flow values for each border node and wish to compute the flow values at the leaves. To do this, use Algorithm 2 given in the previous section on each border node x_i.

Stage 3 *For each border node x_i, process the subtree T_i rooted at x_i by using Algorithm 2. After processing T_i delete all associated flow lists.*

Lemma 4.5 *Performing stage 3 on an m-edge network requires $O(m \log m)$ time and $O(m)$ space.*

Proof: We showed above that Algorithm 2 requires $O(m \log m)$ time and space on an m-edge network. Let x_i be the i^{th} border node, let T_i be its subtree, and let s_i be its size. The time required for processing T_i is $O(s_i \log s_i)$ and the total time is proportional to $\sum_i s_i \log s_i$ which is $O(m \log m)$.

Again, $O(m)$ space is required for storage of the flow lists of the x_i's. During the processing of each x_i, $O(s_i \log s_i)$ space is required for storing the flow lists of the small nodes in its subtree. This implies that the space required for the entire stage is $O(m)$ plus the maximum over all i of $O(s_i \log s_i)$. Since each x_i is a bottom node, $s_i \leq m/\log m$, and thus $s_i \log s_i \leq (m/\log m)\log(m/\log m)$ which is $O(m)$. This implies that the storage required is $O(m)$. ∎

We now describe how to perform stage 2. In stage 2 we are given a decomposition tree T along with the flow lists for the border nodes and wish to compute the flow values for the border nodes.

This can be achieved by applying either of the two flow assignment algorithms given in the previous section to the top tree T'. As described in the previous section, Algorithms 2 and 3 take as input a full binary tree plus the flow lists associated with the leaves. These flow lists consist of a single special pair and one or zero regular pairs.

To process T' these algorithms must accept a slightly more general type of input. The input tree is still a full binary tree but the flow lists associated with the leaves can be long; if T' is the top tree for an m-leaf decomposition tree T, then the sum of the sizes of the flow lists of the leaves is at most m. We will call these types of trees *truncated* decomposition trees.

Both the algorithms given in the previous section can be applied to truncated decomposition trees without any modification.

To process T' in the second stage the first algorithm we will present uses Algorithm 2, in which the flow is computed by preprocessing the tree to compute and store the smaller flow lists and auxiliary lists. This requires $O(m \log m)$ time and uses $O(m \log \log m)$ space.

Algorithm 4

1. *Execute stage 1 as described above.*
2. *Apply Algorithm 2 to T'.*
3. *Execute stage 3 as described above.*

Lemma 4.6 *Algorithm 4 runs in $\Theta(m \log m)$ time and requires $\Theta(m \log \log m)$ space.*

Proof: We have already shown that executing stages 1 and 3 requires $O(m \log m)$ time and $O(m)$ space. We will show that the time and space bounds for the second stage are as stated.

The running time is easy to analyze. On an m-edge network, Algorithm 2 requires $O(m \log m)$ time. For this stage we execute this algorithm only on the top subtree of T. Clearly the time required for this is bounded above by $O(m \log m)$. There exist series-parallel graphs for which the decomposition tree T is a chain of length equal to the number of edges. That is, T consists of a path of length $m - 1$ with m external nodes. In this case, T' consists of a chain of length $m - m/\log m$. For appropriate assignment of edge capacities and costs, using Algorithm 2 to process T' will require at least $\sum_{i=\frac{m}{\log m}}^{m} \log i = \Omega(m \log m)$ time.

Executing stage two requires linear space for storing the flow lists of the border nodes and for temporary storage of flow lists used by Algorithm 1. Storage is also required for the flow lists associated with the small top nodes.

Let T be an m-leaf decomposition tree with top tree T'. Observe that the sibling of a small node in T' must also be contained in T'. Thus the storage required for storing the small flow lists in stage 2 can be expressed as :

$$\sum \{\min\{s(x), s(sib(x))\} | x, sib(x) \in T'\}, \tag{4.6}$$

where $sib(x)$ denotes the sibling of x.

It is easier to bound this quantity by considering a more general problem first. For T a full binary tree with m leaves and k a non-negative integer, we define:

$$S(T, k) = \sum \{min\{s(x), s(sib(x))\} | s(x) \geq k \text{ and } s(sib(x)) \geq k\}$$

and define $f(m, k)$ to be the maximum $S(T, k)$ value taken over all full trees T having m leaves. Thus the expression given in equation 4.6 is bounded above by $f(m, m/\log m)$. We will show that this is $O(m \log\log m)$ by showing that $f(m, k)$ is $O(m \log(m/k))$ for $m > k$.

Consider a particular full binary tree T with m leaves. Let T_1 and T_2 denote the two subtrees of T. Let m_1 be the size of T_1 and m_2 the size of T_2; assume $m_1 \leq m_2$. Clearly, $S(T, k)$ is 0 if $m \leq k$. Otherwise, $S(T, k)$ is $S(T_1, k) + S(T_2, k) + m_1$ if $m_1 \geq k$ and is $S(T_2, k)$ if $m_1 < k$. Note also that $S(T, k) \leq f(m, k)$.

This implies that $f(m, k)$ can be expressed recursively as follows:

$$f(m, k) = \max_{\substack{m_1 + m_2 = m \\ m_1 \leq m_2}} \begin{cases} f(m_1, k) + f(m_2, k) + m_1 & \text{if } k \leq m_1 \\ f(m_2, k) & m_1 < k < m \\ 0 & m < k \end{cases}$$

It is easy to show by induction that $f(m, k) \leq \max\{m \log(m/k), 0\}$. This is clear for $m \leq k$. Now assume that $m > k$. It is necessary to show for all m_1 and m_2 such that $m_1 + m_2 = m$ and $m_1 \leq m_2$, that:

$$m_1 \log(m_1/k) + m_2 \log(m_2/k) + m_1 \leq m \log(m/k) \quad \text{if } m_1 \geq k \text{ and}$$

$$m_2 \log(m_2/k) \leq m \log(m/k) \quad \text{if } m_1 < k.$$

The latter is obvious.

The former is true if :

$$m_1 \log m_1 + m \log m_2 + m_1 \leq m \log m \text{ and}$$

$$m \log k \leq m_1 \log k + m_2 \log k.$$

The first equation given above is easy to show and the second is obvious. This completes the proof that $f(m,k) \leq \max\{m \log(m/k), 0\}$, which implies that the storage required for stage 2 is bounded by $O(m \log(m/(m/\log m)))$, which is $O(m \log \log m)$.

This is also a lower bound on the space required. Consider a complete binary tree T having m leaves where m is a power of 2. The top tree, T' consists of the first $\log \log m$ levels of T and is also a complete binary tree. The i^{th} level contains 2^i nodes, half of which are small. Since each of these nodes has size $m/2^i$, the sum of the sizes of the small nodes at any level i, $i \leq \log \log m$, is $O(m)$. Thus the sum of the sizes of the small nodes in T', a lower bound for the storage required for stage 2, is $m \log \log m$. ▮

4.4.4 An Optimal Space, Almost Optimal Time Algorithm

The final algorithm we present is also sub-optimal by a factor of $\log \log m$. The running time is $O(m \log m \log \log m)$ and the space required is $O(m)$. This algorithm has the same first and third stages as Algorithm 4, but has a different second stage. Now to process the top tree T', use Algorithm 3, the assignment algorithm in which no intermediate flow lists are stored. Note that Algorithm 3 is again applied only to the truncated tree T'.

