February, 1983

EFFICIENT IMPLEMENTATION OF A SHIFTING ALGORITHM

by

*Yehoshua Perl and **Uzi Vishkin

DCS-TR-124

*Rutgers University
New Brunswick, NJ 08903

**Courant Institute
New York University
251 Mercer Street
New York, NY 10012

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

and

Bar-Ilan University Ramat-Gan, Israel

ABSTRACT

An efficient implementation of the shifting algorithm ([BPS]) for min-max tree partitioning is given. The complexity is reduced from $O(Rk^3 + kn)$ to $O(Rk(k + \log d) + n)$ where a tree of n vertices, radius of R edges, and maximum degree d is partitioned into k + 1 subtrees. The improvement is mainly due to the new junction tree data structure which suggests a succinct representation for subsets of edges, of a given tree, that preserves the interrelation between the edges on the tree.

1. Introduction

A technique of shifting algorithms for tree partitioning problems was recently developed in a sequence of papers ([PS],[BPS],[BP],[ABP]). This technique is a top-down greedy technique rather than the bottom up approach used usually for such problems (see e.g., [KM],[KH]).

An efficient implementation of the shifting algorithm ([BPS]) for min-max tree partitioning is given. The complexity is reduced from $O(Rk^3 + kn)$ to $O(Rk(k + \log d) + n)$ where a tree of n vertices, radius of R edges, and maximum degree d is partitioned into k+1 subtrees.

The improvement is mainly due to a data structure which suggests a succinct representation for subsets of edges, of a given tree, that preserves the interrelation between the edges on the tree. Section 2 introduces both the min-max tree partitioning problem and high-level description of the shifting algorithm which was given in [BPS]. Section 3 presents an implementation scheme of the algorithm which forms an intermediate-level description of the algorithm. Section 4 introduces two data-structures. It is then shown how to utilize them by the intermediate level description to form a low-level implementation of the algorithm.

2. High-Level Description

Let T(V,E) be a tree where V is the set of vertices and E is the set of (undirected) edges. We associate a nonnegative weight w(v) with every vertex $v \in V$. A q-partition of the tree T into q connected components $T_1, T_2, \ldots, T_q$ is obtained by deleting i, $i \le k = q-1$ edges of T where $1 \le k < n$ and $n = |V|$. (The definition allows empty T_j, $1 \le j \le q$). The weight $W(T_i)$ of each component T_i is the sum of the weights of its vertices. Each T_i is a tree, hence we may refer to it as a subtree of T.

The min-max q-partition problem. Find a q-partition of T minimizing $\max_{1 \le i \le q} W(T_i)$

Definitions and Notations

For two vertices $u, v \in V$ define d(u,v), the distance between u and v in T, to be the number of edges in the loop-free path from u to v in T. The radius R of T is $\min_{\{u \in V\}} \max_{\{v \in V\}} (d(u,v))$. A vertex that yields the minimum in the definition of R is called a center of T. Add an auxiliary vertex r to T with w(r) = 0. Connect r to a center by an auxiliary edge. Transform the new tree into a rooted tree by choosing r to be the root and imposing a top-down direction on the edges. From now on we denote by T the new rooted tree.

In this paper we use the usual terminology of graph theory. If e is a directed edge incident from v_1 and incident to v_2, denoted by $v_1 \rightarrow v_2$, then we refer to v_1 as tail(e) and to v_2 as head(e). Edge e is said to be the father of edge e_1 if head(e) = tail(e_1), and in this case, e_1 is said to be the son of edge e. Edges e_1 and e_2 are said to be brothers if $tail(e_1) = tail(e_2)$.

Edge e_1 is said to be an ancestor (resp. descendent) of edge e_2 if there is a directed path from e_1 to e_2 (resp. e_2 to e_1). The definitions of father, son, descendent and ancestor extend in the obvious way to vertices. Consider a function $f:\{1,2,\ldots,k\} \rightarrow E$. Each such function represents the i, $i \le k$, edges being deleted from T in order to form a (k+1)-partition, where the definition of a q-partition extends to rooted trees in the obvious way. (The number i is the cardinality of the set $\{f(1),f(2),\ldots,f(k)\}$). Every (k+1)-partition of the rooted tree T induces a (k+1)-partition on the given undirected tree. Let $f(i) = e$ for i, $1 \le i \le k$, and edge $e = v_1 \rightarrow v_2$; then, we say that cut i is assigned to edge e and is incident from v_1. We call the function f a configuration of cuts (configuration, in short). A configuration g is said to be obtainable from a configuration f by a down-shift (resp. side-shift) of a cut i, $1 \le i \le k$, if $f(j) = g(j)$ for every $j \ne i$, $1 \le j \le k$, and f(i) is a father (resp. brother) of g(i).

