ALGORITHMS FOR P-CENTER IN A WEIGHTED TREE

by

O. Kariv and M. Jeger

Technical Report #249

August 1982
ABSTRACT

Algorithms of \( O(p n \log n) \) time to find an (absolute or vertex \( p \)-center on a weighted tree are represented.
1. INTRODUCTION

Let $T(V,E)$ be an undirected tree with $n$ vertices, and assign a positive length ($l_e$) to each edge $e \in E$ and a non-negative weight $w(v)$ with every vertex $v \in V$.

Let $SP$ represent a set of supply points and $OM$ a set of demand points on the tree. Throughout this paper we assume $OM = V$. The domination radius with respect to the demand set $V$ and supply set $SP$ is defined as follows:

$$r(SP,V) = \max \{ \min_{v \in V} w(v) \cdot d(v,x) \}$$

The weighted $P$-center problem is to find an optimal location for $P$ supply points (centers) such that $r$ is minimized.

We distinguish between two types of problems in the weighted tree: The vertex $P$-center ($V|V|P$) in which centers are confined to lie on vertices only, and the absolute $P$-center ($E|V|P$) in which centers may be located also on the internal points of the edges.

[KH] solve the weighted one (vertex or absolute) center problem on a tree in $O(n \log n)$ time. In a recent paper by Meggido [M] an $O(n)$ algorithm to the problem is presented.

The inverse of the center problem, namely, finding a minimum (absolute or vertex) dominating set of radius $r$ is solved by [KH] on a tree in $O(n)$ time. [KH] use that solution in order to find a $P$-center on a tree by computing all possible values of the domination radii $(w(v_i) \cdot d(v_i,v_j) \in V|V|P$ and $\frac{w(v_i) \cdot w(v_j)}{w(v_i)+w(v_j)} \cdot d(v_i,v_j) \in E|V|P$) and using binary search method for finding the optimal $r$. Their algorithms require $O(n^2 \log n)$ time on a tree.
Recently [MTZC] have developed an algorithm for finding the k-th longest path in an unweighted tree which avoids the need for computing all internodal distances. Their algorithm consists of generating a representation of internodal distances in terms of Cartesian matrices. They apply that result in order to perform a binary search on the internode paths and thus find an optimal P-center.

Their solution for \( V|V|P \) and \( E|V|P \) in the unweighted tree case and for the \( V|V|P \) in the weighted tree requires \( O(n \log^2 n) \) time.

[FJ] have improved the algorithm of [MTZC] achieving \( O(n \cdot \log n) \) time for the unweighted tree case.

A recent algorithm developed by [MT], solves \( E|V|P \) in the weighted tree in \( O(n \log^2 n \log(\log n)) \).

In this paper we discuss the \( V|V|P \) and the \( E|V|P \) problem in the weighted tree case. Our algorithm solves those problems in \( O(p \cdot n \log n) \) time, which is an improvement over the best previous result of [MTZC] and [MT] for \( P < \log n \) and for \( P < \log n \cdot \log(\log n) \) respectively. A key procedure for the algorithms is given in Section 2. The algorithms for \( V|V|P \) and \( E|V|P \) are described in Sections 3 and 4 respectively.

2.

Definition: An Inter Domain Edge (IDE), in a weighted tree \( T \), is an edge \((v_1, v_2)\) which satisfies the following conditions:

Let \( T_2 \) be the subtree which is obtained from \( T \) by deleting \((v_1, v_2)\) such that \( v_2 \in T_2 \); let \( r \) be the (vertex or absolute) P-radius of the whole tree, and let \( r_L \) be the radius required to cover \( T_2 \) from a center located in \( v_2 \) and \( r_U \) be the radius required to cover \( T_2 \) from a center located in \( v_1 \). Then the edge 

\((v_1, v_2)\) is an IDE if

\[ r_L < r < r_U \]
Lemma 1: If in a weighted tree $T$ $n > P$ then $T$ has at least one IDE.

Furthermore, let $(v_1, v_2)$ be an edge on $T$, let $T_2$ be the subtree of $T-v_1$, which contains $v_2$ and let $r_1$ be the radius required to cover $T_2$ from a center located at $v_1$. If $r_1 > r$ then $T_2 \cup \{v_1\}$ contains an IDE.

