Max-Min Tree Partitioning

YEHOSHUA PERL

Bar-Ilan University, Ramat Gan, Israel

AND

STEPHEN R. SCHACH

The Weizmann Institute of Science, Rehovot, Israel

ABSTRACT. The max-min k-partition algorithm may be formulated as follows: Given a tree T with n edges and a nonnegative weight associated with each vertex, assign a cut to each of k distinct edges of T so as to maximize the weight of the lightest resulting connected subtree. An algorithm for this problem is presented which initially assigns all k cuts to one edge incident with a terminal vertex of T; thereafter the cuts are shifted from edge to adjacent edge on the basis of local information. An efficient implementation with complexity $O(k^2 \cdot \mathrm{rd}(T) + kn)$, where $\mathrm{rd}(T)$ is the number of edges in the radius of T, is described. An algorithm for a simpler problem, namely, the partitioning of T into the maximum number of connected components whose weight is bounded below, is then described. Combined with the technique of binary search, it yields an alternative algorithm for the max-min k-partition problem with complexity dependent on the range of the given weights.

KEY WORDS: tree partitioning, max-min partition, shifting algorithm, binary search, complexity analysis, NP-completeness

CR CATEGORIES: 5.25, 5.32

1. *Introduction*

Let $T = (V, E)$ be an (unrooted) tree with n edges. We associate a nonnegative weight $w(v)$ with every vertex $v \in V$. A *q-partition* of T into q connected components $T_1, T_2, \ldots, T_q$ is obtained by deleting $k = q - 1$ edges of T, $1 \leq k \leq n$. The weight $W(T_i)$ of a component T_i is then the sum of the weights of its vertices.

In this paper we present algorithms for the following two partition problems:

(a) *Max-min q-partition*: Find a q-partition of T maximizing $\min_{1 \leq i \leq q} W(T_i)$.

(b) *Maximal partition bounded below*: For a given lower bound L, find a partition of T into the maximum number of components satisfying $W(T_i) \geq L$.

Solutions to these two problems can be applied to clustering techniques [11].

The work of the second author was performed while he was on leave from the University of Cape Town, Rondebosch, South Africa, and was supported in part by the South African Council for Scientific and Industrial Research.

Authors' addresses: Y. Perl, Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel; S.R. Schach, Department of Applied Mathematics, The Weizmann Institute of Science, Rehevot, Israel.

Two related problems, the second of which has received attention in the literature, are

(c) *Min–max q-partition*: Find a q-partition of T minimizing $\max_{1 \le i \le q} W(T_i)$.
(d) *Minimal partition bounded above*: For a given upper bound U, find a partition of T into the minimum number of components satisfying $W(T_i) \le U$.

Application areas for the latter two problems include paging and overlaying [12].

Hadlock [4] gives a polynomial algorithm for problem (d). Kundu and Misra [10] describe a linear algorithm for the same problem, for a rooted tree. But just as every such partition of a rooted tree induces a partition satisfying the same conditions for the corresponding unrooted tree (and vice versa), so an algorithm for solving any of the above four problems for a rooted tree (such as that of Kundu and Misra [10] for problem (d)) also provides a solution to the same problem for the corresponding unrooted tree. We use this property in our solutions for problems (a) and (b).

A q-partition of a general graph $G = (V, E)$ is defined to be a partition of V into q disjoint sets $V_1, V_2, \ldots, V_q$, such that the induced graphs $G_i = (V_i, E_i)$, $i = 1, 2, \ldots, q$, are connected.

Consider now the above four optimization problems applied to a general graph G and their respective corresponding decision problems. For example, the decision problem corresponding to the max–min q-partition of a graph G is, does there exist a q-partition of G such that the weight of each component is greater than or equal to a bound B? Even for an unweighted graph the optimization problems are NP-hard, since their corresponding decision problems are NP-complete [2, 7]. The reductions from the problem of *partitioning a graph into paths of length* 2 [3] are straightforward. This reduction is given for problem (d) in [3]. Turning to the partition of a graph into a *fixed* number of components q (say $q = 2$ or 3), the complexities of decision problems (a) and (c) for partitioning an *un*weighted graph into a fixed number of components are not known to us. However, for a general weighted graph we can show that both these problems are NP-complete. The reductions for $q = 2$ are from the *partition problem* [7] for weights $w_1, w_2, \ldots, w_n$ associated with the vertices. Similar reductions apply to decision problems (b) and (d) for a general weighted graph. An alternative approach in the case of weighted decision problem (d) is that of Hadlock [4], who used a reduction from the *bin packing problem* [3].

