An $O(n \log n)$ Version of the Averbakh-Berman Algorithm for the Robust Median of a Tree*

Gerth Stølting Brodal[1], Loukas Georgiadis[2], and Irit Katriel[3]

[1] BRICS, University of Aarhus, Århus, Denmark.
[2] HP Labs, Palo Alto, CA.
[3] Brown University, Providence, RI.

Abstract. We show that the minmax regret median of a tree can be found in $O(n \log n)$ time. This is obtained by a modification of Averbakh and Berman's $O(n \log^2 n)$-time algorithm: We design a dynamic solution to their bottleneck subproblem of finding the middle of every root-leaf path in a tree.

Keywords: Minmax Regret, Tree Median, Robust Optimization.

1 Introduction

Facility location problems deal with the question of deciding where to construct one or more facilities so as to minimize the communication (or travel) costs while providing service to customers. One of the most basic forms of this problem is that of selecting the median of a tree: The input is an n-node tree $T = (V, E)$ with a weight w_v for each node and a positive length (distance) d_e for each edge $e \in E$. The nodes represent the customers, their weights are their demands for service and the distance between two locations on the tree represents the travel or communication cost between them. By a *location* on the tree, we mean either a node or a position along an edge, specified by its distances from the edge's endpoints. The *median* of the tree is the location x that minimizes the weighted sum of the distances from the nodes to x, i.e., the value $\sum_{v \in V} w_v d(x, v)$, where $d(x, v)$ is the distance between x and v in T. Finding the median can be done in linear time [6].

In the *minmax regret* version of this problem, there is uncertainty in the weights of the nodes. That is, the weight of v is only known to lie within an interval $[w_v^-, w_v^+]$. The input is then a collection of many possible *scenarios* and the problem is to find a solution that minimizes the maximum (over the scenarios) of the difference between the value of the solution on a scenario and the value of the optimal solution for the same scenario. In other words, the goal is to minimize the amount by which we will regret our choice of median once the actual input is known. Several researches have studied the problem of selecting

* Corresponding Author: Irit Katriel, Computer Science Department, Brown University, 115 Waterman st., Providence, RI 02912. irit@cs.brown.edu

the minmax regret median in this setting, beginning with Kouvelis et al. [8] who introduced the problem and suggested an $O(n^4)$ solution, through Chen and Lin [4] who obtained an $O(n^3)$ upper bound to Averbakh and Berman who found a simple $O(n^2)$ algorithm [2] and later improved upon it and achieved an $O(n \log^2 n)$ upper bound [3]. We will show that it is possible to implement Averbakh and Berman's algorithm in $O(n \log n)$ time. Our improvement is based on a dynamic version of their bottleneck sub-problem. Recently, Yu et al. [10] have independently achieved the $O(n \log n)$ bound by a different approach, which we believe leads to a significantly more complex solution. It remains open to bridge the gap between this upper bound and the (trivial) linear lower bound.

2 Problem Statement and Preliminaries

Let $T = (V, E)$ be a tree with $|V| = n$ nodes. We also use T to denote the set of all locations (nodes or locations along an edge) in T. Each node $v \in V$ is associated with an interval of weights $W(v) = [w_v^-, w_v^+]$, and each edge $e \in E$ is associated with a positive length d_e. We let d denote the (symmetric) distance function on T: For any $x, y \in T$, $d(x, y)$ is the distance between x and y in T; for an edge $e = \{u, v\}$, $d(u, v) = d(v, u) = d_e$ and for two locations x and y on the tree, $d(x, y)$ is the sum of the lengths of edges on the simple path between x and y on the tree. If x (y) lies along an edge e, the first (last) term in this sum is the appropriate fraction of d_e.

Now let S be the Cartesian product of all $W(v)$, $v \in V$. A *scenario* $s \in S$ assigns to each node v a particular weight $w_v \in W(v)$. For a given point $x \in T$ and scenario $s \in S$, define $F(s, x) = \sum_{v \in V} w_v d(v, x)$. For a given point $x \in T$, the *worst-case regret* for x is defined by the equation

$$Z(x) = \max_{s \in S} \max_{y \in T} \{F(s, x) - F(s, y)\}. \tag{1}$$

Let $(\hat{s}(x), \hat{y}(x)) \in S \times T$ be a pair that maximizes the right hand side of (1). Then, $\hat{s}(x)$ is a *worst-case scenario* for x and $\hat{y}(x)$ is a *worst-case alternative* for x.

