# The $k$-Median Problem for Directed Trees
# (Preliminary Version)

Marek Chrobak* Lawrence L. Larmore† Wojciech Rytter‡

December 11, 2003

**Abstract**

The $k$-median problem is a classical facility location problem. We consider the $k$-median problem for *directed trees*, motivated by the problem of locating proxies on the World Wide Web. The two main results of the paper are an $O(n \log n)$ time algorithm for $k = 2$ and an $O(n \log^2 n)$ time algorithm for $k = 3$. The previously known upper bounds for these two cases were $O(n^2)$.

## 1 Introduction

In the *$k$-median problem* we are given a graph $G$ in which each node $u$ has a non-negative weight $w_u$ and each edge $(u, v)$ has a non-negative length $d_{uv}$. Our goal is to find a set $F$ of $k$ vertices that minimizes $Cost(F) = \sum_{x \in G} \min_{u \in F} w_x d_{xu}$. We think of the nodes of $G$ as customers, with each customer $x$ having a demand $w_x$ for some service. $F$ is a set of $k$ *facilities* that provide this service. It costs $d_{xu}$ to provide a unit of service to customer at $x$ from a facility located at $u$. We wish to place $k$ facilities in $G$ so that the overall service cost is minimized. This optimal set is called the *$k$-median* of the weighted graph $G$. The $k$-median problem is a classical facility location problem which has been a subject of study for several decades. The general case is known to be NP-complete (see Problem ND51 in [8]), and most of the past work focussed on efficient heuristics (see, for example, [14]) and approximation algorithms [2, 6, 5].

**Trees.** In this paper we focus on the case of the $k$-median problem when $G$ is a tree. In 1979 Kariv and Hakimi [13] presented a dynamic programming algorithm for this problem that runs in time $O(k^2n^2)$. Hsu [12] gave an algorithm with running time $O(kn^3)$. More recently, Tamir [18] showed that the Kariv-Hakimi algorithm is faster than previously thought, namely that its running time is only $O(kn^2)$. This is the best upper bound known to this date.

*Department of Computer Science, University of California, Riverside, CA 92521. Email: marek@cs.ucr.edu.
Research supported by grant EPSRC GR/N09077 (UK) and by NSF grant CCR-9988360.

†Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019. Research supported by grant EPSRC GR/N09077 (UK).

‡Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097, Warszawa, Poland, and Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK. Email: rytter@csc.liv.ac.uk. Research supported by grant EPSRC GR/N09077.

Some other special cases of the problem have been considered. Hassin and Tamir [11] gave an $O(kn)$ time algorithm when the tree is a path. Another $O(kn)$-time algorithm for this problem was later given by Auletta *et al.* [4]. The case $k = 2$ was considered by Gavish and Sridhar [9] who presented an algorithm with running time $O(n \log n)$. The dynamic version of the case $k = 2$ was studied by Auletta *et al.* in [3][1].

**Directed trees.** We consider the $k$-median problem for *rooted directed trees.* Let $T$ be a directed tree with root $r$. If $y$ is the father of $x$, denote by $d_{xy}$ the length of edge $(x, y)$. The length function extends in a natural way to any pair $x, y$ where $y$ is an ancestor of $x$. As in the undirected case, each node $x$ is assigned a non-negative weight $w_x$. Our goal is to find a set of nodes $F$ of size $k$ that minimizes the quantity:

$$Cost(F) \quad = \quad \sum_{x \in T} \min_{u \in F + r} w_x d_{xu}.$$

We use the same terminology as in the undirected case, that is, the nodes in $F$ are called *facilities*, and the optimal set $F$ is called the *k-median* of $T$. Note, however, that in our case we actually have $k + 1$ facilities in the tree, since the default "root facility" at $r$ is fixed and not counted in $F$. For specific values of $k$ this distinction makes the undirected case quite different from the undirected one. For example, for $k = 2$, in the undirected case we need to partition the tree into two subtrees that form the domains of the two facilities. This task can be accomplished by searching the tree for an edge that separates the tree into two appropriately balanced subtrees [9, 3]. In the directed case, however, we need to partition $T$ into *three* subtrees.