We present two algorithms for problem (a), namely, max–min partitioning of a tree T. The first algorithm, described in Section 2, employs a top-down approach. An arbitrary terminal vertex is chosen as the root of T. Initially all $k = q - 1$ cuts are assigned to the unique edge incident with the root. Cuts are then shifted, on the basis of local information, from edge to adjacent edge in a top-down direction, yielding local improvements until the optimal partition is finally obtained. An efficient implementation of the algorithm with complexity $O(k^2 \cdot \mathrm{rd}(T) + kn)$, where $\mathrm{rd}(T)$ is the number of edges in the radius of T, is presented in Section 3. In [1] we present a more complicated shifting algorithm for problem (c).

In Section 4 we first apply the bottom-up approach of end-order scanning [8] of the (rooted) tree T to obtain a linear algorithm for problem (b). Combining this algorithm with the technique of binary search [9] yields an alternate algorithm for problem (a). The complexity of the latter algorithm depends on the range of the given weights.

2. *The Max–Min q-Partition Algorithm*

Transform the given tree into a rooted tree by choosing an arbitrary terminal vertex as root and imposing a top-down direction on the edges. We choose a terminal vertex

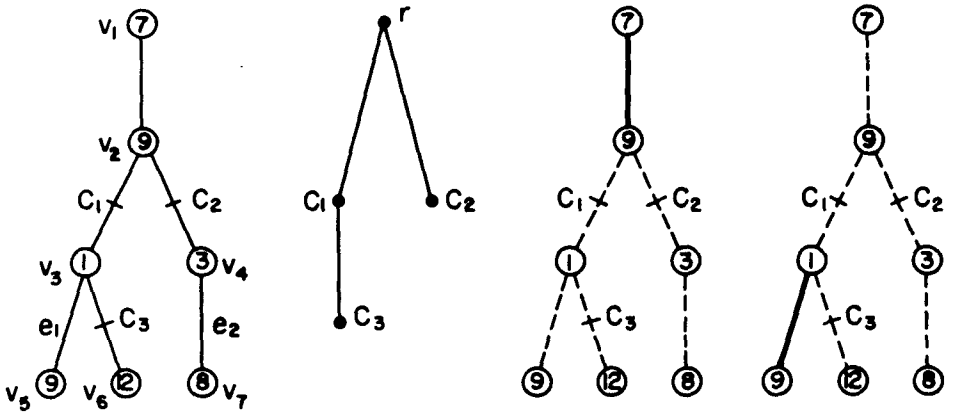

FIG. 1. (a) Tree T. (b) Cut tree C. (c) Root component. (d) Down component of C_1.

as root so that there is only one edge incident with the root. If $e = (v_1 \rightarrow v_2)$ is a directed edge, then we refer to v_1 as *tail*(e) and to v_2 as *head*(e). For convenience, if a cut c is assigned to an edge e, then we shall use head(c) and tail(c) for head(e) and tail(e), respectively.

A *shift* of a cut c is a transfer of c from an edge e_1 to an adjacent edge e_2 containing no cut, such that head(e_1) = tail(e_2); i.e., shifts are always top-down. This restriction on the permitted direction of shifting ensures that no cut can be assigned to any edge more than once and hence enables us to bound the complexity of our algorithm (see Section 3).

We require the notions of *partial* and *complete rooted subtrees*: A subtree T' of T is a partial (complete) subtree of T rooted at a vertex v if v is the root of T' and T' contains one (every) son of v, together with all the latter's descendents.

Let A be an arbitrary assignment of the k cuts to the edges of T. We define a *cut tree* $C = C(T, A)$ to be a rooted tree with $k + 1$ vertices representing r, the root of T, and the k cuts of A. A cut c_1 is the son of a cut c_2 if there exists a path from head(c_2) to tail(c_1) containing no cuts, and a cut c_1 is the son of r if there exists a path from r to tail(c_1) containing no cuts. A *down component* of a cut c in T is obtained from the complete subtree of T rooted at head(c) by deleting the complete subtrees rooted at the heads of the sons of c in C, if any. The *root component* of T is obtained from T by deleting the complete subtrees rooted at the heads of the sons of r in C. These definitions are illustrated in Figure 1. Throughout the remainder of this paper we say that one component T_i is *lighter* than another component T_j (T_j is *heavier* than T_i) if $W(T_i) < W(T_j)$.