The *Minmax Regret Median* problem on T is to find a point $x \in T$ that minimizes $Z(x)$.

3 The Averbakh-Berman Algorithm

In this section we sketch the $O(n \log^2 n)$ solution by Averbakh and Berman [3]. The algorithm consists of two main parts. The first identifies the edge on which the solution lies and the second identifies the precise location along this edge. The second part takes linear time, hence we will not refer to it any further (see [3], Section 4).

The first part performs a recursive search on the tree ([3], Section 3). At the i'th recursive level, a *pivot* node x_i is selected and used to determine the part of the tree in which the solution lies. Before giving the details of this search we

need to establish some notation. For a node $x \in V$ the *branches* of x in T are the subtrees B_j of T which are formed as follows: First remove x from T; this partitions $T \setminus \{x\}$ into connected subtrees C_j, where each C_j contains a node y_j such that $\{x, y_j\} \in E$. Then, B_j is formed by including x and $\{x, y_j\}$ in C_j. For any point $y \in T$, we let $B(x, y)$ denote the branch of x containing y.

Averbakh and Berman prove the following key lemma:

Lemma 1 ([3], Lemma 1). *Let x be a node of the tree and let $(\hat{s}(x), \hat{y}(x))$ be a pair that maximizes the right hand side of* (1)*. If $x = \hat{y}(x)$ then x is the optimal solution, i.e., x minimizes $Z(x)$. Otherwise, the solution belongs to $B(x, \hat{y}(x))$.*

This lemma suggests a recursive search procedure. Let $T_1 = T$. In the i'th recursive step, a node x_i is selected and used to restrict the search for a solution to the tree $T_{i+1} = T_i \cap B(x_i, \hat{y}(x_i))$. The recursion terminates when $x_i = \hat{y}(x_i)$ or T_i consists of a single edge (in which case the optimal location along this edge is found by the second part of the algorithm). By selecting x_i to be the *centroid* of T_i, we have that the depth of the recursion is $O(\log n)$. The centroid of an n-node tree T is a node $x \in T$ such that any branch of x contains at most δn nodes, where $\delta = 3/4$; it easy to find such a node in linear time [7].

Finally, Averbakh and Berman describe how the worst-case alternative $\hat{y}(x_i)$ of the pivot x_i can be found in $O(n \log n)$ time. Note that even when we know that the solution to our original problem is guaranteed to be contained in T_i, the worst-case alternative $\hat{y}(x_i)$ of the pivot $x_i \in T_i$ may be located in $T \setminus T_i$. This means that the search for $\hat{y}(x_i)$ must consider all nodes of T, i.e., this part of the problem does not reduce in size as the recursion proceeds.

Their algorithm for computing the worst-case alternative of x_i is based on the follow idea. Instead of iterating over the scenarios, they iterate over the nodes of the tree. For each node $y \in V$, they compute $\rho(x_i, y) = \max_{s \in S}\{F(s, y) - F(s, x_i)\}$. The scenario s_y that maximizes $\rho(x_i, y)$ is easy to specify: the weight of a node v is w_v^- if $d(x_i, v) \le d(y, v)$ and w_v^+ otherwise.

The bottleneck of a single worst-case alternative computation is in solving the problem *Middle*(T, r), which is defined as follows:

Input: A tree T rooted at a node r with a distance d_e for each edge e.
Output: For each node v of T, the *middle node* of v, denoted by *mid*(v), which is the highest ancestor u of v that satisfies $d(r, u) > d(u, v)$.

I.e., for every node we need to identify the middle of the path from this node to the root. For a node y, scenario s_y switches at *mid*(y) between the w_v^+ and w_v^- values. Averbakh and Berman use the following method for computing the middle nodes: Perform a DFS traversal of the tree, and whenever a node is visited for the first time, find its middle node by a binary search on the current search path. This process spends $O(\log n)$ time per node, hence $O(n \log n)$ per recursive level and $O(n \log^2 n)$ for the whole algorithm.

Given the middle points with respect to x_i, Averbakh and Berman show how to determine the $\rho(x_i, y)$ values for all $y \in V$ in $O(n)$ time ([3], Lemma 2). To

that end, they use the following auxiliary values

$$W^+(B) = \sum_{v \in B} w_v^+, \qquad W^-(B) = \sum_{v \in B} w_v^-,$$
$$D^+(B,x) = \sum_{v \in B} w_v^+ d(v,x), \qquad D^-(B,x) = \sum_{v \in B} w_v^- d(v,x),$$

for each node x and each branch B of x. These values can be pre-computed in linear time via dynamic programming. Once computed, they can be used to determine all $\rho(x_i, y)$ values by a single, linear-time traversal of the tree. The node y which maximizes $\rho(x_i, y)$ is the worst-case alternative of x_i.