Algorithm 5

1. *Execute stage 1 as described above.*
2. *Apply Algorithm 3 to T'.*

3. Execute stage 3 as described above.

Lemma 4.7 *Algorithm 5 runs in $\Theta(m \log m \log\log m)$ time and requires $\Theta(m)$ space.*

Proof: We have already shown that steps one and three require $O(m \log m)$ time and $O(m)$ space. We showed in Section 4.4.2 that the space required for executing Algorithm 3 is $\Theta(m)$. The upper bound of the running time required for the second stage can be analyzed as follows.

Recall that Algorithm 3 works by processing the decomposition tree top down. To process node x having flow list L and flow value k, run the flow list algorithm on the small child of x to get L_1, uncompose to get L_2, distribute to get the flow values for the children, and recurse on the children. Note that all operations are executed only on T'; nodes not in T' are never encountered during this stage.

Distribution and uncomposition on T' can be done in $O(m \log m)$ time. Recall from the proof of Lemma 4.3 that for node x the value $l(x)$ gives the number of times the composition procedure is executed at x. If u is the small child of x, composing at x requires time proportional to $s(u)\log((s(x)+s(u))/s(u))$. This implies that for a particular decomposition tree T with top tree T', the time required for executing Algorithm 3 on T' is on the order of

$$S(T) = \sum_{x \in T'} l(x)\left(s(u)\log\frac{s(x)+s(u)}{s(u)}\right).$$

The running time of Stage 2 is the maximum such value taken over all decomposition trees having m leaves. We will show that this is $\Theta(m \log m \log\log m)$.

First we show that for any decomposition tree T with m leaves, $S(T) = O(m \log m \log\log m)$. Let T be a tree with m leaves and top tree T'. From the definition of the node labels, it is easy to see that for all x, $l(x) \geq k$ implies that $s(x) \leq m/2^k$. This implies that all nodes x in T' have labels less than or equal to $\log\log m$, since all such nodes have size at least $m/\log m$.

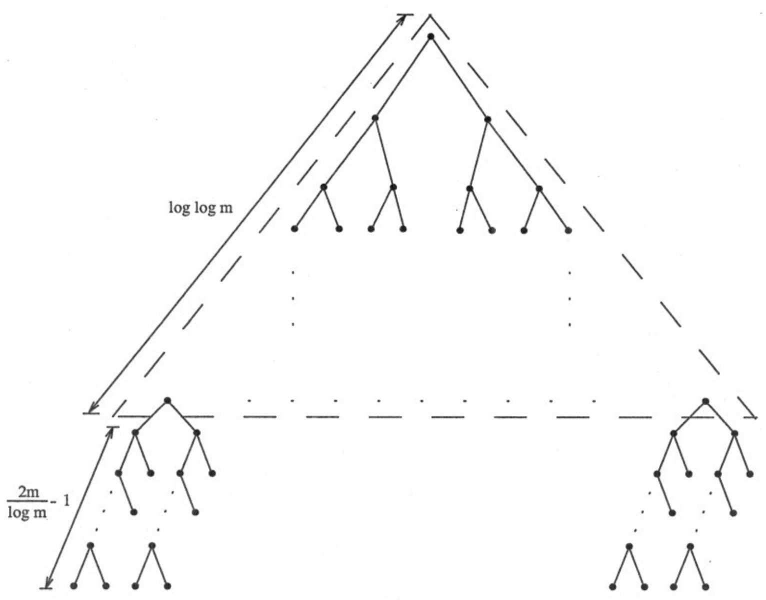


Figure 4.10: *The tree used to prove the lower bound on the running time of Algorithm 5.*

This implies that $S(T)$ is bounded above by $\log\log m$ times the following sum:

$$\sum_{x \in T'} s(u) \log \frac{s(x)}{s(u)}.$$

But this is exactly the cost of running the flow list algorithm on T'. By Theorem 4.1 this sum is bounded above by $O(m \log m)$ which implies $S(T) = O(m \log m \log\log m)$.

This bound is tight. We will show that for all m there exists an m-leaf tree T such that $S(T) = \Omega(m \log m \log\log m)$. Such a tree is constructed as follows.

Let T_0 be a complete binary tree having $\log\log m$ levels. Hang from each leaf of T_0 a chain of length $(2m/\log m) - 1$ so that the root of the chain is a leaf of T_0. Each node of the chain (except for the last) has 2 children; one is on the chain and the other is an external node. The leaf of the chain has two external nodes as children. Figure 4.10 gives an example. The tree T so formed has m leaves.

The set of top nodes, T', contains T_0 plus the first $m/\log m$ internal nodes of each chain (see Figure 4.10). The cost of performing the composition procedure on the chain nodes alone, denoted by $S'(T)$, is given by:

$$S'(T) = \sum_{x \text{ a large chain node}} \left(s(u) \log \frac{s(x)}{s(u)} \right) l(x).$$

Clearly, $S'(T) \leq S(T)$.

Let x be a leaf of T_0. Each large descendant of x has the label $l(x)$, the same label as x. In addition, the small child of each large chain node has size 1. Thus the value $S'(T)$ can be rewritten as:

$$\sum_{x \text{ a leaf of } T_0} l(x) \sum_{i=m/\log m}^{2m/\log m} \log i,$$

which is at least $cm \sum\{l(x)|x \text{ is a leaf of } T_0\}$ for any constant $0 < c < 1$. The corollary below gives a lower bound for this sum.

Claim 1 *Let T be a complete binary tree with at least d levels. Associated with each node x of T is an integer label such that the label of the root is 1 and the children of a node with label i have labels i and $i+1$. Then at level d the number of nodes with label i is $\binom{d-1}{i-1}$.*

Proof: We will show this by induction. Let n_i^l be the number of nodes with label i at level l. At level 1 the root is the only node and $n_1^1 = 1$. At any level l, each node with label i has two children, one with label i and one with label $i+1$. Thus for $i > 1$, $n_i^l = n_i^{l-1} + n_{i-1}^{l-1}$. If the claim is true for levels $1, 2, \ldots, d-1$, then $n_i^d = \binom{d-2}{i-1} + \binom{d-2}{i-2} = \binom{d-1}{i-1}$.

Corollary 4.2 *The sum of labels of nodes at level d is $2^{d-2}(d+1)$.*

By Claim 1, the sum is $\sum_{i=1}^{d} \binom{d-1}{i-1} i$, which is $2^{d-2}(d+1)$.