Remark. Before presenting the algorithm we note that in the case of two cuts c_i and c_j assigned to the same edge as in the initialization of the algorithm below, we refer to one of the cuts, say c_j, as the son of c_i in the cut tree. We still refer, in such a case, to the partition as a q-partition; the definition of a down component implies that the down component of c_i contains no vertices and is of weight zero. The situation in the case of more than two cuts assigned to the same edge is similarly handled.

The Shifting Algorithm: Max–Min q-Partition of a Tree

1. Assign all k cuts to the *unique* edge incident with the root r.
2. Find the weight $W_{\min}$ of the current lightest component.

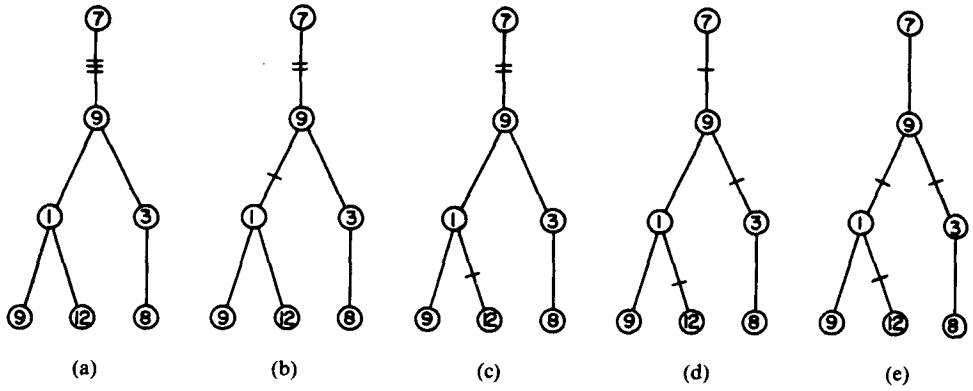

FIG. 2. (a) Initial assignment. (b) Step 1. (c) Step 2. (d) Step 3. (e) Step 4.

3. Find a shift of a cut c to an edge e containing no cuts, maximizing the weight RD_c of the resulting down component of the shifted cut c.
4. If $RD_c \geq W_{min}$, then perform the shift of the cut c to the edge e and return to step 2.
5. Terminate. The weight W_{min} is the weight of the lightest component of the max–min q-partition obtained.

Note that the above remark implies that the algorithm will never terminate while more than one cut is still assigned to the edge incident with the root.

We apply this algorithm to the problem of finding a max–min four-partition for the tree shown in Figure 1. The sequence of shifts is shown in Figure 2.

We show that the Shifting Algorithm yields a max–min q-partition of the rooted tree T and hence of the unrooted tree as well. For a given tree T there will in general be a number of different max–min q-partitions whose lightest components have weight W_{min}. For the purposes of the proof we want to consider only solutions to the max–min q-partition problem satisfying certain conditions. In particular, we wish to introduce the concept of a max–min q-partition P as being *close* to the final partition A obtained by the algorithm, in the sense that no reassignment of a cut of P results in a max–min q-partition which is closer to A. To this end we define a *maximal partial one-to-one correspondence* between the cuts of P and the cuts of A to be a one-to-one correspondence satisfying the following conditions:

(i) There is a directed path from a P-cut to its corresponding A-cut. (This path may be of length zero if both cuts coincide.)
(ii) No directed path between two corresponding cuts contains a noncorresponding cut.
(iii) No directed path between two corresponding cuts contains another pair of corresponding cuts.
(iv) The maximality of the partial one-to-one correspondence means that there is no directed path between noncorresponding P-cuts and A-cuts.

To construct such a correspondence, one initially identifies as many pairs as possible consisting of a P-cut and an A-cut such that there is a directed path from the P-cut to its corresponding A-cut. If condition (ii) is violated by having, say, a noncorresponding P-cut on the path connecting a P-cut to an A-cut, then the noncorresponding P-cut replaces the corresponding P-cut in the correspondence. If condition (iii) is violated in that a second corresponding pair lies on the path connecting a given corresponding pair, then the lower P-cut is associated with the

lower A-cut, and similarly with the upper cuts. We may now define the term "close." A max–min q-partition P is *close* to the algorithm partition A if for any maximal partial one-to-one correspondence set up between P and A, any one of the following operations results in a component which is strictly lighter than the previously lightest component of P (thus destroying the max–min q-partition property):

(i) Any shift of a P-cut toward its corresponding A-cut.
(ii) Any reassignment of a noncorresponding P-cut to an edge on the directed path to a noncorresponding A-cut making these two cuts into a corresponding pair. (This condition is required for the proof of case 2 of Lemma 2.1.)