4 Our Improvements

In order to eliminate a logarithmic factor from the running time of the Averbakh-Berman algorithm, we will show that the middle nodes can be updated dynamically in linear time, for all but a constant fraction of T_i. We begin by describing an alternative method for computing the middle nodes. The advantage of this method is that the middle nodes can be updated fast when we choose a different node to root T.

4.1 Finding the middle nodes from scratch

We reduce the problem of finding the middle nodes to a disjoint set union (DSU) problem [1], after a sorting phase. The algorithm traverses the nodes by decreasing order of their distance from the root r, and computes the middle node for each of them in turn:

1. Compute the distance of each node of T from the root r by a DFS traversal starting at r.
2. Sort the nodes of T by non-increasing distance from the root. Let the sorted list of distances be $U = (d_1, d_2, \ldots, d_n)$, where $d_i \geq d_{i+1}$. Each record u_i in U has a pointer to its corresponding node v_i in T. (This list represents union operations on the DSU data structure.)
3. Create another list F of the half-distances to r, i.e., $F = (d_1/2, d_2/2, \ldots, d_n/2)$, where each record has a pointer to its corresponding node v_i. (This list represents find operations on the DSU data structure.)
4. Merge U and F to a list L of event points; in case of a tie between a $u \in U$ and an $f \in F$, we give priority to f (i.e., the find operation is performed before the union). In order to distinguish between elements of different lists we mark the items in F.
5. Initialize a DSU data structure on the tree rooted at r; each node v forms a singleton set with v as the representative element.
6. Perform a sweep over the event points of L (in non-increasing order). Let d be the current event point corresponding to node v. If d is unmarked then

unite the sets of the children of v with v; this forms a single set containing all descendants of v, with v as the representative element. If, on the other hand, d is marked, perform a find on v, which returns the representative element w of the set containing v; assign $mid(v) = w$.

Lemma 2. *For each node v, the find operation on v returns $mid(v)$.*

Proof. At the moment the find on v needs to be performed, v is contained in a set Σ that includes all ancestors of v at distance strictly greater than $d(r, v)/2$ from r; let w be the highest such ancestor of v. Step 6 guarantees that w is the representative of Σ and $d(r, w) > d(w, v)$. Also, any proper ancestor u of w is not in Σ and therefore satisfies $d(r, u) \leq d(u, v)$. Thus, $w = mid(v)$. □

Excluding the sorting step, the running time of this procedure is dominated by the time required for the DSU operations. With a standard DSU data structure we get an $O(n\alpha(n))$ running time [9] (where α is an extremely slow-growing functional inverse of Ackermann's function). However, since the structure of the unite operations is given by the tree T, which is static, the DSU operations can be performed in linear total time using the DSU data structure by Gabow and Tarjan [5].

4.2 A dynamic version

Let $\hat{T}$ be a subtree of T, rooted at a node r. Suppose that we already computed the middle nodes for $\hat{T}$ by the procedure described above and saved the sorted list U. Now, assume that we modify $\hat{T}$ by adding a new root r' and a path from r' to r (see Figure 1). We wish to recompute the middle nodes with respect to the new root. For the nodes that were in $\hat{T}$ before, the relative distances from the new root are unchanged. The new nodes are closer to r' than all other nodes and their relative distances are determined by their location along the path from r' to r (recall that the distances are all positive). Hence, we can update U and create the new event list L in linear time. Thus, we skip the sorting phase in Step 2 of the algorithm of Section 4.1, and simply perform the sweep over the new list L to compute the new middle nodes in linear time.

In level i of the recursion we will use this dynamic version to recompute the middle nodes for the nodes in $T \setminus T_i$; the operation of adding a new root to a tree corresponds to the selection of a new pivot in the Averbakh-Berman algorithm.

4.3 Speeding up the Averbakh-Berman algorithm

The search at the i'th level of recursion ($1 \leq i \leq c \log n$, for some constant c) divides the current tree T_i into a *white* subtree $T_i^w = T_{i+1}$ where the solution lies, and a *black* subtree T_i^b. The two subtrees have a common root, which is the current pivot x_i, but are otherwise disjoint. (It is convenient to have the root of the black subtree be a white node, since otherwise we would have to refer to a collection of $O(n)$ black trees.) Also, the sizes of the two subtrees satisfy $|T_i^w| \leq \delta^i n$ and $|T_i^b| \leq |T_{i-1}^w| \leq \delta^{i-1} n$.