This implies that the sum of the labels of nodes at level $\log \log m$ is at least $(1/4) \log m \log \log m$ and hence that $S'(T)$ is $\Omega(m \log m \log \log m)$, completing the proof. ∎

4.5 Remarks

There are several open problems arising from this work. We would like to find an algorithm for finding the flow function that runs in $O(m \log m)$ time and uses $O(m)$ space. The algorithms presented here all require a factor of $O(\log \log m)$ more in either space or time. It is not known whether this algorithm is optimal with respect to time alone. Finding a faster algorithm or proving that $O(m \log m)$ time is required is a second open problem. A final problem is that of finding a minimum-cost flow in parallel.

Appendix 4.1

Given below are the procedures for performing parallel and series composition on the lists $L_1 = (l_0^1, c_0^1); L_1'$ and $L_2 = (l_0^2, c_0^2); L_2'$, where L_1' is $(u_1^1, c_1^1), \ldots, (u_j^1, c_j^1)$ and L_2' is $(u_1^2, c_1^2), \ldots, (u_k^2, c_k^2)$.

Parallel Composition

```
$l_0^p = l_0^1 + l_0^2;\ c_0^p = c_0^1 + c_0^2$
for $i = 1, 2, \ldots, j$
    $x =$ search cost($L_2'$, $c_i^1$)
    split($L_2'$, $x$, $A$, $L_2'$)
    if $cost(x) = c_i^1$
        $cap(x) = cap(x) + u_i^1$
        replace $A$ by concatenate($A$, $x$)
    else replace $A$ by concatenate($A$, create list($u_i^1$, $c_i^1$))
    replace $L_p'$ by concatenate($L_p'$, $A$).
```

Series Composition

```
if $l_0^1 = l_0^2$
    $l_0 = l_0^1;\ c_0 = c_0^1 + c_0^2$
else
    if $l_0^1 < l_0^2$
```

```
            $l_0^s = l_0^2$
            $c_0^s = c_0^1$; $U = l_0^1$
            let $(u^1, c^1)$ be the first pair of $L_1'$
            while $(U < l_0^2)$ and $(L_1' \neq \emptyset)$
                $U = U + u^1$
                if $U \leq l_0^2$
                    $c_0^s = c_0^s + u^1 c^1$
                    delete the first pair of $L_1'$
                    let $(u^1, c^1)$ be the new first pair
            if $L_1' = \emptyset$ and $U < l_0^2$
                exit; /* no valid flow exists */
            if $U > l_0^2$
                let $\hat{u} = u^1 + l_0^2 - U$
                $c_o^s = c_0^s + \hat{u} c^1$
                replace $u^1$ by $u^1 - \hat{u}$
        else /* perform the same procedure as above,
            reversing the roles of $L_1'$ and $L_2'$ */
    $i = 1$
    while $i \leq j$ and $L_2' \neq \emptyset$
        $x = search\ capsum\ (L_2',\ u_i^1)$
        $split(L_2',\ x,\ A,\ L_2')$
        if $capsum(x) = u_i^1$, replace $A$ by $concatenate(A,\ x)$
        else
            let $L = create\ list((cap(x) - capsum(x) + u_i^1,\ cost(x))$
            replace $A$ by $concatenate\ (A,\ L)$
            let $L = create\ list((capsum(x) - u_i^1,\ cost(x)))$
            replace $L_2'$ by $concatenate(L,\ L_2')$
        $add\ cost(A,\ c_i^1)$
        Replace $L_s'$ by $concatenate(L_s',\ A)$
```

5. Linear Algorithms for Analysis of Minimum Spanning and Shortest Path Trees of Planar Graphs

5.1 Introduction

We give an algorithm which solves several related graph problems. The algorithm runs in linear time and space when the input graph is planar. It can be used to analyze the sensitivity of a minimum spanning tree to changes in edge costs, to find its replacement edges, and to verify its minimality. It can also be used to analyze the sensitivity of a single-source shortest path tree to changes in edge costs, and to analyze the sensitivity of a minimum cost network flow. The algorithm is simple and practical. It uses the properties of a planar embedding, combined with a heap-ordered queue data structure.

Let $G = (V, E)$ be a planar graph, either directed or undirected, with n vertices and $m = O(n)$ edges. Each edge $e \in E$ has a real-valued cost $cost(e)$. This cost is allowed to be negative. A minimum spanning tree of a connected, undirected planar graph G is a spanning tree of minimum total edge cost. If G is directed and r is a vertex from which all other vertices are reachable, then a shortest path tree from r is a spanning tree that contains a minimum-cost path from r to every other vertex.

Sensitivity analysis measures the robustness of a minimum spanning tree or shortest path tree by determining how much the cost of each individual edge can be perturbed before the tree is no longer minimal [42,50].

We consider the following problems:

- Finding the replacement edges of a minimum spanning tree, and verifying its minimality.
- Performing sensitivity analysis of a minimum spanning tree.
- Performing sensitivity analysis of a shortest path tree.
- Performing sensitivity analysis of a minimum-cost network flow.

Let e be some edge in a minimum spanning tree of G. The *replacement edge* for e is the non-tree edge that replaces e in the minimum spanning tree of $G' = (V, E - \{e\})$. Finding replacement edges is an important component of an algorithm for determining the k smallest spanning trees of a graph [19,22].

Sensitivity analysis of shortest paths and network flows has been studied by Shier and Witzgall [42] and Gusfield [30]. The fastest known algorithms for all these problems are due to Tarjan [50,46] and run in time and space $O(m\alpha(m,n))$, where α is the functional inverse of Ackermann's function. Gabow [23] also achieves these bounds.

Here we show that in the special case of planar graphs, these problems can be solved in $O(n)$ time and space.

Our result also remedies a lacuna in Fredrickson's proof of Theorem 9, reference [22], which is incorrect without an $O(n)$ algorithm for finding replacement edges in a planar graph.

The above problems can all be solved by an algorithm for what we call the *critical edge problem.* We are given an undirected planar graph G containing a spanning tree T, rooted at vertex r. We allow G to have multiple edges and loop edges, but for convenience we will continue to call G a graph, rather than a multigraph or a pseudograph. For each vertex v we wish to determine the minimum-cost non-tree edge with exactly one endpoint a descendant of v. We call this edge the *critical edge* for vertex v. In this chapter we first give a critical edge algorithm and then describe its application to the problems listed above. The results of this chapter are joint work with Jeffery Westbrook.

5.2 Preliminaries

We assume we are given an embedding of $G = (V, E)$ in the plane in which r, the root of T, is on the outer face. Such an embedding always exists (see Harary [31], p. 105) and can be generated in $O(n)$ time using the algorithms of Hopcroft and Tarjan [33] or Booth and Leuker [7] (see Chiba et al. [12]). The embedding of G specifies the order in which edges incident to $v \in V$ are encountered as we walk around v in the counterclockwise direction (this ordering defines the embedding). We assume a standard representation of the embedding in which the counterclockwise successor of an edge around a vertex can be found in constant time.