We further need the notions of complete and partial rooted subtrees: A subtree T^1 of T is a complete (resp. partial) subtree of T rooted at vertex v (resp. edge $v \rightarrow u$) if v is the root of T^1 and T^1 contains every son (resp. the son u) of v together with all the descendents of these sons (resp. this son).

Given a configuration of cuts f let $v \rightarrow u_1$, $v \rightarrow u_2, \ldots$, $v \rightarrow u_a$ be all edges incident from vertex v which are not assigned cuts. The down-component of v is a subtree of the complete subtree of T rooted at v; its vertices include v and the vertices of the down-components of vertices $u_1,u_2,\ldots,u_a$. (Note, that this is a recursive definition). The down-component of a cut i is the down-component of head(f(i)), and i is called the top-cut of that component. The weight W(i) of a cut i is the weight of its down component. The down-component of an edge e is the down-component of head(e).

A component T_i is lighter than another component T_j (or T_j is heavier than T_i) if $W(T_i) < W(T_j)$.

The Shifting Algorithm

1. Initially, the k cuts are all assigned to the edge incident from r.

2. While The root component is not a heaviest component

   do

   2.1 Down-shift a cut i of heaviest down component to a vacant (i.e. not assigned any cut) son-edge of a heaviest down component. If no such vacant edge exists then halt.

   2.2 Let e be the edge assigned to the down-shifted cut i before the down-shift.

2.3 <u>While</u> The edge $e(=v_2 \rightarrow v_1)$ is vacant and $v_2 \neq r$

<u>do</u>

2.3.1 <u>if</u> there is a cut incident from v_2 such that its down component is lighter than the down component of v_1

<u>then</u> side-shift the lightest such cut to e.

2.3.2 $e \leftarrow$ the father edge of e

<u>od</u>

<u>od</u>

In words, the algorithm employs an external loop (instructions 2.1, 2.2 and 2.3) which contains an internal loop (instruction 2.3). Given a configuration of cuts let us describe the way it is changed by one application of the external loop. First, one of its cuts is down-shifted. Second, starting from the location of this cut we follow the path towards r (up the tree) until we first encounter an edge assigned a cut or arrive to r. Along this path side-shifts are performed. See the criteria for where to perform a down-shift or side-shifts in the algorithm itself.

In [BPS] it is shown that the configuration which implies the min-max partition is obtained before the algorithm halts. It is not necessarily the last partition.

3. Intermediate-Level Description

The <u>junction-tree data-structure</u> which is given in the next section represents the configurations obtained through the algorithm. It equips us with the ability to:

1. Update the junction tree in O(k) time whenever a down-shift occurs
2. Update the junction tree and compute in O(k) time all side-shifts required per each down-shift.

Now, we describe additional information which is kept and updated.

For each cut i, $1 \leq i \leq k$, we maintain the edge $f(i)$ $(=tail(f(i)) \rightarrow head(f(i))$ which is assigned to this cut by the present configuration f and the weight of its down-component denoted W(i).

We keep for each vertex $v \in V$, $Z(v)$ - the weight of the complete subtree rooted at v, and for each edge $e \in E$, $L(e)$ - the number of edges in the directed path from r to e called the level of $e(1 \leq L(e) \leq R+1)$. It takes O(n) time to compute Z(v) and L(e) for the entire tree.

For each $v \in V$ we keep and update a data-structure called cut-free(v). It is described in the next section. This data-structure represents the edges $e=v \rightarrow u$ such that the partial subtree rooted at e contains no cuts, and satisfies the following:

1. Each delete operation takes $O(\log d_v)$ time where d_v is the number of sons of v in T.
2. Each insert operation takes O(l) time.
3. It enables us to get such an edge e of maximum Z(head(e)) in O(1) time.

It takes O(n) time to initialize this data-structure.

In addition to that we keep a cut-level array of size k(R+1) which is initially undefined. The entry (i,h) will include the h-th edge on the directed path from r to the edge currently assigned to cut i.

Let us overview the implementation of the shifting algorithm.

Theorem 1. Each application of the external while loop of the shifting algorithm takes O(k+logd) time, where d is the maximum degree of a vertex in T.

Remark: We give in this section a proof which includes some unproved statements related to the junction tree and cut-free data structures. These statements are proved in the next section.