The $k$-median problem for directed trees was introduced by Li *et al.* [16] as a mathematical model for optimizing the placement of web proxies to minimize average latency. In [16], the authors present a $O(k^2n^3)$-time algorithm for this problem. This was later improved to $O(k^2n^2)$ by Vigneron *et al.* [19]. In [19] the authors also noted that the Kariv-Hakimi algorithm can be modified to find $k$-medians in directed trees in time $O(k^2n^2)$. Tamir (private communication) observed that his method from [18] can be adapted to reduce the time complexity to $O(kn^2)$. For the line, Li *et al.* [15] give an algorithm with running time $O(kn^2)$. This was just recently improved by Woeginger [20] to $O(kn)$.

**Our results.** We present several algorithms for the $k$-median problem on directed trees. We start by describing, in Section 2, the basic dynamic programming algorithm, Algorithm AncDP, which is an adaptation of the Kariv-Hakimi algorithm to directed trees (see also [19]). Then, in Section 3, we propose an alternative dynamic programming algorithm, Algorithm DepthDP. For $k = 2$ this algorithm can be implemented in time $O(n \log^2 n)$. The problem of estimating the worst-case running time of Algorithm DepthDP reduces to an interesting combinatorial problem of estimating the number of line segments in a certain piece-wise linear function. In Section 3.3 we show that for $k = 2$ this number is $O(n)$ and we conjecture that it is also linear for any fixed $k$. This conjecture would imply that, for each fixed $k$, Algorithm DepthDP has time complexity $O(n^2)$. We believe that Algorithm DepthDP is faster than Algorithm AncDP on realistic data. Both algorithms are being implemented, and will be tested on randomly generated trees with

[1]In [3], the authors also claimed a linear-time algorithm; however, their running time analysis is flawed.

various characteristics. Our experimental results will be reported in the final version of this paper.

In Section 4 we show that with an additional requirement that the facilities are independent (no facility is an ancestor of another) the problem can be solved in time $O(k^2 n)$.

Our main results are given in Sections 5 and 6. In Section 5 we improve the asymptotic running time for $k = 2$ to $O(n \log n)$. By the result from Section 4, it is sufficient to consider only the case when one facility is an ancestor of the other. For this case we show that the objective function satisfies the Monge property. Our algorithm Fast2Median uses the divide-and-conquer approach. A given tree $T$ is split into two smaller trees of approximately equal size, for which the solutions are computed recursively. Yet another tree is constructed, on which we apply the SMAWK algorithm [1] for computing maxima in Monge matrices. We use the solutions for these three instances to obtain the solution for $T$.

In Section 6, we deal with the case $k = 3$. For this case we present Algorithm Fast3Median that runs in time $O(n \log^2 n)$. The algorithm breaks the problem into cases depending on various genealogical patterns among facilities, and solves each case independently. The cases are solved by a combination of dynamic programming and divide-and-conquer methods.

## 2 Ancestor-Based Dynamic Programming

Throughout the paper, we will assume that $T$ is a binary tree with $n$ nodes in which each non-leaf node has two sons. Any tree can be transformed into such a binary tree by adding a linear number of nodes with zero weight. By $T_u$ we denote the subtree of $T$ rooted at $u$, and $n_u = |T_u|$ is the number of descendants of $u$. By $d_u = d_{ur}$ we denote the *depth* of $u$, and by $\omega_u$ the total weight of $T_u$, that is $\omega_u = \sum_{x \in T_u} w_x$. We often refer to $\omega_u$ as the *tree weight* of $u$.