Since a loop edge cannot be a critical edge, we assume that any loops in G have been removed in an $O(n)$ preprocessing stage. If u and v are vertices, $\{u, v\}$ denotes the undirected edge with endpoints u and v, (u, v) denotes a directed edge from u to v and $p(v)$ denotes the parent of v in T.

From the embedding of G we construct for each vertex, v, a linear edge list $\{e_0, e_1, \ldots, e_d\}$ where $d+1$ is the degree of v. Edge e_0 is the edge from v to its parent in T and the remaining edges $e_1, \ldots, e_d$ are listed in the order in which they are encountered by walking counterclockwise around v from e_0. At the tree root r, e_0 is the edge such that e_0 and its counter-clockwise predecessor e_d both lie on the outer face. Note that the edge list contains both tree and non-tree edges. We assign preorder and postorder numbers to the vertices of T according to a ***topological depth-first search.*** This is a depth-first search [48] in which each child of a vertex is visited according to the order in which it appears in the edge list. Figure 5.1 gives an example. Note that no edges can cross above r since, in our embedding, it lies on the outer face.

We denote the preorder and postorder numbers of v by $pre(v)$ and $post(v)$, respectively. It is well-known (see e.g. [49]) that for any pair u and v of vertices, v is an ancestor of u if and only if $pre(v) \leq pre(u)$ and $post(u) \leq post(v)$.

Let f be a non-tree edge $\{u, v\}$. We denote by $nca(f)$ the nearest common ancestor of u and v. We use $P(v, f)$ to denote the tree path from $nca(f)$ to v. Edge f is a potential critical edge for every vertex on the tree path connecting u and v except for their nearest common ancestor. The cycle formed by f taken together with the tree path between u and v separates the plane into two regions, one finite and one infinite. We call the finite

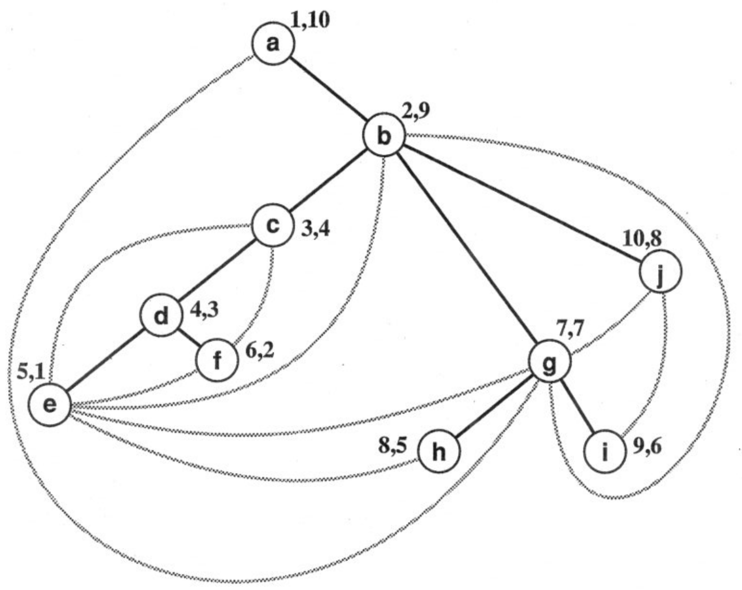


Figure 5.1: A planar graph with spanning tree rooted at a. Solid edges are tree edges. Vertices have been labeled by the preorder and postorder numbers given by a topological depth-first search. Edge $\{e, g\}$ is a right edge of e and a left edge of g. Edge $\{g, b\}$ is right edge of g and a dead edge of b.

region *interior* and denote it by $R(f)$. Given a tree edge $e = \{v, p(v)\}$ in the boundary cycle of $R(f)$, we say $R(f)$ is a left or right region of e if $R(f)$ lies to the left or right, respectively, of e as we look along e from v to $p(v)$.

The edges incident to a given vertex v are partitioned into three classes. Edge f is called a *dead edge* of v if $v = nca(f)$ or if $f = \{v, p(v)\}$. Note that every tree edge is a dead edge of some vertex. If f is not a dead edge of v, it is called a *left edge* of v if $R(f)$ is a left region of the tree edge $\{v, p(v)\}$, and a *right edge* of v otherwise. Thus a non-tree edge f is a left edge of one of its endpoints and is a right edge of the other, unless the endpoints are related, in which case f is a dead edge of the ancestor endpoint and a left or right edge of the descendent endpoint. Note that if f is a left edge of v, then $R(f)$ is a left region of all vertices on $P(v, f)$. The analogous property holds for right edges.

We can determine whether an edge $f = \{u, v\}$ is a dead, left, or right edge of v by using the topological preorder and postorder numbering in the following manner. If v is an ancestor of u or if $u = p(v)$, then f is dead. If u and v are unrelated then f is a left edge if $pre(u) < pre(v)$ and a right edge otherwise. If u is an ancestor (but not parent) of v, then let $e = \{u, x\}$ be the first tree edge following f in the edge list of u. If there is no such tree edge, or if $pre(x) > pre(v)$, then f is a right edge of v; otherwise f is a left edge. As mentioned above, ancestor queries can be answered in constant time using the preorder and postorder numbering.

Lemma 5.1 *Let f be a left edge of u and let g be a non-tree edge that precedes f in the edge list of u, i.e., is encountered before the tree edge $\{u, p(u)\}$ in clockwise order around u from f. Then g is also a left edge, and $nca(f)$ is a (not necessarily proper) ancestor of $nca(g)$.*

Proof. Since g lies between f and $\{u, p(u)\}$ in clockwise order, g must be contained wholly within the region $R(f)$, by the assumption of planarity. This implies that $R(g)$ is a left region of $\{u, p(u)\}$. Let $g = \{u, w\}$. Either w is on the boundary of $R(f)$ or lies within $R(f)$; i.e., is a proper descendent of a node on the boundary. Since all boundary nodes are descendents of $nca(f)$ by definition, $nca(g)$ must also be a descendent of $nca(f)$. □

The analogous result holds for right edges.

Let v be a leaf of T with edge list $\{e_0, f_1, f_2, \ldots, f_d\}$, where $e_0 = \{v, p(v)\}$ and $f_1 \ldots f_d$ are non-tree edges. We use $f_1, \ldots, f_d$ to denote non-dead edges. Since v is a leaf with no incident loop edges, e_0 is the only dead edge incident on v. Lemma 5.1 implies that there is an index ℓ, $0 \leq \ell \leq d$, such that the edges in $\{f_1, \ldots, f_\ell\}$ are left edges and the edges in $\{f_{\ell+1}, \ldots, f_d\}$ are right edges. Furthermore, let f_i and f_j be edges such that $1 \leq i < j \leq d$. Then $nca(f_j)$ is an ancestor of $nca(f_i)$ for $i, j \leq \ell$ (the left edges) while $nca(f_i)$ is an ancestor of $nca(f_j)$ for $i, j > \ell$ (the right edges).