It is possible to prove the existence of such a close partition. The proof uses maximality condition (iv) of the above correspondence in an essential way.

For brevity, we refer to a close max–min q-partition as an *optimal partition* and its cuts as *optimal cuts*, as opposed to an algorithm partition and algorithm cuts (which are shifted from edge to edge).

The following lemma is used in the validity proof of the algorithm.

LEMMA 2.1. *For any given tree T and any optimal partition, no rooted partial subtree of T can contain more algorithm cuts than optimal cuts during execution of the Shifting Algorithm.*

PROOF. The lemma is proved by contradiction. The initial assignment of the cuts to the edge incident with the root r satisfies the lemma, so assume that at some stage the shift of cut c from edge e_1 to edge e_2 causes a partial subtree rooted at a vertex $v = \text{head}(e_1)$ to become the first to violate the conditions of the lemma. Let A denote the algorithm partition of T just before performing this shift from e_1 to e_2.

By our assumption, at this point no rooted partial subtree of T contains more algorithm cuts of partition A than optimal cuts. We may therefore set up a one-to-one correspondence between the optimal cuts and the algorithm cuts satisfying the following two conditions. (Note the difference and the similarity between the definitions of a maximal partial one-to-one correspondence and of the one-to-one correspondence defined below.)

(i) There is a directed path (possibly of length zero) from an algorithm cut to the corresponding optimal cut.
(ii) No directed path between two corresponding cuts contains another pair of corresponding cuts.

We consider the various subcases.

Case 1. Edge e_1 is assigned its corresponding optimal cut (see Figure 3).

Case 1.1. The partition A coincides with the optimal partition. Changing the optimal partition by transferring the optimal cut from e_1 to e_2 yields a partition closer to the final partition obtained by the algorithm than the optimal partition. But by definition of an optimal partition, it must be close to the final partition of the algorithm. Hence a transfer of the optimal cut c from e_1 to e_2 yields a non-max–min q-partition; the down component of the cut assigned to e_2 is lighter than the lightest component in the optimal partition. Thus the down component of c in the algorithm partition after shifting it to e_2 is lighter than the lightest component of A, since A and the optimal partition coincide. Hence the algorithm should not have shifted c from e_1 to e_2, a contradiction.

Case 1.2. The partition A does not coincide with the optimal partition. The

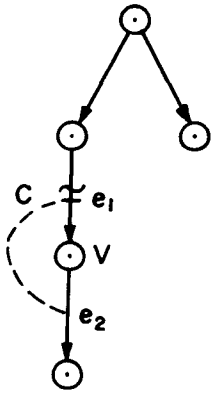

FIG. 3. Case 1 of Lemma 2.1. (The horizontal line denotes the algorithm cut; the tilde, the corresponding optimal cut.)

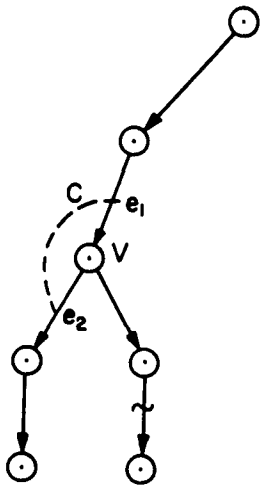

FIG. 4. Case 2 of Lemma 2.1. (The horizontal line denotes the algorithm cut; the tilde, the corresponding optimal cut.)

subtree rooted at v is the first during the execution of the algorithm to contain more algorithm cuts than optimal cuts. The one-to-one correspondence set up above implies a one-to-one correspondence between a subset of the optimal cuts and the algorithm cuts of A, which are represented by the sons of c in the cut tree (if such sons exist). Hence the down component of c in A is contained within the down component of the corresponding optimal cut; it is therefore no heavier than the latter.

As in Case 1.1, condition (i) of the closeness property of the optimal partition implies that the shifting of c from e_1 to e_2 yields a down component whose weight is smaller than the weight $W_{\min}$ of the lightest component of the optimal partition. (Note that this does not imply a contradiction; the weight of the lightest component of A may be smaller than $W_{\min}$ because we have not yet proved that A is a max–min q-partition.) But there exists a cut c' of A which does not coincide with its corresponding optimal cut, while all its descendents in the cut tree C do. A shift of c' toward its corresponding optimal cut therefore yields a down component of c' with weight greater than or equal to $W_{\min}$. Hence the algorithm will shift c' rather than c, a contradiction.