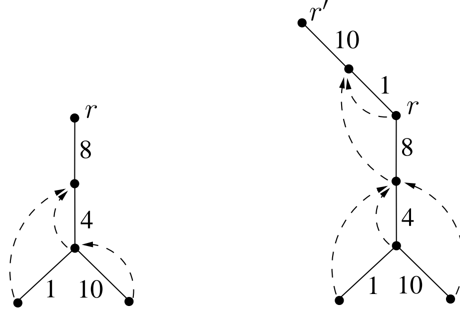

Fig. 1. Left: A tree rooted at r with a dashed arrow from each node v to the middle node $mid(v)$ (omitted when $v = mid(v)$). Right: The middle nodes after adding a new root r' and a path from r' to r.

Note that at the end of recursive level i, the tree T is divided into one white subtree T_i^w and i black subtrees $T_1^b, \ldots, T_i^b$, where T_j^b is the set of nodes excluded by the j'th level of the recursion. Any two of these trees may overlap at a single node.

Our goal is to use the procedure of Section 4.1 to compute the middle nodes in the white tree T_i^w, and the method of Section 4.2 to update the middle nodes for all T_j^b, $1 \leq j \leq i$. More precisely, we will create a single event list for all nodes in T by merging the event lists corresponding to T_i^w (that contains the current pivot) and to each T_j^b, but the sorting phase will be required only for the list of T_i^w. To that end, for each black subtree T_j^b we keep an event list $L(T_j^b)$, where the distances in that list are relative to the root of T_j^b. The important observation here is that the order of the event points in such a list does not change as we move the root of T to the new pivot, since the new pivot is always in the white subtree. Hence, we can update $L(T_j^b)$, with respect to the new pivot, as described in Section 4.2. Next, we need to merge all $L(T_j^b)$, and finally merge the resulting list with the event list of T_i^w. The last merging step takes $O(n)$ time, so it remains to show how to merge the lists of the black subtrees in linear time. We describe two methods to achieve this; both give the same asymptotic bound but the second maintains a simpler partition of T.

First method. As we already pointed out, in the original version of the Averbakh-Berman algorithm, the black subtrees T_j^b, $1 \leq j \leq i$, may be unconnected (i.e., any path connecting them traverses at least one white node), while some of them may overlap at previous pivot nodes. Fortunately, we can guarantee a linear bound for merging the event lists $L(T_j^b)$ because the size of each new black tree is reduced by a factor of δ (at least), i.e., $|T_j^b| \leq \delta |T_{j-1}^b|$. Hence, if we merge the event lists successively starting from the smallest black subtree to the largest (i.e., by decreasing j), the total time to create a single event list for the union

of the black subtrees is bounded by

$$\sum_{j=1}^{i} j|T_j^b| \leq n \sum_{j=1}^{\infty} j\delta^{j-1} \leq n\frac{1}{(1-\delta)^2} = 16n ,$$

since $\delta = 3/4$. It is not hard to implement this idea, but it requires some care so that the merge operations are performed in the correct order. The next method avoids this complication.

Second method. For our alternative method we make the observation that it is possible to bound the number of black subtrees, at the end of each recursive level, to be at most two. The tradeoff is increasing the number of computations for worst-case alternatives, by a factor of at most two. This is done as follows. Assume, without loss of generality, that the current partition of T consists of two unconnected black subtrees T_1^b and T_2^b, and a white subtree T^w. Let r_1 be the root of T_1^b and r_2 the root of T_2^b (i.e., r_1 and r_2 are two past pivots). Let r_3 be the next pivot node in T^w. The search for $\hat{y}(r_3)$ first roots T at r_3; let T_{r_3} denote this rooted tree, and let z be the nearest common ancestor in T_{r_3} of r_1 and r_2. Because we assumed that T_1^b and T_2^b are unconnected, $z \in T^w$.