Proof: There are k cuts. We choose in O(k) time the cut i of maximum W(i) to be down-shifted in instruction 2.1. To which edge should we down-shift cut? An edge $v_1 \rightarrow v_2$ with down-component of maximum weight is chosen among the candidates suggested by the cut-free(v_1) data-structure (v_1=head(f(i)) where f is the configuration before the down-shift) and candidates representing the partial subtrees containing cuts. The candidate of the first kind and its weight are obtainable in O(1) time. All the candidates of the second kind and their weights are obtainable in O(k) time. To see this we introduce an auxiliary vector weight(e) for each $e \in E$. Initialize weight(e) to z(head(e)) before each applicaiton of the following procedure. Upon termination of this loop all the entries of this vector changed during the loop contain weights of candidates of the second kind.

```
for each j, 1≤j≤k
do  e←cut-level(j,L(f(i))+1)
    if tail(e)=v1
    then weight(e) ←weight(e) - W(j)
od
```

Explanation: The cut-level array includes for each cut the information whether it passed through v_1 and if yes through which edge incident from v_1 it proceeded.

The remaining details in order to complete the proof that a down-shift takes O(k) time do not require new ideas and are therefore omitted. Note, however, that if the candidate suggested by the cut-free(v_1) data-structure is chosen we need to delete it, implying additional $O(\log d_{v_1})$ time. So instruction 2.1 takes $O(k + \log d_{v_1})$ time. The next lemma deals with the internal loop and thus completes the proof of the theorem for the external loop.

Lemma 1. The total time required by invocations of the internal loop per each application of the external loop of the algorithm is O(k).

Proof: There are at most O(k) side-shifts (in the internal loop) per each down-shift of instruction 2.1, because each cut can be side-shifted at most once as we 'climb up' the tree towards the root. The next section presents an implementation in which we make up to O(k) stops on our path up the tree to check for possible side-shifts. In this implementation the total amount of time for update and computation of the required weights of down-components does not exceed O(k). This is due to the second property of the junction-tree data structure mentioned above. Both the proof of the theorem and the lemma are completed at this point provided that we present the promised properties of the two data-structures.

4. Low-level description

Given a configuration f we define J(f), the junction tree of f as follows. It is a rooted tree. The set of vertices contains the set of cuts {1,2,..., k}. In addition to that it contains vertices $v \in V$ such that v is a lowest common ancester of two or more edges of the set {f(1), f(2),..., f(k)}. The latter are called junction vertices while the former are called cut vertices. There is an edge from a cut vertex i to a cut vertex j (resp. junction vertex v) if there is a directed path from f(i) to f(j) (resp. v) in T that does not include any other vertex of J(f). There is an edge from a junction vertex v to a cut vertex i (resp. a junction vertex u) if there is a directed path from v to f(i) (resp. u) in T that does not include any other vertex of J(f). Figure 1(a) describes an example of T and f. Figure 1(b) gives J(f). Note that a cut vertex has at most one son and a junction tree has at most 2k−1 vertices.

Let us show that any down-shift or all side-shifts per down shift can be reflected on the junction tree data structure in O(k) time.

Below, we use the convention that f(resp. g) is the configuration of cuts before (resp. after) the described shift. The letters v (resp. i, resp. α, β) represent junction vertices (resp. cut vertices representing cuts being shifted, resp. portions of a junction tree) of J(f) or J(g). Let us describe possible changes in J(f) following one down-shift. Figure 2 describes the case where v=head(f(i)) is a junction vertex (see Figure 2(a)). There are two possibilities: either i is down-shifted to an edge of T on the path from v to some cut