We now show an adaptation of the Kariv-Hakimi algorithm to directed trees. Let $Cost_x^j(u)$ be the optimal cost of placing $j$ facilities within $T_x$ given that the closest facility above $x$ is $u$. Formally, define $T_x^u$ to be the tree that consists of $T_x$ with an additional root node $r_u$ whose only son is $x$, and with edge $(x, r_u)$ having length $d_{xu}$. Then $Cost_x^j(u)$ is the minimum cost of placing $j$ facilities in $T_x^u$. Note that we allow $u = x$, in which case $d_{xu} = 0$, and $Cost_x^j(x)$ will be equal to the minimum cost of placing $j$ facilities in $T_x$ (not counting the default facility at $x$).

**Algorithm** AncDP**.** We only show how to compute optimal cost. The algorithm can be easily modified to determine the $k$-median. We compute all quantities $Cost_x^j(u)$ bottom-up. If $x$ is a leaf, set $Cost_x^0(u) = w_x d_{xu}$ and $Cost_x^j(u) = 0$ for $j \geq 1$. Suppose that $x$ is not a leaf and $y, z$ are the sons of $x$. For each $u$ that is an ancestor of $x$ and $j = 0, \ldots, k$ we then compute:

$$Cost_x^j(u) \quad = \quad \min \left\{ \begin{array}{l} \min_{p+q=j} \left\{ \, Cost_y^p(u) + Cost_z^q(u) \, \right\} + w_x d_{xu} \\ Cost_x^{j-1}(x) \end{array} \right.$$

For $j = 0$, assume that $Cost_x^{j-1}(x) = \infty$. In the the first case we have $p$ facilities in $T_y$, $q$ facilities in $T_z$ and there is no facility at $x$. In the second case we have one facility at $x$. When we are done, return $Cost_r^k(r)$.

**Running time.** Algorithm AncDP clearly runs in time $O(k^2 n^2)$. The bound can be improved to $O(kn^2)$ with careful analysis.

**Theorem 1** *Algorithm* AncDP *computes $k$-medians in directed trees in time $O(kn^2)$.*

As noted by Tamir (private communication), the proof of the above theorem is very similar to the ones given in [10, 18]. For the sake of completeness, we present a sketch. A node $u$ is *heavy* if both of its subtrees have size at least $\lceil k/2 \rceil$; otherwise $u$ is called *light*. The number of heavy nodes is at most $\frac{n-k}{k} = O(n/k)$, so the contribution of the heavy nodes is $O(kn^2)$. To estimate the contribution of the light nodes, denote by $\ell_u$ the total cost of processing the light nodes in $T_u$. If $x$ is heavy, and $y, z$ are the sons of $x$, then $\ell_x \le \ell_y + \ell_z$. If $x$ is light then $\ell_x \le \ell_y + \ell_z + n \min(n_y, k) \min(n_z, k)$. By induction, it is not hard to show that $\ell_x = O(nn_x \min(n_x, 2k))$. Therefore the total contribution of the light nodes is also $O(kn^2)$.

## 3 Depth-Based Dynamic Programming

In this section we give a different algorithm called DepthDP. The idea is that instead of computing $Cost_x^j(u)$ for the ancestors of $u$, we will compute a real-argument function $Cost_x^j(\alpha)$, for an appropriate range of $\alpha$, which represents the distance $d_{xu}$. For $k = 2$ we show that an implementation of this algorithm called DepthDP2 that runs in time $O(n \log^2 n)$. Although later in Section 5 we improve the upper bound further to $O(n \log n)$, we include DepthDP2 in the paper, since it is considerably simpler and easier to implement. The main principle of DepthDP will, however, play a major role in Section 6, where we improve the upper bound for $k = 3$ to $O(n \log^2 n)$.

The problem of estimating the time complexity of DepthDP for all $k$ reduces to an interesting combinatorial problem of estimating the number of line segments in a certain piece-wise linear function. We show that this bound is $O(n)$ for $k = 2$, and we conjecture that it is also linear for all $k$. We also believe that Algorithm DepthDP is faster than Algorithm AncDP on realistic data. Both algorithms are being implemented, and will be tested on randomly generated trees with various characteristics. Our experimental results will be reported in the final version of this paper.