Proof: We prove the lemma by describing a procedure (not necessarily efficient) to find an IDE. Let $y_1$ be an arbitrary vertex of $T$ and let $y_3$ be the vertex of maximum weighted distance from $y_1$. Denote by $y_2$ the vertex next to $y_1$ on the path from $y_1$ to $y_3$ and denote by $T_y$ the subtree of $T-v_1$, which contains $y_2$. Then, clearly the radius $r_1$, which is required to cover $T_y$ from a center located at $y_1$, is greater or equal to the optimal $P$-radius $r$ of the whole tree.

Now there are two possibilities: If $r_2$, the radius required to cover $T_y$ from a center located at $y_2$, is less than $r$, then $(y_1, y_2)$ is the desired IDE. Otherwise, assign $y_1 + y_2$, let $y_3$ be the vertex of maximum weighted distance from $y_1$ in $T_y$. Now repeat the whole process, namely: denote by $y_2$ the next vertex to $y_1$ on the path from $y_1$ to $y_3$ and denote by $T_y$ the subtree of $T-v_1$, which contains $y_2$. Again $r_1$ is defined to be the radius which is required to cover $T_y$ from a center located at $y_1$. Moreover, since the (new) $r_1$ is in fact equal to (the old) $r_2$, then $r_1 > r$ and again there are two possibilities, thus the process can be repeated.

If at no stage the process halts with $r_1 > r > r_2$ then we finally reach a point where $w_2$ is a leaf, in which case $r_2 = 0$, the relation $r_1 > r > r_2$ holds and the procedure successfully terminates.

Q.E.D.
The significance of the definition of the IDE is rather illuminated by the following lemma, on which the algorithms of this paper are based.

**Lemma 2**: Given a tree $T$, an IDE $(v_1, v_2)$, then there exists a vertex [absolute] $P$-center $x^*_p$ such that all vertices of $T_2$ (as defined in Definition 1) are covered by a center point $x_1 \in X_p$ where $x_1$ is either $v_1$ or $v_2$ [where $x_1$ is on the IDE $(v_1, v_2)$].

**Proof**: The proof follows from the algorithm for finding a dominating set of radius $r$ as appears in [KH] (Algorithm 4.3). If, when applying Algorithm 4.3 to the given tree, we first process the vertices of the subtree $T_2$, then clearly one of the following cases occurs:

a) If $r_u = r$ then Algorithm 4.3 locates a (vertex or absolute) center point that covers all vertices of $T_2$ at the vertex $v_1$.

b) If $r_u > r > r_L$ then a vertex center point that covers all the vertices of $T_2$ is located on $v_2$ [an absolute center point is located on the IDE $(v_1, v_2)$]. Thus the lemma follows.

We now present an algorithm to find an IDE in a given tree. The algorithm is based on the proof of Lemma 1, but the search for the vertex $y_1$ is carried out more efficiently. Rather than considering consecutive vertices for $y_1$, one by one, the search is done in a binary manner which assures a quick convergence of the algorithm. The improved algorithm therefore walks as follows:

As an initial candidate for $y_1$ we choose the centroid, $C$, of $T$. As before we denote by $y_3$ the vertex of maximum weighted distance from that centroid. We also define a subtree denoted by $CS$ to be the subtree on which the search for an IDE is performed, where initially $CS$ consists of $y_1$ and of that subtree of $T\{y_1\}$ which contains $y_3$. 
Now the algorithm iteratively proceeds in the following way:

we first find the centroid of the subtree CS and reassign it as the new $y_1$. Let IS be the subtree of $T - \{y_1\}$ which contains $C$ (the centroid of $T$), and let $y_3$ be the vertex of maximum weighted distance from $y_1$ on $T$. Denote the weighted distance from $y_3$ to $y_1$ by $r_y$, and call the algorithm to find a dominating set of radius $r_y$ on $T$. Let $p_y$ be the number of centers required to cover $T$ by radius $r_y$. Then there exist two cases:

a) If $p_y \leq P$ then clearly $r_y > r$ (where $r$ is the optimal $P$-radius of $T$). Thus, if we call by $T_v$ that subtree of $T - \{y_1\}$ which contains $y_3$, then, according to Lemma 1, the subtree $T_v \cup \{y_1\}$ contains an IDE and, according to the definition of CS and of IDE, we assign $CS + CS \cap (IS \cup \{y_1\})$ and proceed to the next iteration.

b) If $p_y > P$ then $r_y < r$. Thus, following the definitions of CS and IDE we assign $CS + CS \cap (IS \cup \{y_1\})$ and proceed to the next iteration.

Since, by our choice of $y_1$ the size of CS in each iteration is cut by half, then after no more than $\log n$ iterations we reach a point where CS consists of a single edge, which is the desired IDE.