Case 2. Edge e_1 is not assigned its corresponding optimal cut (see Figure 4). A subtree T_1 rooted at $v = \text{head}(e_1)$ is the first partial subtree to contain more algorithm cuts than optimal cuts. Thus there must exist another partial subtree T_2 also rooted at v which contains more optimal cuts than algorithm cuts. By definition, the optimal partition is a max–min q-partition close to the final partition obtained by applying the Shifting Algorithm to T. Therefore, the fact that the extra optimal cut of T was not placed on the edge e_2 on the path from v to the head of the edge to which c is finally shifted implies, by condition (ii) of the closeness property, that the weight of the resulting down component of this optimal cut would then be smaller than $W_{\min}$. Hence, as in the proof of Case 1.2, the weight of the down component of c after its shift to e_2 is also less than $W_{\min}$.

At the same time there similarly exists a shift of a cut of A which results in a down component of weight not less than $W_{\min}$. If each of the algorithm cuts of subtree T_2 coincides with its corresponding optimal cut, then a shift of c along the path to the

extra optimal cut is such a move; otherwise, there exists such a shift of unmatched cut c'' in T_2, all of whose descendents in C coincide. In either case the algorithm would not have performed the shift of c to edge e_2, a contradiction. □

We can now show that the Shifting Algorithm yields a max–min q-partition of T.

VALIDITY PROOF OF THE SHIFTING ALGORITHM. The validity is proved by contradiction. Assume that when the algorithm terminates, the resulting partition A is not a max–min partition, i.e., the smallest partition has weight less than $W_{\min}$. Let B be an optimal partition of T, defined as in the proof of Lemma 2.1. By that lemma, no rooted partial subtree of T contains more algorithm cuts than optimal cuts. Thus, as in the proof of Lemma 2.1, there exists a one-to-one correspondence between algorithm cuts and optimal cuts.

By assumption, A contains a cut c which does not coincide with its corresponding optimal cut f in B, but all the descendents of c in the cut tree C do. Thus if c were to be shifted along the path to f, the resulting down component would have weight greater than or equal to the weight of the down component of f, which in turn is greater than or equal to $W_{\min}$. Thus the algorithm should not have terminated, a contradiction. □

3. *Complexity Analysis of the Algorithm*

The implementation of the Shifting Algorithm presented below requires at most $O(k^2 \cdot \mathrm{rd}(T) + kn)$ operations (where, as before, the tree T consists of n edges and is to be partitioned by means of k cuts). Before detailing the implementation we present the following lemma, which leads to a simplification in the algorithm.

LEMMA 3.1. *If the weight of the root satisfies $w(r) < W_{min}$, then before the Shifting Algorithm yields a max–min partition, no component of T is lighter than the root component (ignoring subpartitions of the edge incident with the root due to the initial assignment of the cuts).*

PROOF. The lemma is satisfied initially because the root component consists only of the root itself, and $w(r) < W_{\min}$. Consider the partition of T just after the first time that a shift of a cut c causes the resulting down component of c to be lighter than the root component. If c is not a son of the root r in C, then the weight of the resulting down component of c must be less than the weight of the root component before the shift. By the rules of the algorithm, the previous partition was optimal, and the algorithm terminated. Otherwise, if c is r's son, then the shift of c increased the weight of the root component. However, no shift can yield a resulting down component heavier than the down component of c, since otherwise that shift should have preceded the shift of c. Hence no shift can increase the weight of the lightest component, and therefore the partition obtained after the shift of c is a max–min partition of T. □

Now suppose we initially assign the cuts to a dummy terminal edge incident with the center [5] of the tree; let the (dummy) root vertex have zero weight, thereby satisfying the condition $w(r) < W_{\min}$ of the above lemma. Then (a) the length of the path traversed by any cut is bounded by the number of edges in the radius of the tree, and (b) there is no need to find the weight $W_{\min}$ of the current lightest component, since by Lemma 3.1 this is the root component (until possibly the final shift). Throughout this section the index i, $1 \le i \le n$, labels an edge e_i in the tree T. With regard to the cut tree C, for simplicity we denote the root r by c_0. The index j

TABLE I. VALUES OF DATA STRUCTURES L, F, MD AND ME FOR THE TREE IN FIGURE 1a

	c_0	c_1	c_2	c_3
L	c_1, c_2	c_3	nil	nil
F	—	c_0	c_0	c_1
MD	—	9	8	cannot move
ME	—	e_1	e_2	cannot move

TABLE II. ENTRIES OF DATA STRUCTURE P FOR THE TREE IN FIGURE 1a

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
c_1	v_2	v_3	—				
c_2	v_2	v_4		—			
c_3	v_2	v_3	v_6			—	

refers specifically to a cut (and thus $1 \leq j \leq k$); the index t, $0 \leq t \leq k$, then refers either to a cut c_j or to the root c_0.