Let $Cost_x^j(\alpha)$ be the optimal cost of placing $j$ facilities within $T_x$ given that the closest facility above $x$ is at distance $\alpha$. Formally, define $T_u^\alpha$ to be the tree that consists of $T_x$ and an extra root node $r_\alpha$ whose only son is $x$ and with edge $(x, r_\alpha)$ having length $\alpha$. Then $Cost_x^j(\alpha) = Cost_x^j(r_\alpha)$, the minimum cost of placing $j$ facilities in $T_x^\alpha$. We allow $\alpha = 0$, in which case $Cost_x^j(0) = Cost_x^j(x)$.

**Algorithm** DepthDP. As before, we compute all functions $Cost_x^j(\alpha)$ bottom-up. If $x$ is a leaf, set $Cost_x^0(\alpha) = w_x\alpha$ and $Cost_x^j(\alpha) = 0$ for $j \ge 1$. Suppose that $x$ is not a leaf and $y, z$ are the sons of $x$. Similarly as in Algorithm AncDP, we have

$$Cost_x^j(\alpha) \;=\; \min \begin{cases} \min_{p+q=j} \left\{ Cost_y^p(\alpha + d_{yx}) + Cost_z^q(\alpha + d_{zx}) \right\} + w_x\alpha \\ Cost_x^{j-1}(0) \end{cases}$$

where $Cost_x^{-1}(0) = \infty$. When we are done, return $Cost_r^k(0)$.

**Functions** $Cost_x^j(\alpha)$**.** We still need to explain how to store and maintain the functions $Cost_x^j(\alpha)$ throughout the computation. Given a set $F \subseteq T_x^\alpha$ of size $j$, denote by $Cost_x^F(\alpha)$ the cost of having $j$ facilities in $F$. For a fixed $F$, $Cost_x^F(\alpha)$ is a linear function, namely $Cost_x^F(\alpha) = a_F\alpha + b_F$ for some constants $a_F, b_F$. Thus $Cost_x^j(\alpha)$ is a concave, piece-wise linear-function.

The crucial questions are: (1) What is the complexity (number of segments) of $Cost_x^j(\alpha)$? (2) How to update our data structures efficiently? For $k = 2$, we address both questions later in this section.

### 3.1 The Lower Envelope Data Structure

Consider the following problem. We want to maintain a lower envelope $L(\alpha)$ of a finite number of linear functions, with the following operations:

| | |
|---|---|
| VerticalShift: | $L(\alpha) \leftarrow L(\alpha) + a\alpha + b$, for some constants $a, b$. |
| HorizontalShift: | $L(\alpha) \leftarrow L(\alpha + c)$, for some constant $c$. |
| NewSegment: | $L(\alpha) \leftarrow \min\{L(\alpha), a\alpha + b\}$, for some constants $a, b$. |
| Evaluate: | Output $L(\gamma)$, for some given constant $\gamma$. |

We can keep the segments of $L(\cdot)$ in a balanced binary search tree ordered from left to right (or, equivalently, by the slope values), and we can keep track of the horizontal and vertical offsets. This will give us the following lemma.

**Lemma 1** *There is a data structure* LowEnv *for maintaining the lower envelope $L(\cdot)$ such that each operation can be implemented in time $O(\log m)$, where $m$ is the number of line segments in $L(\cdot)$ (at the time the operation is performed).*

### 3.2 Algorithm DepthDP for $k = 2$

We now show how to implement Algorithm DepthDP in time $O(n \log^2 n)$ for $k = 2$. When $k = 2$, we have $j \in \{0, 1, 2\}$. For $j = 0$, functions $Cost_x^0(\alpha)$ are linear functions and they can be easily updated and evaluated in time $O(1)$ per node. For $j = 2$, we are only interested in the values of $Cost_x^2 = Cost_x^2(d_x)$, since there can be no facilities between $x$ and $r$. The hardest case is for $j = 1$, since $Cost_x^1(\alpha)$ is a function consisting of up to $n_x$ segments (where $n_x = |T_x|$), and we need to update and evaluate these functions efficiently. We will achieve this with the LowEnv data structure described above.