The procedure to find an IDE is formally formulated in the following algorithm:

Algorithm 1: Find an IDE

0) [Initialization]: Find a centroid, $C$ of the tree and assign $y_1 + C$

Find a vertex $y_3$ of maximum weighted distance from $y_1$.
Let $CS$ be the subtree consisting of $y_1$ and that subtree of $T - \{y_1\}$, which contains $y_3$. 

Technion - Computer Science Department - Technical Report CS0249 - 1982
1) [New iteration]: If \(|CS| = 2 \) then halt. \( CS \) is the desired IDE.

   Else, assign \( y_1 \) the centroid of \( CS \);

   denote by IS the subtree of \( T - \{y_1\} \) which contains \( C \); find a
   vertex \( y_3 \) of maximum weighted distance from \( y_1 \) on \( T - IS \) and
   assign \( r_y + w(y_3) \od (y_1, y_3) \).

2) [A dominating set]: Call Algorithm 4.3 of [KH] to find the number \( P_y \)
   of center points required to cover \( T \) by radius \( r_y \). If \( P_y \leq P \)
   then goto 3, else, goto 4.

3) \([P_y \leq P]\): Let \( T_r \) be that subtree of \( T - \{y_1\} \) which contains \( y_3 \);
   assign \( CS \rightarrow CS \cap (T_r \cup \{y_1\}) \) and goto 1.

4) \([P_y > P]\): Assign \( CS \rightarrow CS \cap (IS \cup \{y_1\}) \) and goto 1.

The correctness of the algorithm follows directly from Lemma 1 and
the discussion which precedes. The reader should only convince
himself that at no stage of the algorithm \( CS \) is a disconnected tree,
nor that it is empty or consists of a single vertex. Therefore,
the steps of the algorithm are well defined.

Each step of the algorithm requires \( O(n) \) time, and since the
loop of Steps 1-4 can be repeated at most \( \log n \) times, the total
complexity of Algorithm 1 is \( O(n \log n) \).

3. \( V \mid V \mid P \)

   In this section we apply Algorithm 1 and Lemma 2 to build an
   algorithm which locates one by one the center points of a vertex P-
   center on a weighted tree.

   At each stage, the algorithm uses Algorithm 1 to find IDE, then
   applies Lemma 2 in order to locate a new point of the P-center.

   Let \( T \) be a weighted tree on which a vertex P-center is searched for.
Lemma 2 assures us that either $v_1$ or $v_2$ is a center point, which covers all vertices of $T_2$ (see Def. 1). Moreover, if the center point is in $v_1$, then it covers all the vertices of $T_2$ by radius $r_u$, while step 2 of Algorithm 1 guarantees that a $P$-center which covers the whole tree $T$ by radius $r_u$ does really exist. Thus, we can assign $r_f + r_u$ to be a feasible (although not necessarily optimal) solution of the $P$-radius problem on $T$.

Let us now treat the case where the center point which covers $T_2$ is in $v_2$ rather than $v_1$ (in this case we know that the radius $r_L$ by which $v_2$ covers the vertices of $T_2$ is less than the optimal $P$-radius $r$). Applying (in a binary search manner) Algorithm 4.3 of [KH] we can now find in $T - T_2$ a vertex $v_3$ of minimum weighted distance $r_3$ from $v_2$ such that $r_3 \geq r$. Again we should consider two cases: if (in the optimal solution) $v_3$ is covered by $v_2$, then the radius of this solution is $r_3$. Therefore, if $r_3 < r_f$, then we must update our feasible solution $r_f$, in order to keep it the best solution found so far. The other case is a solution in which $v_2$ does not cover $v_3$ but covers all the closer vertices of $T - T_2$. In this case, we define a tree $\hat{T}$ which is identical to the original tree $T$ with the exception that all vertices which are covered in $\hat{T}$ by $v_2$ have weight 0 in $\hat{T}$. If we now recursively call the algorithm of this section to find $\hat{r}$, the optimal $(P-1)$ radius of $\hat{T}$, then clearly the optimal $P$-radius of the original tree $T$ is $r = \min(r_f, \hat{r})$. 
Algorithm 2: $\text{VPCNTR}(T,p)$

[This algorithm receives a weighted tree $T$ and a parameter $p$ and returns the optimal vertex $P$-radius $r$ of the tree $T$.]

0) If $p = 1$ then find the vertex $1$-radius $r_1$. RETURN($r_1$)

1) [Find and IDE]: call Algorithm 1 to find an IDE $(v_1,v_2)$ of the tree $T$. Let $r_u$ be the radius by which $v_1$ covers the vertices of $T_2$. Assign $r_f + r_u$.