We consider the possible locations of $\hat{y}(r_3)$ relative to z. If $\hat{y}(r_3) \notin B(r_3, z)$ then z belongs to the new black subtree T_3^b of T, and hence all three black subtrees will become connected, forming a single black subtree T' rooted at r_3. On the other hand, if $\hat{y}(r_3) \in B(r_3, z)$ then T_1^b, T_2^b and T_3^b are unconnected (see Figure 2). We can now join at least two of them as follows. We select z as the next pivot and compute $\hat{y}(z)$. If $\hat{y}(z)$ is not in any of $B(z, r_i)$ (for $i = 1, 2, 3$), then again a single black subtree T' is formed. Otherwise, assume $\hat{y}(z) \in B(z, r_i)$, for some $i \in \{1, 2, 3\}$, and let j and k be the indices of the other two black-subtree roots. In this case T_j^b and T_k^b become connected in a black subtree T' rooted at z. Whenever we connect two or three black subtrees we merge their event lists creating a single list $L(T')$, where the distances are relative to the current pivot (either r_3 or z) which is the root of T', as required.

Thus, the algorithm to compute a worst-case alternative for the pivot is applied at most twice at each recursive call, once to decrease the size of the white subtree by a factor of δ and the second time to ensure that at most two black subtrees remain.

4.4 Running time

Putting everything together, we have that at each level i, the middle nodes of only n_i tree nodes need to be computed from scratch, where $n_i \leq \delta^{i-1}n$; this takes $O(n_i \log n_i)$ time. For the remaining $n - n_i$ nodes this computation takes linear time. Hence, the total time spent on all $mid(v)$ computations throughout the recursion is bounded by

$$\sum_{i=0}^{c \log n} (n - n_i + n_i \log n_i) = O(n \log n).$$

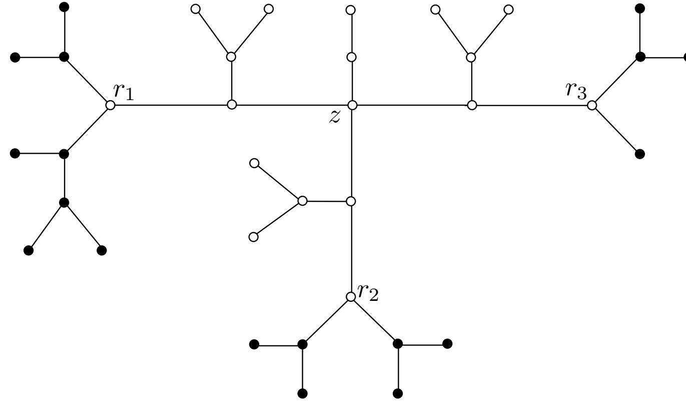

Fig. 2. Reducing the number of black subtrees. There are three black subtrees rooted at r_1, r_2 and r_3. Node r_3 is the current pivot and $\hat{y}(r_3)$ is in $B(r_3, z)$, where z is the nearest common ancestor of r_1 and r_2 in T_{r_3}.

Acknowledgements

We thank Kristoffer Arnsfelt Hansen for helpful discussions. This work was done while Loukas Georgiadis and Irit Katriel were at the University of Aarhus. Gerth Stølting Brodal is supported by the Danish Natural Science Research Council (grant # 21-04-0389). Irit Katriel was supported by the Danish Research Agency (grant # 272-05-0081). BRICS, Basic Research in Computer Science, funded by the Danish National Research Foundation.

References

1. A. V. Aho, J. E. Hopcroft, and J. D.Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Series in Computer Science and Information Processing, MA, 1974.
2. I. Averbakh and O. Berman. Minmax regret median location on a network under uncertainty. *INFORMS J. on Computing*, 12(2):104–110, 2000.
3. I. Averbakh and O. Berman. An improved algorithm for the minmax regret median problem on a tree. *Networks*, 41(2):97–103, 2003.
4. B. Chen and C. Lin. Minmax-regret robust one-median location on a tree. *Networks*, 31:93–103, 1998.
5. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.
6. A. J. Goldman. Optimal center location in simple networks. *Transportation Sci.*, 5:212–221, 1971.
7. O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. II: The p-medians. *SIAM Journal on Applied Mathematics*, 37(3):539–560, 1979.
8. P. Kouvelis, G. L. Vairaktarakis, and G. Yu. Robust 1-median location on a tree in the presence of demand and transportation cost uncertainty. Working

Paper 93/94-3-4, Department of Management Science and Information Systems, Graduate School of Business, The University of Texas, Austin.
9. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
10. H.I. Yu, T.-C. Lin, and B.-F. Wang. Improved algorithms for the minmax regret 1-median problem. In D. Z. Chen and D. T. Lee, editors, *Computing and Combinatorics, 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006, Proceedings*, volume 4112 of *Lecture Notes in Computer Science*, pages 52–62. Springer, 2006.