**Algorithm** DepthDP2. We compute all functions bottom-up. If $x$ is a leaf, set $Cost_x^0(\alpha) = w_x\alpha$, $Cost_x^1(\alpha) = 0$ and $Cost_x^2 = 0$. Suppose that $x$ is not a leaf and $y, z$ are the sons of $x$. The recursive formulas for the cost functions are as follows:

$$
\begin{aligned}
Cost_x^0(\alpha) &= Cost_y^0(\alpha + d_{yx}) + Cost_z^0(\alpha + d_{zx}) + w_x\alpha \\
Cost_x^1(\alpha) &= \min \begin{cases} Cost_y^1(\alpha + d_{yx}) + Cost_z^0(\alpha + d_{zx}) + w_x\alpha \\ Cost_y^0(\alpha + d_{yx}) + Cost_z^1(\alpha + d_{zx}) + w_x\alpha \\ Cost_x^0(0) \end{cases} \\
Cost_x^2 &= \min \begin{cases} Cost_y^0(d_y) + Cost_z^2 + w_x d_x \\ Cost_y^2 + Cost_z^0(d_z) + w_x d_x \\ Cost_y^1(d_y) + Cost_z^1(d_z) + w_x d_x \\ Cost_x^1(0) \end{cases}
\end{aligned}
$$

**Running time.** For $j = 0$, it is easy to see that $Cost_x^0(\alpha)$ is a linear function, say $Cost_x^0(\alpha) = a_x\alpha + b_x$. We maintain $Cost_x^0(\alpha)$ by keeping track of the coefficient of $\alpha$ and the additive constant at each step: $a_x = a_y + a_z + \omega_x$ and $b_x = a_y d_{yx} + b_y + a_z d_{zx} + b_z$. The numbers $Cost_y^0(d_y)$, $Cost_z^0(d_z)$ can be evaluated in time $O(1)$. For $j = 2$, using LowEnv, evaluating $Cost_y^1(d_y)$ and $Cost_z^1(d_z)$ above will take time at most $O(\log n)$, for the total of $O(n \log n)$.

Consider now the case $j = 1$. The computation of $Cost_x^1(\alpha)$ is dominated by computing the minimum of the functions: $Cost_y^1(\alpha + d_{yx}) + Cost_z^0(\alpha + d_{zx})$ and $Cost_y^0(\alpha + d_{yx}) + Cost_z^1(\alpha + d_{zx})$, since the rest costs time $O(\log n)$ per operation using the LowEnv data structure. If $n_y$ and $n_z$ are the cardinalities of $T_y$ and $T_z$, respectively, assume, without loss of generality that $n_y \geq n_z$. Then we can compute the above minimum by executing NewSegment $n_z$ times, each at cost $O(\log n)$. We amortize it, by charging $O(\log n)$ to each of the nodes in $T_z$. A node gets charged only if its subtree gets merged with a bigger (or equal size) subtree. This can occur at most $O(\log n)$ time for each node. Summarizing, we get the following results.

**Theorem 2** *Algorithm* DepthDP2 *computes* 2*-medians in directed trees in time* $O(n \log^2 n)$.

### 3.3 Minimizers

In order to estimate the worst-case running time of Algorithm DepthDP, one needs to bound the complexity (that is, the number of segments) of functions $Cost_x^j(\cdot)$. We consider this issue in this section. Without loss of generality we can restrict our attention to the case when $x$ is the root of $T$ and $j = k$. For simplicity, we will write $Cost(\alpha)$ for $Cost_x^k(\alpha)$.