2) [Sort the vertices of $T-T_2$]: Arrange the vertices of $T-T_2$ in an ordered list according to their weighted distances from $v_2$.

3) [Find $v_3$]: Apply Algorithm 4.3 of [KH] to perform a binary search on the list of step 2 in order to find in $T-T_2$ a vertex $v_3$ of minimum weighted distance $r_3$ from $v_2$ such that $r_3 \geq r$ (namely, $P_3$, the domination number of radius $r_3$ on $T$ is not greater than $P$). If $r_3 < r_f$ assign $r_f \leftarrow r_3$.

4) [Define $\hat{T}$]: Let $\hat{T}$ be the tree $T$, where all vertices of $T-T_2$, whose weighted distances from $v_2$ are less than $r_3$, and all vertices of $T_2$ have weight 0.

5) [Recursive call]: $\hat{r} = \text{VPCNTR}(\hat{T},(p-1))$

6) RETURN $\min(r_f,\hat{r})$.

The correctness of Algorithm 2 follows from the discussion above.

The complexity of each of the steps 1-3 is $O(n \log n)$. The complexity of step 4 is $O(n)$. Thus each recursive call of step 5 requires $O(n \log n)$ time and the whole algorithm requires $O(p n \log n)$ time.

In this section we present an algorithm to find an absolute $P$-center in a weighted tree. This algorithm is based on the ideas of the previous...
According to Lemma 2, given an IDE, there exists an absolute P-center, \( x^*_p \), such that all vertices of \( T_2 \) are covered by a center point \( x_1 \), where \( x_1 \) belongs to the IDE. Thus we could think of an algorithm for finding an absolute P-center which essentially follows the lines of Algorithm 2. The problem, however, lies in the fact that in the absolute case the center point \( x_1 \), which belongs, by Lemma 2, to the IDE, is not necessarily confined to be on the vertices. Therefore, a priori, instead of considering only the two endpoints of the IDE we apparently must check many possible locations of the center point on the IDE.

The solution to this difficulty arises from the properties of the absolute P-center. It is well known (e.g. [KH]) that the location of an absolute center point can be restricted to at most \( O(n^2) \) points on each edge of the graph, namely the end point of the edge and those internal points on the edge, which lie exactly in equal weighted distance between two vertices. Moreover, the center point \( x_1 \) that we are looking for, must cover all vertices of the subtree \( T_2 \) which is defined by the given IDE. Thus, if we define on the points, \( x_1 \), of the IDE the following function:

\[
D(x) = \max_{v \in T_2} \{ w(v) \cdot d(x,v) \} \quad x \in \text{IDE}
\]

then a possible location for the required center point, \( x_1 \), on the IDE must either be the end point \( v_1, v_2 \), or satisfy for some vertex \( v' \in T-T_2 \):

\[
D(x_1) = w(v') \cdot d(x_1, v').
\]

Clearly, there exist at most \( O(n) \) such possible points on the IDE.

Let \( u_1 u_2 \ldots u_k \) be those points arranged according to their order on
the IDE; such that \( u_1 = v_1 \) and \( u_k = v_2 \), and denote by \( r_i \) the radius required to cover all vertices of \( T_2 \) by a center point located at \( u_i \) (i.e. \( r_i = D(u_i) \), in particular \( r_1 = r_u \) and \( r_k = r_L \)).

Before we go on discussing the whole algorithm to find an absolute P-center on a tree, let us describe a procedure which finds the points \( u_1 \) ... \( u_k \). Since those points (except for \( u_1 \) and \( u_k \), the end points of the IDE) are given by the relation

\[
D(v_i) = w_i(v') \cdot d(v', u_i).
\]

(where \( v' \) is some vertex of \( T - T_2 \) and \( 1 < i < k \)), then a subroutine and an appropriate data structure to build a function \( D(\cdot) \) should be provided. However, such a procedure was already presented in [KH]: it is Algorithm 2.1 in [KH] which finds a local center on a given edge.

To achieve that goal Algorithm 2.1 builds in steps 1-9 the function \( D(\cdot) \), which is defined there exactly in the same way as determined here.

Thus, we can apply steps 1-9 of Algorithm 2.1 in [KH] in order to build a function \( D(\cdot) \), where, in our case, the graph and the given edge, \( \ell \), on which Algorithm 2.1 runs, are the subtree \( T_2 \) and the IDE respectively.