Let $e = \{u, v\}$ be a tree edge, with $v = p(u)$. A *contraction* of e shrinks u into v leaving only the single vertex v. The new edge list of v is constructed by removing e from the list of u and v and inserting the edge list of u into the edge list of v at the position formerly occupied by e. Edge contraction preserves planarity [38, Lemma 1]. and the edge list produced by the contraction specifies a valid embedding. It may, however, produce new loop and multiple edges.

5.3 The Critical Edge Algorithm

The algorithm is based on the approach of Shier and Witzgall [42].

If $v \in T$ is a leaf its critical edge is simply the minimum-cost edge in its edge list, excluding the tree edge from v to its parent, which we will ignore from now on.

To compute the critical edges for the remaining vertices, we construct a series of graphs $G_0, G_1, \ldots, G_j$ with corresponding spanning trees $T_0, T_1, \ldots, T_j$, where G_0 is G minus its loop edges and j is the number of non-leaf vertices in G_0. Graph G_i is constructed from G_{i-1} by the following procedure:

1. Choose any vertex v in G_{i-1} all of whose children in T_{i-1} are leaves.

2. Contract the edges from v to its children and delete any resultant loop edges. This produces graph G_i and tree T_i in which v is a leaf.

3. Set *critical(v)* to be the minimum weight non-tree edge incident to v in G_i.

The correctness of the algorithm is proved in [42] and is easily seen. For a vertex $v \in G$, let the *relevant* edges, denoted $rel(v)$, be the set of non-tree edges with exactly one endpoint in the subtree of T rooted at v. By definition, $critical(v)$ is the minimum weight edge in $rel(v)$. A simple induction on i shows that when v becomes a leaf in stage i, its edge list contains exactly its relevant edges.

So far our algorithm does not particularly depend upon the planarity of G; this algorithm solves the critical edge problem in any general graph, and can be implemented in $O(m \log m)$ time using a mergeable heap data structure to store the edges at each vertex [30]. To improve the running time of the algorithm to $O(n)$ in the planar case, we take advantage of the properties discussed in Section 5.1.

Given G and T rooted at r, we first perform a topological depth-first search on T as described in Section 5.1, computing and storing preorder and postorder numbers and constructing the edge lists for each vertex. As part of this preprocessing, we determine for each edge whether it is dead, left, or right with respect to each of its endpoints. This requires two scans of the edge lists, one to determine for each non-tree edge the first subsequent tree edge in its edge list, and one to classify each edge using the constant-time test described in Section 5.1. These scans can be combined with the topological depth-first search. Loop edges can be removed at the same time.

Let v be a leaf with edge list $\{e_0, f_1, f_2, \ldots, f_d\}$. Lemma 5.1 implies that there is an index ℓ such that all edges f_1 to f_ℓ are left edges of v and all edges $f_{\ell+1}$ to f_d are right edges of v. Using ℓ we split the edge list into two lists $L_v = f_1, f_2, \ldots, f_\ell$ and $R_v = f_d, f_{d-1}, \ldots, f_{\ell+1}$. By Lemma 5.1 the edges in L_v and R_v are *nca-ordered*; i.e., if edges a and b belong to the same list and edge a precedes edge b, then $nca(a)$ is a descendant of $nca(b)$.

At each stage of the processing, our algorithm explicitly maintains the two nca-ordered lists L_v and R_v for each leaf node v. The lists for each leaf of the initial tree T are constructed during the preprocessing. The lists are maintained in a *heap-ordered concatenable queue* data structure that supports the following operations:

1. *make queue*(x): Create and return a new queue containing the single element x.

2. *pop*(q) : Delete and return the first item from queue q.

3. *concatenate*(q_1, q_2): Return the queue formed by concatenating q_2 to the back of q_1.

4. *find min*(q): Return the item of minimum weight in q without removing it from q. If q is empty return null.

5. *first*(q): Return the first element in q without removing it. If q is empty return null.

Let v be a non-leaf vertex of T processed in the i^{th} stage. Any child u of v must be a leaf, with left and right lists L_u and R_u, respectively. In Step 2 of the above procedure, the edges from v to its children are contracted, and two lists L_v and R_v are constructed from the non-tree edges incident to v and from the left and right lists of the children of v. The union of L_v and R_v is exactly the set of relevant edges for v. Step 3 is simply performed by finding the minimum of *find min*(L_v) and *find min*(R_v). Thus the bulk of the work occurs in Step 2.

Let $\{e_0, e_1, \ldots, e_d\}$ be the edge list of v. To begin Step 2, with each edge e_i, $i > 0$, we associate two lists L_i and R_i. If e_i is a tree edge $\{u, v\}$, where u is a child of v, then $L_i = L_u$ and $R_i = R_u$. If e_i is a left edge then $L_i = make\ queue(e_i)$ and $R_i = make\ queue(\emptyset)$. If e_i is a right edge then then $R_i = make\ queue(e_i)$ and $L_i = make\ queue(\emptyset)$. If e_i is a dead non-tree edge, i.e., if both endpoints of e_i are descendants of v, then both L_i and R_i are empty queues.

Next we delete the loop edges formed by the contractions. The loop edges are those edges f with $nca(f) = v$. Since each left or right list is nca-ordered, all such loop edges are grouped at the front of the lists. Each edge contained in a left or right list of v at this stage has as its nearest common ancestor either v or an ancestor of v. If f is an edge in one of the lists of v, then $nca(f) = v$ if and only if both endpoints of f are descendants of v. This can be tested in constant time by using the preorder and postorder numbers. as described in Section 5.1. Thus to delete loop edges, we simply examine each list and pop edges off until reaching the first edge whose nearest common ancestor is not v, i.e., the first that is a relevant edge of v.

After deleting loop edges, the collection of left and right lists contains only relevant edges. We form L_v by concatenating the left lists from left to right and form R_v by concatenating the right lists from right to left. That is, we form L_v by initializing L_v to the empty queue and then performing $L_v = concatenate(L_v, L_i)$ for $i = 1, 2, \ldots, d$. The same is done for R_v, except the index runs from d to 1. Code for the general processing step is given in Figure 5.2.

The edge list of v in G_i, after performing the concatenations, contains the edges of L_v in order followed by the edges of R_v in reverse order. Since G_i is planar and v is a leaf, Lemma 5.1 implies that L_v and R_v are nca-ordered and, further, that there is an index ℓ such that for $i < \ell$, R_i is empty and for $i > \ell$, L_i is empty.

Let d be the degree of v. Excluding the work involved in popping loop edges, the processing of vertex v requires $O(d)$ queue operations plus $O(d)$ additional work. The total number of pops is $O(n)$, since each edge is inserted into at most two queues and so can be popped at most twice. The preprocessing phase requires $O(n)$ time. If each queue operation takes $O(1)$ amortized time, then the total running time of the algorithm is $O(n)$.