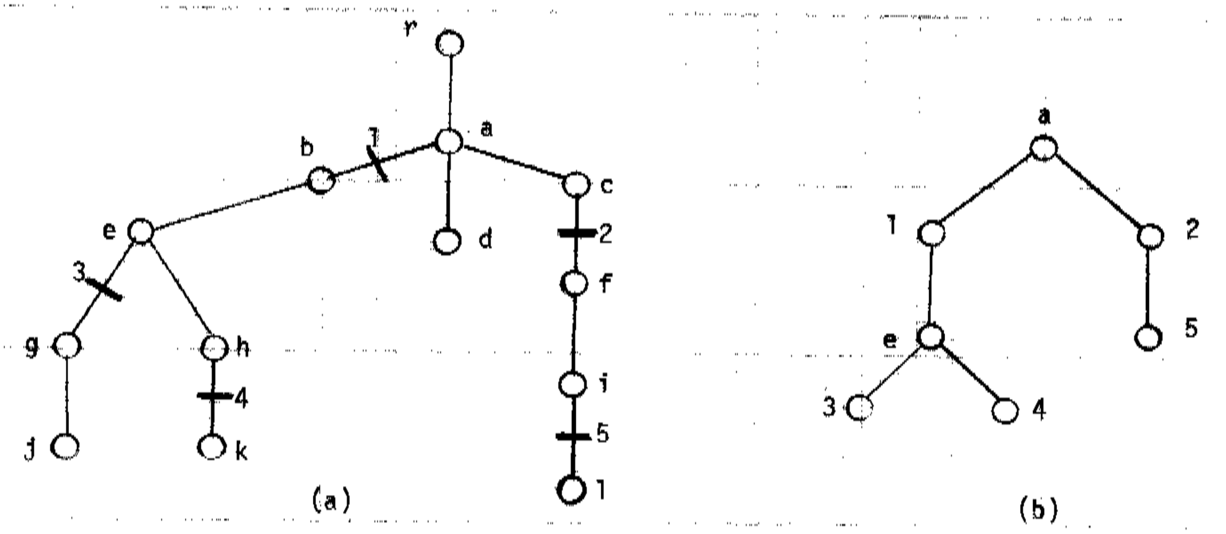

FIGURE 1

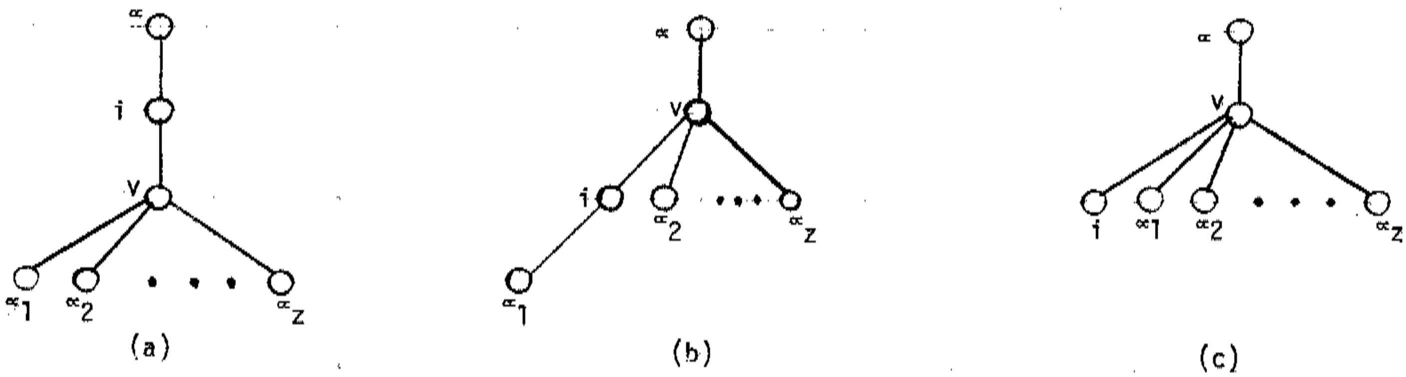

FIGURE 2

(Figure 2(b)) or not (Figure 2(c)). Figure 3 describes the case where head (f(i)) is not a junction vertex (see Figure 3(a)). Figure 3(b) describes, actually, two possibilities: the junction tree portion β is empty or it is not empty and cut i was down-shifted to the next edge in T on the path from f(i) to the (vertex or edge) in T represented by the son of i in J(f). Figure 3(c) describes the other case of Figure 3. Note that in the last case v=head(f(i)) becomes a junction vertex.

How to actually perform each such change in O(k) time? Notice that in the cases described in Figures 2(b) and 3(b) (when β is not empty) the edge to which the cut is down-shifted is not taken from the cut-free data-structure. The computation of weights of the candidates required for these cases can be slightly modified to yield the edge in J(f) to which cut-vertex i is moved: start from a cut vertex that changed weight (g(i)- the down shifted cut) in the implementation of instruction 2.1 described in the proof of Theorem 1. Follow the path up J(f) in order to identify the edge of J(f) to which cut vertex i is moved to form J(g). The other changes required in this case are simple. In all the other cases of down-shifts cut vertex i 'creates' a new edge in J(g) which is trivial to implement.