Let $s(c_j)$ denote the number of sons of cut c_j in the cut tree C, and let $d(e_i)$ denote the outdegree of vertex head(e_i) in the tree T. For each vertex v in the tree T we compute the weight $Z(v)$ of the complete subtree of T rooted at v, that is to say, the sum of the weights of v and of all its descendents. We may compute $\{Z(v)\}$ once and for all by scanning the tree in end order [8]. As before, if a cut c_j is assigned to an edge e_i, for convenience we shall use $Z(c_j)$ for $Z(\text{head}(e_i))$.

During the execution of the algorithm we update the following data structures:

(1) For each vertex (cut) c_t in the cut tree C, we maintain a list L of the sons of c_t in C.
(2) We also require a pointer F from each c_j to its father in C.
(3) The weight of the root component is contained in $W_{\min}$, since, as explained above, this is the current lightest component during the execution of the algorithm (until possibly the final shift).
(4) At each step of the algorithm we have to consider every possible shift of each cut so that we can perform that shift which results in a down component of maximum weight. To this end, for each cut we store MD(c_j), the maximum weight of a resulting down component currently achievable by a shift of cut c_j.
(5) ME(c_j) is the edge through which c_j achieves this maximum.
(6) For each cut c_j we maintain the path in T from the root r to head(c_j). For this we keep a table P of order $k \times n$, such that if v_l lies on the path to cut c_j, then $P(j, l)$ contains the vertex following v_l on this path.

The values of L, F, MD, and ME for the tree illustrated in Figure 1a are given in Table I, and table P is displayed in Table II.

Initialization. The k cuts $\{c_j\}$ are initially assigned to the edge incident with the root c_0; the corresponding cut tree C is therefore a directed chain. The initialization of data structures $\{L\}$ and $\{F\}$ is thus straightforward. With regard to $W_{\min}$, we initially set $W_{\min} \leftarrow w(r)$. Initially MD(c_j) and ME(c_j), $j \neq k$, are undefined. For initialization of MD(c_k) consider the edges $\{e_i\}$ emanating from head(c_k). Clearly MD(c_k) = $\max_i Z(\text{head}(e_i))$, and ME($c_k$) is an edge for which the maximum is obtained. The number of operations required for the initialization is clearly $O(kn)$, since all entries of table P are initially set equal to "undefined."

Shifting. Step 3 of the algorithm requires finding a cut c_j of maximum MD(c_j). If

$\mathrm{MD}(c_j) \geq W_{\min}$, then c_j is shifted to $\mathrm{ME}(c_j)$; otherwise, the algorithm terminates. Thus $O(k)$ operations are required for a shift, since there are k cuts $\{c_j\}$.

Updating. Consider the effect on the sons of c_j in C of shifting a cut c_j from edge e_1 to edge e_2. Let C' denote the cut tree after this shift has been performed.

Case A. If, as a result of the shift, a cut c_h, $1 \leq h \leq k$, is no longer a son of c_j in C' (i.e., $P(h, \mathrm{tail}(e_2)) \neq \mathrm{head}(e_2)$), then delete c_h from the son list L of c_j, add it to the son list of $F(c_j)$, its father, and set $F(c_h) \leftarrow F(c_j)$.

Case B. If c_h is still a son of c_j (i.e., $P(h, \mathrm{tail}(e_2)) = \mathrm{head}(e_2)$), then no updating of L or F is required.