We now describe the heap-ordered concatenable queue data structure.

```
ProcessVertex($v$ : vertex) begin
/* Let $v$ have edge list $\{e_0, f_1, f_2, \ldots, f_d\}$ */

/* Delete loop edges */
    for $i = 1$ to $d$ do begin
        if $f_i$ is a tree edge
            while $nca(first(L_i)) = v$ and $L_i \neq \emptyset$
                $pop(L_i)$;
            while $nca(first(R_i)) = v$ and $R_i \neq \emptyset$
                $pop(R_i)$;
    end
/* Initialize lists */
    for $i = 1$ to $d$ do begin
        if $f_i$ is a non-tree edge begin
            $L_i$ = make queue($\emptyset$);
            $R_i$ = make queue($\emptyset$);
            if $f_i$ is a left edge $L_i = concatenate(L_i, make\ queue(f_i))$;
            if $f_i$ is a right edge $R_i = concatenate(R_i, make\ queue(f_i))$;
        end
    end
/* Compute $L_v$ and $R_v$ */
    $L_v$ = make queue($\emptyset$);
    $R_v$ = make queue($\emptyset$);
    for $i = 1, 2, \ldots, d$ do $L_v = concatenate(L_v, L_i)$
    for $i = d, d-1, \ldots, 1$ do $R_v = concatenate(R_v, R_i)$
/* Compute critical(v) */
    set $critical(v) = min\{$find min$(L_v)$, find min$(R_v)\}$
end
```

Figure 5.2: Algorithm for processing one vertex. In the algorithm, $nca(e) = v$ if and only if $pre(u) \leq pre(v)$ and $post(u) \geq post(v)$ for both nodes u that are endpoints of e.

The data structure we present is a simple extension of ideas presented by Gajewska and Tarjan [25]. The items in queue q are stored in a linked list, with additional pointers to the front and back items. In order to answer the *find min* queries a second list r is maintained, consisting of the *rightward minima* of q. The first rightward minimum is the minimum element of the entire list q. The i^{th} rightward minimum is the minimum element occurring after the $(i-1)^{st}$. We store r as a doubly-linked list, with pointers to the first and last elements. To make a new queue containing item x, we initialize both lists to contain x. The operations *first*(q) and *find min*(q) are implemented in $O(1)$ worst-case time by using the pointer to the front of the appropriate linked list. The operations *pop* and *concatenate* are implemented as follows:

- *pop*(q) : Delete the first element from q and if it is also the first element of r, delete it from r.
- *concatenate*(q_1, q_2): Link the list of q_2 to the back of q_1. Let y be the first element of r_2. Remove the elements of r_1 from the last element forward, until reaching an element x with $cost(x) < cost(y)$. Link y to x, concatenating r_2 to the back of the modified r_1. Reset all pointers to first and last elements appropriately.

The worst-case time required for *pop* is $O(1)$. The time to perform a concatenation is $O(1)$ plus the number of items removed from the rightward minima list of the first queue. The removal of an element x is charged to the *make queue* that created x. Once x is removed from the minima list it can never be put back on it; x is removed because there is some other item y of lesser key following it in the queue, and y cannot be popped before x. Thus the amortized time per *make queue* and *concatenate* is $O(1)$. This in turn implies that the critical edge algorithm runs in time $O(n)$. The space required is also $O(n)$, since at any time each edge appears in at most two lists.

5.4 Minimum Spanning Tree Analysis

In this section we use the critical edge algorithm to solve the following problems in linear time: finding all replacement edges of a minimum spanning tree, verifying the minimality of a spanning tree, and performing sensitivity analysis on the edges of a minimum spanning tree. The following lemma will be useful.

Lemma 5.2 [22,46] *Tree T is a minimum spanning tree if and only if for each non-tree edge $f = \{u, v\}$, the cost of f is greater than or equal to the cost of each edge on the path from u to v.*

Let T be the minimum spanning tree of $G = (V, E)$. As defined in Section 5.1, the replacement edge of edge $e \in T$ is the non-tree edge f that replaces e in the minimum spanning tree of $G' = (V, E - \{e\})$. The removal of edge e breaks T into two fragments T' and T''. It is well-known ([46]) that the replacement edge f is the edge with minimum cost in the cut induced by (T', T''). We arbitrarily root T at some vertex r. Then for each tree edge $e = \{v, p(v)\}$, f is the minimum-cost edge with one endpoint in the subtree rooted at v. Thus to find replacement edges we run the critical edge algorithm and set the replacement edge of e to be $critical(v)$.

Lemma 5.2 implies that a tree T is a minimum spanning tree if and only if for each tree edge $e = \{v, p(v)\}$, the cost of $critical(v)$ is greater than or equal to the cost of e. This gives an algorithm for verification of a minimum spanning tree in a planar graph. An alternative method is to run the $O(n)$-time algorithm for computing minimum spanning trees of Cheriton and Tarjan [11], comparing the cost of the computed minimum spanning tree with cost of the given one.

To analyze the sensitivity of a minimum spanning tree T we determine for each edge e how much its cost can be perturbed before T is no longer minimal. We compute lower and upper bounds $[a, b]$ such that T remains minimal as long as $a \leq cost(e) \leq b$. If e is an edge in T, then the lower bound is $-\infty$. The above discussion implies that the upper limit is the cost of the replacement edge of e. Now consider a non-tree edge $f = \{u, v\}$. The upper

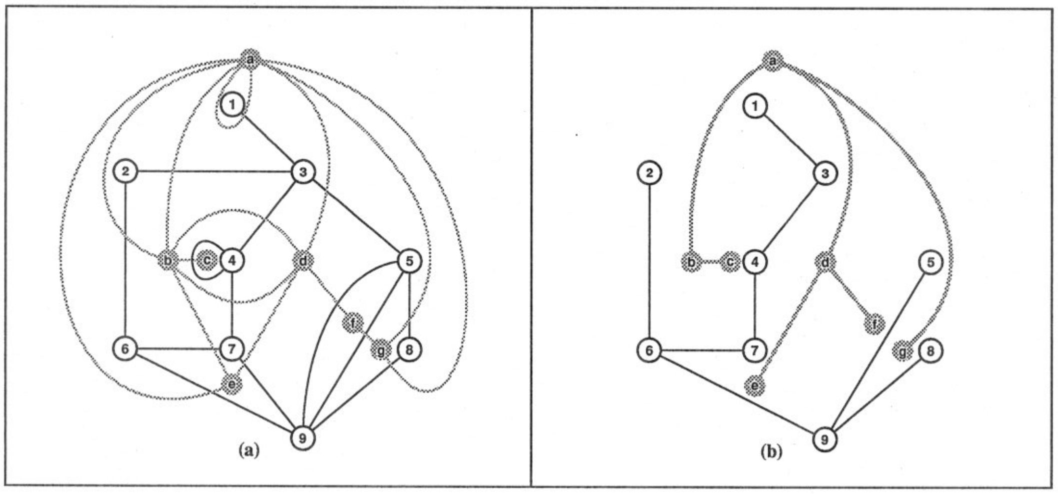


Figure 5.3: (a) A planar graph (bold) and its dual (shaded).
(b) Primal and dual spanning trees for the graph of (a).

limit for the cost of f is $+\infty$. By Lemma 5.2, the lower limit for $cost(f)$ is the cost of the maximum cost edge on the path from u to v. To compute the non-tree edge lower bounds in linear time we find critical edges in the *dual graph* $G^* = (V^*, E^*)$ of G.