The possible changes in J(f) following a side-shift are described in figures 4 and 5. Figure 4 describes the case where cut vertex i does not have descendents in J(f) and junction vertex v=tail(f(i)) has only two sons (including i). This causes the elimination of junction vertex v in J(g), see Figure 4(b). Figure 5 includes the other cases: at least one of the junction tree portions α_2 and α_3 is not empty. The result is seen in Figure 5(b). The implementation here is trivial assuming that we know which side-shift to perform. We see below that we use the junction-tree structure to show us which side-shifts to perform.

<u>Implementation of the internal loop</u>

We keep an auxiliary variable W which is set to zero before each application of instruction 2.2. Instead of climbing on T we climb on the junction tree. At each point, before climbing further on the junction tree, W is set to equal the sum of weights of all down components of cuts which are assigned to edge-descendents of our present position in the climbing. We initialize W at the down-shifted cut. For instance, in the case described in Figure 2(a) W is initially the sum of all weights of down-components of cuts that belong to the junction tree portions α_1, α_2,...,α_3 and the weight of the down component of cut i. The implementation of the internal loop goes then as follows.

Let vertex u denote our present position in the climbing. Vertex u belongs to the junction-tree data-structure. Initially u is the father, in the junction-tree, of the last down-shifted cut.

<u>While</u> u is a junction vertex

<u>do</u>

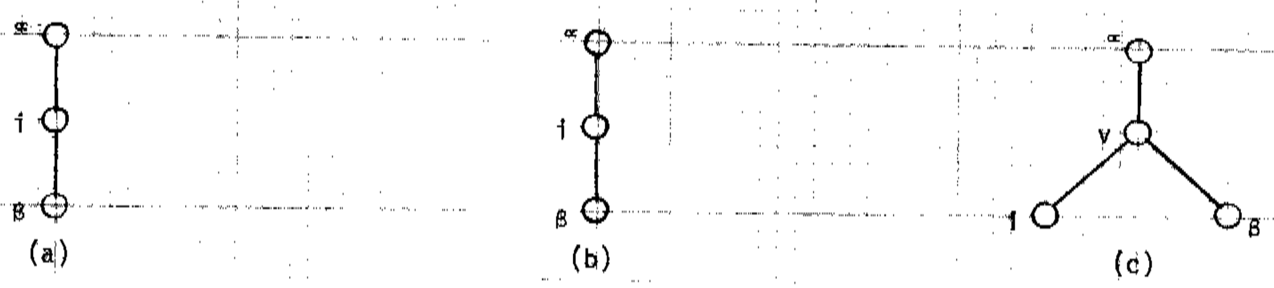

FIGURE 3

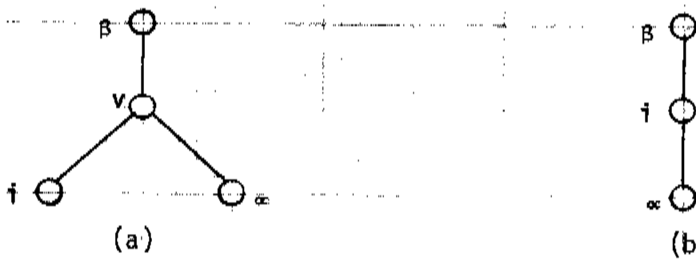

FIGURE 4

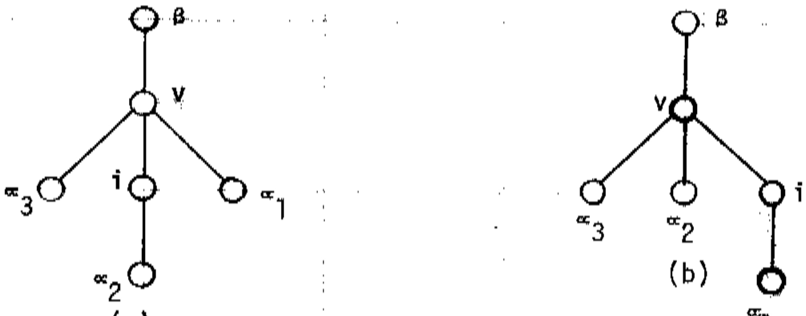

FIGURE 5

(let u⟶t be the edge in the junction tree on which we climbed to u; let u⟶v be the first edge in the path in T, which corresponds to u⟶t in the junction tree. The weight of the down-component rooted at v is Z(u)−W)

<u>if</u> there is a cut vertex j which is a son of u in the junction tree, f(j) is incident from u in T and the weight of cut j is lighter than Z(v)−W

<u>then</u> the lightest such cut is chosen and it is side-shifted to u⟶v, and W⟵Z(u) + ΣW(j). (The sum is over cuts j which were descendents of u, but not v, before the last side-shift, not including the cut which was side-shifted above)

<u>else</u> W⟵W+ΣW(j) (The sum is over cuts j which are descendent of u, but not v).