After the function \( D(\cdot) \) is built, what is left to do is to find for each vertex \( v' \in T-T_2 \) the point \( x' \) on the IDE, where

\[
D(x') = w(v') \cdot d(v', x') \quad \text{(if such point exists).}
\]

Arranging those points according to their order in the IDE gives us the desired list

\( u_1, u_2, ..., u_k \) and the corresponding values of the radii \( r_1, r_2, ..., r_k \).

We now return to the main problem of finding an absolute P-center of a weighted tree. Since the points \( \{u_i\} \) are arranged according to their order on the IDE, clearly if \( i > j \) then \( r_i > r_j \).

On the other hand, by the definition of IDE we have

\[
r_u = r_1 \geq r = r_2 \geq r_k = r_L.
\]

Therefore, there must exist two adjacent points
\( u_j \) and \( u_{j+1} \) (\( j \leq j < k \)) such that \( r_j > r > r_{j+1} \).

It is obvious that out of all the possible points \( \{u_i\}_{i=1}^k \) only \( u_j \) and \( u_{j+1} \) should be considered as candidate points for the center point \( x_1 \) on the IDE. For if we locate \( x_1 \) on \( v_j \), then we can build an absolute P-center of radius \( r_j \), that covers the whole tree \( T \), and therefore the points \( u_1 \ldots u_{j-1} \) of greater radius need not be considered. On the other hand, if we locate the center points \( x_1 \) on \( v_{j+1} \), then we cover all vertices of \( T \) and maybe some vertices of \( T \) by radius \( r_{j+1} \), which is less than the total optimal P-radius \( r \). Therefore, there is no use to cover the vertices of \( T_2 \) by a center point of smaller radius located at \( u_{j+2} \ldots u_k \) because the only possible result would be that less vertices of \( T \) will now be covered by \( x_1 \) while total absolute radius will not be improved (it may even become worse).

To conclude, only two cases should be examined: If \( x_1 \) is located on \( u_j \), then the whole tree can be covered by an absolute P-center of radius \( r_j \). If \( x_1 \) is located on \( u_{j+1} \), then this point covers all vertices of \( T_2 \) and maybe some vertices of \( T \), while the rest of the tree should be covered by an absolute p-1 center for which our algorithm should recursively be called.

Algorithm 3: APCNTR(T,p)

[This algorithm receives a weighted tree \( T \) and a parameter \( p \), and returns the optimal absolute P-radius, \( r \), of the tree.]

0) If \( p=1 \) then find the absolute one radius, \( r_1 \), of \( T \). RETURN(\( r_1 \)).
1) [Find an IDE]: Call Algorithm 1 to find an IDE \((v_1,v_2)\) on the tree.
2) [Build function \( D(\cdot) \)]: Apply steps 1-9 of Algorithm 2.1 in [KH] with respect to the subtree \( T_2 \) in order to build the function \( D(\cdot) \) on the IDE.
3) [Find the candidate points on the IDE]: For each vertex \( v' \in T-T_2 \) find in the IDE a point \( x' \) such that \( D(x') = w(v')d(v', x') \) (if such a point exists).

4) [Build the list \( u_i \)]: Arrange the points which were found in Step 3 according to their order on the IDE and add to them the points of the IDE. Denote those points by \( u_1, u_2, \ldots, u_k \), such that \( u_1 = v_1 \), \( u_k = v_4 \) and denote \( r_i^1 = D(u_i) \).

5) [Find \( u_j \) and \( u_{j+1} \)]: Perform a binary search and apply algorithm 4.3 of [KH] in order to find two adjacent points \( u_j \) and \( u_{j+1} \) such that \( r_j^1 > r > r_{j+1}^1 \).

6) [Prepare the procedure call]: Assign zero weight to all the vertices of the tree whose weighted distance from \( u_{j+1} \) is not greater than \( r_{j+1}^1 \). Denote the now weighted tree by \( T \).

7) [Receive call]: \( f \leftarrow \text{APCNTR}(T, \{P-1\}) \).

8) RETURN \( \text{Nim}(f, r_j^1) \).

The complexity of each of the step 1-5 is \( O(n \log n) \). The complexity of step 6 is \( O(n) \). Thus each recursive call of the algorithm requires \( O(n \log n) \) time and the whole algorithm requires \( O(p \ n \log n) \) time.
REFERENCES


[M] N. Meggido: "Linear-time algorithm for linear programming in R^3 and related problems", Statistics Department, Tel Aviv University (Jan. 1982).