Let S denote the planar subdivision given by the embedding of G. For each face in S there is a corresponding vertex in G^* and for each edge e in G there is a dual edge e^* connecting the dual vertices representing the two faces adjacent to e in S. Thus G^* is dependent on the embedding of G. The dual graph is clearly planar; an embedding S^* is given by placing each dual vertex inside the face it represents and placing each dual edge so that it crosses only the primal edge corresponding to it. We set $cost(e^*) = cost(e)$. Some representations of planar graphs simultaneously maintain both primal and dual graphs; in any case, given the embedding of G the dual can easily be computed in $O(n)$ time. Further discussion of dual graphs can be found in Harary [31]. An example is given in Figure 5.3.

Lemma 5.3 [20,24] *Given a spanning tree T in G, let T^* be the set of* *dual edges $\{e^*|e$ is not in $T\}$. The set T^* is a spanning tree for G^* and, furthermore, T is a minimum spanning tree for G if and only if T^* is a maximum spanning tree for G^*.*

Lemma 5.4 *Let $f = \{u, v\}$ be a non-tree edge in G with dual edge f^*. The replacement edge for f^* in G^* is the dual of the maximum-cost edge on the path between u and v in T.*

Proof. As in Section 5.1. let $R(f)$ denote the interior region of the plane bounded by f and the path in T from u to v. Removal of f^* breaks T^* into two fragments, $T^{*\prime}$ and $T^{*\prime\prime}$. All the vertices of one of these fragments (which are faces in the embedding of G) lie inside $R(f)$. Thus the edges in the cut $(T^{*\prime}, T^{*\prime\prime})$ are exactly the duals of the boundary edges of $R(f)$. The replacement edge for f^* is the minimum-weight edge in this cut. □

Lemma 5.4 implies that the lower limits for the non-tree edges can be computed by finding replacement edges in the dual graph and setting the lower bound for each non-tree edge to be the cost of its dual replacement edge. The result of this section are summarized in the following theorem.

Theorem 5.1 *The problems of computing replacement edges and determining the sensitivity of a minimum spanning tree of a planar graph can be solved in linear time and space.*

5.5 Shortest Path Tree Analysis

Let G be a directed graph whose edges each have an associated cost and let T be a single-source shortest path tree from source vertex s. In shortest path sensitivity analysis we are interested in finding bounds $[a, b]$ on the cost of each directed edge e such that, in the absence of other changes, T remains a shortest path tree for $a \leq cost(e) \leq b$.

Let $d(v)$ denote the distance from s to v, which is the sum of the costs of the edges on the path from s to v in T. The following lemma is well-known.

Lemma 5.5 [50] *A spanning tree T in G is a shortest path tree if and only if for all non-tree edges $e = (u, v)$, $d(u) + cost(e) \geq d(v)$.*

Let $e = (u, v)$ be a non-tree edge. By Lemma 5.5, T remains a shortest path tree for $(d(v) - d(u)) \leq cost(e) \leq +\infty$. Now consider a tree edge $e = (p(v), v)$. Adding Δ to $cost(e)$ changes the distances $d(v)$ for all nodes x in T_v, the subtree rooted at v. Lemma 5.5 implies that for T to remain a shortest path tree, Δ must satisfy the following constraints:

1. for each non-tree edge $f = (x, y)$ such that $x \in T_v$ and $y \notin T_v$ (the *outgoing* edges), $\Delta \geq d(y) - d(x) - cost(f)$
2. for each non-tree edge $f = (x, y)$ such that $y \in T_v$ and $x \notin T_v$ (the *incoming* edges), $\Delta \leq d(x) - d(y) + cost(f)$.

For each non-tree edge $f = (x, y)$ we compute a transformed cost $cost'(f) = cost(f) + d(x) - d(y)$. Then the lower bound on the cost of tree edge $e = (p(v), v)$ is $cost(e) - cost'(f_{out})$, where f_{out} is the edge going out from T_v of minimum transformed cost. The upper bound for e is $cost(e) + cost'(f_{in})$, where f_{in} is the edge coming in to T_v of minimum transformed cost. To compute f_{out} for each vertex v we initialize the edge lists of each vertex to contain only outgoing edges and run the critical edge algorithm, setting $f_{out} = critical(v)$. To compute f_{in} we initialize the edge lists to contain only incoming edges and again run the critical edge algorithm, setting $f_{in} = critical(v)$. (Note that these initializations preserve the planarity of G.)

Theorem 5.2 *Sensitivity analysis of a single-source shortest path tree in a planar graph can be performed in linear time and space.*

5.6 Minimum Cost Network Flow

We consider the network flow problem in which each edge e of the network G has upper and lower capacity bounds $[l, u]$ and a cost per unit flow through

e, and each vertex has a demand $D(v)$. If $D(v) > 0$, v is a *source*; if $D(v) < 0$, v is a *sink*. A minimum-cost flow in G assigns flow values $x(e)$ to the directed edges of G that satisfy the flow constraints and minimize the sum over all edges of $x(e)cost(e)$. Sensitivity analysis determines how much the edge costs can be perturbed without changing the optimality of the flow.

If there is any feasible flow there is an optimal flow with *basis* T; T is a spanning tree of G such that any non-tree edge has flow $x(e) = l(e)$ or $x(e) = u(e)$. There exists a price function on the nodes π such that the transformed cost of $e = (x, y)$, $cost'(e) = cost(e) + \pi(x) - \pi(y)$, is zero for all tree edges. The flow is optimal if and only if for all non-tree edges f, $cost'(f) \geq 0$ if $x(e) = l(e)$ and $cost'(f) \leq 0$ if $x(e) = u(e)$ [42].

If we root T at some vertex r, we can regard T as a shortest path tree from r, with $\pi(v)$ the distance from r in the tree, by replacing any edge e pointing up the tree by a reversed edge e' with cost $-cost(e)$. Then the sensitivity of the flow can be found by computing the sensitivity of the shortest path tree, reversing upper and lower bounds for reversed edges. Further details can be found in [42].

Theorem 5.3 *Sensitivity analysis of a minimum-cost network flow in a planar network can be performed in linear time and space.*

5.7 Remarks

We have considered here only planar graphs; it is natural to ask whether the same problems can be solved in general graphs in linear time. Dixon, Rauch, and Tarjan [17] have recently addressed this problem. They have a linear-time algorithm for verification of a minimum spanning tree in a general graph and a randomized linear-time algorithm for finding its replacement edges. In addition, they have discovered a deterministic algorithm for the replacement edge problem which is provably optimal although the running time is unknown. This leaves open the deterministic replacement edge problem. The best time bound known is $O(m\alpha(m, n))$ [46].