<u>If</u> u is the root of the junction tree

<u>then</u> go to the next performance of the external loop

<u>else</u> u⟵ the father of u in the junction tree

<u>od</u>

(we arrive at a cut vertex i) W(i)⟵Z(head (f(i))) − W.

<u>Some remarks have to be added.</u>

1. The connection between locations in the junction tree and their corresponding locations in T is done by using at any time the level L in conjunction with the row of the cut-level array corresponding to the cut which was down-shifted.

2. Since we never go back to a portion of the junction-tree which was visited before, the computation time is bounded by time which is proportional to the size of the junction tree−O(k). Note that during the computation of W we only visit new edges of the junction tree (due to the reservation: 'descendents of u, but not v').

<u>The cut-free data-structure</u>

The cut-free(v) (for every v ∈ V) data-structure is a priority queue. It is actually a combination of two standard data-structures: a heap and a stack. Initially the heap part of cut-free(v) contains all the edges v⟶u in T. The heap is organized according to Z(u) with the son edge of heaviest Z(u) at the top. It takes 0(n) operations to initialize the heaps, of all vertices in T.

To remind the reader, the cut-free(v) data-structure represents the edges v⟶u in T for which the partial subtree rooted at v⟶u contains no cuts. Our heap contains initially d_v

elements where d_v is the number of sons of v, v≠r in T. The deletion of an element from the heap and the update that follows takes $O(\lg d_v)$ time.

We still have to consider the possibility of returning an edge v⟶u to cut-free(v) while a side-shift of a cut from (v,u) is performed and the partial subtree rooted at v⟶u contains no cuts. We add a stack to the cut-free data-structure to contain such edges. Say that a down-shift of a cut i on the edge w⟶v is decided, and a choice to which edge to down-shift should be made. For this we have to consider the candidate of the stack as well as the candidate of the heap of cut-free(v). Any access to a stack for insertion or deletion purposes require O(1). The following lemma shows that the top of the stack of cut-free(v) contains a heaviest partial subtree.

Lemma 2: The edges v⟶u in the stack of cut-free(v) are stored in non-decreasing order of the weights Z(u) of the partial subtrees which are rooted at them.

Proof. Suppose a cut i_1 was side-shifted from an edge $v \to u_1$ which was then inserted into the stack and later another cut i_2 was side-shifted from an edge $v \to u_2$ which was then inserted into the stack. If i_2 was at the edge $v \to u_2$ when i_1 was side-shifted from $v \to u_1$ then by the shifting algorithm $Z(u_1) \le Z(u_2)$. Otherwise, i_2 was down-shifted to $v \to u_2$ while $v \to u_1$ was vacant. Again by the shifting algorithm $Z(u_1) \le Z(u_2)$. Hence the order of storing edges in the stack satisfies the Lemma. | |

The number of down-shift operations of the external loop is bounded by kR. Therefore Theorem 1 implies that the running time of the present implementation of the min-max shifting algorithm is O(n+Rk(k+log d)).

Acknowledgement

We wish to thank D. Coppersmith for helpful technical discussions, and one of the referees for suggestions regarding the presentation of the results.

References

[ABP] Agasi E., Becker R. I. and Perl Y., "A shifting algorithm for min-max constrained partition on trees', to appear in the Proceedings of the 16th annual CISS Conference, Princeton 1982.

[BPS] Becker R. I., Perl Y. and Schach S. R., "A shifting algorithm for min-max tree partitioning", JACM 29(1982) 58-67

[BP] Becker R. I. and Perl Y., "Shifting algorithms for tree partitioning with general weighting functions", Proceedings of the 19th annual Allerton Conference on Communication Control and Computing 1981, 746-751. Also to appear in J. of Algorithms.

[KH] Kariv O. and Hakimi S. L., "An algorithmic approach to network location problems. Part 1: The p-centers. Part 2: The p-melians", SIAM J. Appl. Math. 37(1979) 513-560.

[KM] Kundu S. and Misra J., "A linear tree partitioning algorithm", SIAM J. Computing

6(1977) 151–154.

[PS] Perl Y. and Schach S. R., "Max-min tree partitioning", JACM 28(1981) 5–15.