In both Cases A and B the cut c_j is shifted to edge e_2, so set

$$P(j, \mathrm{tail}(e_2)) = \mathrm{head}(e_2).$$

We now turn to the remaining data structures. At each step of the algorithm we must consider every possible shift of a cut in order to be able to select that shift which maximizes the weight of the resultant down component. But just as only two components are altered per shift, so we need to update only the maximum resulting down components $\mathrm{MD}(c_j)$ and $\mathrm{MD}(F(c_j))$, as well as their associated edges $\mathrm{ME}(c_j)$ and $\mathrm{ME}(F(c_j))$. Consider each possible shift of c_j to one of the edges e_p, $1 \leq p \leq d(e_2)$, emanating from $\mathrm{head}(e_2)$. For each such edge e_p we calculate the weight of the resulting down component $\mathrm{RD}_{e_p}(c_j)$, say, by subtracting from $Z(\mathrm{head}(e_p))$ the sum $\sum_h Z(c_h)$ over all sons c_h of c_j in the cut tree C'' obtained from C' by a shift of c_j from e_2 to e_p. In fact, we can compute simultaneously all $d(e_2)$ values of $\mathrm{RD}_{e_p}(c_j)$ in the following manner. Initially assign $\mathrm{RD}_{e_p}(c_j) \leftarrow Z(\mathrm{head}(e_p))$. Now scan the list L of sons of c_j in C', and for each cut c_h on this list identify the edge e_p on the path (in the tree T) from e_2 to the edge to which cut c_h has been assigned, using $P(c_h, \mathrm{head}(e_2))$, and then subtract $Z(c_h)$ from the appropriate $\mathrm{RD}_{e_p}(c_j)$, thus giving the value of each possible resulting down component. Then set $\mathrm{MD}(c_j) \leftarrow \max_{1 \leq p \leq d(e_2)} \mathrm{RD}_{e_p}(c_j)$. At this stage update $\mathrm{ME}(c_j)$, the corresponding edge e_p through which this maximum is achieved. If $c_0 \neq F(c_j)$, then we update $\mathrm{MD}(F(c_j))$ and $\mathrm{ME}(F(c_j))$ similarly. However, if $c_0 = F(c_j)$, then the root component is changed, and $W_{\min}$, rather than $\mathrm{MD}(F(c_j))$, has to be updated as above.

Complexity Analysis. Updates other than those of $\mathrm{MD}(c_k)$ described above require at most $O(k)$ operations per shift, because $s(c_j)$, the number of sons of cut c_j in C, is bounded by k. The total number of shifts during the execution of the algorithm is bounded by $k \cdot \mathrm{rd}(T)$, since the maximum length of the path any of the k cuts can traverse from the root is bounded by $\mathrm{rd}(T)$, the number of edges in the radius of T. Hence the above updates require at most $O(k^2 \cdot \mathrm{rd}(T))$ operations. The same result applies to the overall number of operations required for finding the cut to be shifted at each stage, as described above.

The update of $\mathrm{MD}(c_j)$ requires $O(k)$ operations for scanning the list L of sons of c_j and $O(d(e_2))$ for initializing $\mathrm{RD}_{e_p}(c_j)$, $1 \leq p \leq d(e_2)$. We separately analyse the contribution of each of these two components of updating $\mathrm{MD}(c_j)$ to the overall number of operations required for the algorithm.

Scanning the list L contributes $O(k^2 \cdot \mathrm{rd}(T))$ operations, as in the previous updates. The contribution of the second component, namely, initializing $\mathrm{RD}_{e_p}(c_j)$, is computed in a different way. Let $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$ denote the edges through which a cut c_j was shifted during the execution of the algorithm. While performing these shifts of c_j, $O(\sum_{j=1}^{m} d(e_{i_j}))$ operations were required for initializing $\mathrm{RD}_{e_p}(c_j)$. But in a tree of n

edges, we have that $\sum_{j=1}^{m} d(e_{i_j}) \leq n$. Hence the number of operations required for these initializations for all k cuts during the execution of the algorithm is bounded by $O(kn)$ (as was required for the initialization of the algorithm). Therefore, the Shifting Algorithm is of complexity $O(k^2 \cdot \mathrm{rd}(T) + kn)$.

4. *Maximal Partition Bounded Below*

In this section we consider the following problem: Find a partition of a tree T into the maximum number of connected components such that the weight of each component is greater than or equal to a given lower bound L. As mentioned in the introduction, we may choose an arbitrary vertex v as a root of T and use the solution we thus obtain for the rooted tree for the unrooted tree as well.

A partition of a rooted tree T into a maximum number of components bounded below by L is obtained as follows.

Scanning Algorithm

Scan the rooted tree in end order [8]. While scanning, calculate for each vertex the weight of the complete subtree of T rooted at u by adding the weights of the complete subtrees rooted at the sons of u. If there is encountered during the scan a node v for which the complete subtree rooted at v has weight greater than or equal to L, then delete the unique edge joining v to its father in T (unless the weight of the remainder is less than L, in which case terminate).