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman, *Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[2] G. Adelson-Velskii and E. Landis, "On an information organization algorithm," *Doklady Akademica Nauk SSSR*, 146 (1962), pp.263-266.

[3] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan, "Finding minimum-cost flows by double scaling," Technical Report CS-TR-164-88, Department of Computer Science, Princeton University, 1988.

[4] R. Bayer, "Symmetric binary B-trees: data structure and maintenance algorithms," *Acta Informatica*, 1 (1972), pp. 290-306.

[5] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," *Acta Informatica*, 1 (1972), pp. 173-179.

[6] W. W. Bein, P. Brucker, and A. Tamir, "Minimum cost flow algorithms for series-parallel networks," *Discrete Applied Mathematics*, 10 (1985) pp. 117-124.

[7] K. Booth and G. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms". *J. Comput. System Sci.*, 13:335–379, 1976.

[8] M. Brown and R. E. Tarjan, "A fast merging algorithm," *J. Assoc. Comput. Mach.*, 26 (1979) pp. 211-226.

[9] M.R. Brown and R. E. Tarjan, "Design and analysis of a data structure for representing sorted lists," *SIAM J. Comput.*, 9 (1980) pp. 594–614.

[10] B. Chazelle, "A theorem on polygon cutting with applications," *Proc.* 23^{rd} *Symposium on Foundations of Computer Science*, 1982, pp. 339-349.

[11] D. Cheriton and R. E. Tarjan, "Finding minimum spanning trees", *SIAM J. Comput.*, 5:724–742, 1976.

[12] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa, "A linear algorithm for embedding planar graphs using PQ-trees", *J. Comput. System Sci.*, 30:54–76, 1985.

[13] R. Cole and A. Siegel, "River routing every which way, but loose", *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, (1984), 65–73.

[14] R. Cole and U. Vishkin, "The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time," *Algorithmica*, 3 1988, pp. 329–346.

[15] P. F. Dietz and D. D. Sleator, "Two algorithms for maintaining order in a list," *Proc.* $19^{t}h$ *Annual Symposium on Theory of Computer Science*, (1987), pp. 365–372.

[16] R. Dial, F. Glover, D. Karney, D. Klingman, "A computational analysis of algernative algorithms and labeling techniques for finding shortest path trees," *Networks*, 9:215-248, 1979.

[17] B. Dixon, M. Rauch, and R. E. Tarjan, "Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time," To appear.

[18] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. Assoc. Comput. Mach.*, 19 (1972), pp. 248-264.

[19] D. Eppstein, "Finding the k smallest spanning trees", Manuscript, 1989.

[20] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a minimum spanning forest in a dynamic planar graph", *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, Submitted to SODA special issue of *Journal of Algorithms.*

[21] L. R. Ford, Jr. and D. R. Fulkerson *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

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

[23] H. N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking", In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, 434–443, 1990.

[24] H. N. Gabow and M. Stallmann, "Efficient algorithms for graphic matroid intersection and parity" (extended abstract), In *Automata, Languages, and Programming,* 12th *Colloquium, Lecture Notes in Computer Science, vol. 194*, 210–220. Springer-Verlag, Berlin, 1985.

[25] H. Gajewska and R. E. Tarjan, "Deques with heap order", *Inform. Process. Lett.*, 22:197–200, 1986.

[26] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan, "Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons," *Algorithmica*, 2 (1987), pp.209–233.

[27] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts "A new representation for linear lists," *Proc. Ninth Annual ACM Symposium on Theory of Computing*, (1977), pp.49-60.

[28] L. J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees," *Proc. Nineteenth Annual Symposium on Foundations of Computer Science*, (1978), pp. 8-21.

[29] L. J. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams", *ACM Trans. on Graphics*, 4:74–123, 1985.

[30] D. Gusfield, "A note on arc tolerances in sparse shortest path and network flow problems", *Networks*, 13:191–196, 1983.

[31] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA., 1972.

[32] D. S. Hirschberg and L. L. Larmore, "Efficient optimal pagination of scrolls", Techincal Report # 224, ICS Dept., Univ. of California at Irvine, CA, 1984.

[33] J. Hopcroft and R. E. Tarjan, "Efficient planarity testing", *J. Assoc. Comput. Mach.*, 21:549–568, 1974.

[34] S. Huddleston and K. Mehlhorn, "A new data structure for representing linear lists," *Acta Informatica* 17 (1982), pp. 157-184.

[35] S. Kannan, E. Lawler, and T. Warnow, "Determining the evolutionary tree," *Proc. First Annual Symposium on Discrete Algorithms.*, (1990), pp. 475-484.

[36] D. Knuth, *The Art of Computer Programming, Volume 1*, Addison Wesley, Reading, Massachusetts, 1973.

[37] S. R. Kosaraju, "Localized search in sorted lists," *Proc. Thirteenth Annual ACM Symposium on Theory of Computing*, 1981, pp. 62-69.

[38] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs", *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[39] D. Maier and C. Salveter, "Hysterical B-trees," *Inform. Proc. Lett.*, 12 (1981), pp. 199-202.

[40] K. Mehlhorn, *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching* Springer-Verlag, Berlin, 1984.

[41] J. B. Orlin, "A faster strongly polynomial minimum cost flow algorithm," *ACM Symp. on Theory of Computing*, (1988), pp. 377-387.

[42] D. R. Shier and G. Witzall, "Arc tolerances in shortest path and network flow problems", *Networks*, 10:277–291, 1980.

[43] D. D. Sleator, Personal communication.

[44] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, 32:3 (1985), pp. 652-686.

[45] R. E. Tarjan, "Amortized computational complexity," *Siam J. Alg. Disc. Methods*, 6, No. 2, (1985), pp. 306-318.

[46] R. E. Tarjan, "Applications of path compression on balanced trees", *J. Assoc. Comput. Mach.*, 26:690–715, October 1979.

[47] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[48] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, 1:146-160, 1972.

[49] R. E. Tarjan, "Finding dominators in directed graphs", *SIAM J. Comput.*, 3, 1974.

[50] R. E. Tarjan, "Sensitivity analysis of minimum spanning trees and shortest path trees", *Inform. Process. Lett.*, 14:30–33, 1982.

[51] R. E. Tarjan and C. J. Van Wyk, "An $O(n \log\log n)$-time algorithm for triangulating a simple polygon," *SIAM J. Comput.*, 17 (1988), pp. 143-178.

[52] A. K. Tsakalidis, "AVL-trees for localized search," *Lecture Notes in Computer Science 172*, J. Paredaens, ed., Springer-Verlag, Berlin, 1984, pp. 473-485.

[53] J. Valdes, R. E. Tarjan, and E. Lawler, "The recognition of series-parallel digraphs," *SIAM J. Comput.*, 11 (1982), pp.298-313.