The proof of the validity of the above algorithm for a maximal partition bounded below is not difficult; we omit it for brevity. It is also obvious that the algorithm is linear. It is interesting to note that Kundu and Misra's algorithm [10] for obtaining a minimal partition bounded above is also linear, but only as a consequence of applying a linear algorithm for finding the median. This serves to demonstrate the difference in difficulty of two apparently similar problems.

The Scanning Algorithm can be used in an alternative approach to solving the max–min q-partition problem. Suppose we know the weight $W_{\min}$ of the lightest component of such a partition of T. Then applying the above algorithm with lower bound $L = W_{\min}$ until $q - 1$ cuts have been assigned yields a max–min q-partition in linear time.

By means of a binary search as applied in [6] we can find the value of $W_{\min}$ using the initial inequality $\min_{v \in T} w(v) \leq W_{\min} \leq W(T)/q$. The number of times the Scanning Algorithm is applied depends on the range of the given weights. For example, if the weights are integers bounded by c, then the algorithm is applied $O(\log nc)$ times in finding $W_{\min}$. Hence the complexity of the binary-search approach in finding a max–min q-partition is $O(n \log nc)$, since the algorithm for a maximal partition bounded below is linear. (Contrast this result with the complexity of the Shifting Algorithm of Section 2, which is independent of the weights assigned to the vertices.) For the case of equal weights, the binary-search approach yields a complexity $O(n \log n)$.

Comparing the two methods for finding a max–min q-partition, the binary-search approach is attractive when the number of cuts k is $O(n)$ (since the complexity is independent of k) and the range of the weights is small. On the other hand, the $O(k^2 \cdot \mathrm{rd}(T) + kn)$ algorithm of Section 2 is preferable for small values of k and $\mathrm{rd}(T)$.

The Scanning Algorithm for a maximal partition bounded below may also be used to obtain a better initial assignment of cuts for the Shifting Algorithm (for a max–min q-partition). Apply the Scanning Algorithm with lower bound $L = W(T)/q$ to the tree rooted at any terminal vertex. Suppose the resulting partition requires m

cuts. Then we know that $m \leq k$, since as stated above, $W_{\min} \leq W(T)/q$. If $m < k$, then the remaining $k - m$ cuts are assigned to the edge incident with the root. Using the proof techniques of Section 2, it can be shown that this arrangement can serve as a valid initial assignment for the Shifting Algorithm. It is obvious that the number of shifts required to reach an optimal partition is reduced, but unfortunately the complexity of the algorithm is not decreased by this stratagem. For this reason we chose in Section 2 to describe the Shifting Algorithm with the more straightforward initial assignment in order to simplify the presentation, despite the fact that in many examples the number of shifts required after initialization by means of the Scanning Algorithm was very small indeed.

REFERENCES

1. BECKER, R.I., PERL, Y., AND SCHACH, S.R. A shifting algorithm for min–max tree partitioning. To appear *J. ACM*.
2. COOK, S.A. The complexity of theorem-proving procedures. Proc. 3rd ACM Symp. on Theory of Computing, Shaker Heights, Ohio, 1971, pp. 151–158.
3. GAREY, M.R., AND JOHNSON, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.
4. HADLOCK, F. Minimum spanning forests of bounded trees. Proc. 5th Southeast Conf. on Combinatorics, Graph Theory, and Computing, 1974, pp. 449–460.
5. HARARY, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969, Ch. 4.
6. KARIV, O., AND HAKIMI, S.L. An algorithmic approach to network location problems: Part 1: The p-centres; Part 2: The p-medians. *SIAM J. Appl. Math. 37* (1979), 513–560.
7. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85–104.
8. KNUTH, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1969, Sec. 2.3.2.
9. KNUTH, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1969, Sec. 5.2.3.
10. KUNDU, S., AND MISRA, J. A linear tree partitioning algorithm. *SIAM J. Comput. 6,* 1 (1977), 151–154.
11. SALTON, G. *Dynamic Information and Librar Processing.* Prentice-Hall, New Jersey, 1975, Ch. 8.
12. TSICHRITZIS, D.C., AND BERNSTEIN, P.A. *Operating Systems.* Academic Press, New York, 1974, Ch. 4.

RECEIVED MAY 1979; REVISED FEBRUARY 1980; ACCEPTED FEBRUARY 1980