For a set $F$ of $k$ vertices in $T$, let $Cost^F(\alpha)$ be the cost of placing $k$ facilities on $F$ in $T^\alpha$. We say that $F$ is a *minimizer of* $Cost(\cdot)$ *at* $\alpha$ if $F$ minimizes $Cost^F(\alpha)$ among all $k$-element sets. Each segment of $Cost(\cdot)$ belongs to some line $Cost^F(\cdot)$. Thus we are interested in the following question: How many different minimizers we may have? Clearly, this number cannot exceed $\binom{n}{k}$. We conjecture that this number is substantially smaller.

**Conjecture 1** *The Linearity Conjecture: For each fixed* $k$*, the number of minimizers is* $O(n)$*.*

The linearity conjecture immediately implies that, for any fixed $k$, Algorithm DepthDP runs in time $O(n^2)$. For varying $k$, the complexity of the algorithm depends on how fast the bound on number of maximizers grows with $k$.

The linearity conjecture is trivially true for $k = 1$. In the rest of this section we show that it also holds for $k = 2$.

Instead of minimizing cost, we will consider the equivalent problem of maximizing the *gain* resulting from placing $k$ facilities on set $F$, that is $Gain(F) = \sum_{x \in T} w_x d_x - Cost(F)$. Similarly to $Cost^F(\alpha)$ we define the *gain function* $Gain^F(\alpha)$. A set $F$ that maximizes $Gain^F(\alpha)$ is called a *maximizer of* $Gain(\cdot)$ *at* $\alpha$. Obviously, the maximizers of $Gain(\cdot)$ are the same as the minimizers of $Cost(\cdot)$.

Let $F = \{u, v\}$ be a maximizer of $Gain(\cdot)$. We have two cases: the *ancestral case*, when one of $u, v$ is an ancestor of another, and the *independent case*, when none is an ancestor of the other. It is easy to see that there are at most $n$ ancestral maximizers, so it suffices to prove that the number of independent maximizers is $O(n)$. To this end, we consider the function

$$IndGain(\alpha) \quad = \quad \max_{u \perp v} Gain^{\{u,v\}}(\alpha)$$

where $u \perp v$ indicates that $u, v$ are independent. Our goal now is to show that function $IndGain(\cdot)$ has $O(n)$ maximizers. For the rest of this section, unless otherwise stated, a "maximizer" will mean a maximizer of $IndGain(\cdot)$. For independent nodes $u, v$, the gain function can be written in the form

$$Gain^{\{u,v\}}(\alpha) \quad = \quad f^u(\alpha) + f^v(\alpha)$$

where $f^x(\alpha) = \omega_x \alpha + b_x$, for some constant $b_x \geq 0$. To simplify the arguments, we will assume that there are no degeneracies, that is, all tree weights $\omega_x$ are distinct, and for all pairs $u, v$ the intersection points of lines $f^u(\alpha)$ and $f^v(\alpha)$ are distinct. The proof can be easily adapted to allow degenerate situations by appropriately breaking ties.

We think of the parameter $\alpha$ as *time* that ranges from 0 to $\infty$. While $\alpha$ increases, different pairs are maximizers of $IndGain(\alpha)$. We will say that $v$ is *better* than $u$ at $\alpha$ if $f^v(\alpha) \geq f^u(\alpha)$. Similarly, a pair $\{u, v\}$ is *better* than $\{p, q\}$ at $\alpha$ if $Gain^{\{u,v\}}(\alpha) \geq Gain^{\{p,q\}}(\alpha)$.

**Lemma 2** (a) *Let $\alpha < \beta < \gamma$. If $v$ is better than $u$ at $\alpha$ and $u$ is better than $v$ at $\beta$, then $u$ is better than $v$ at $\gamma$.*
(b) *Let $\alpha < \beta$ and $v \in T_u$. If $u$ is better than $v$ at $\alpha$ then it is also better at $\beta$.*

*Proof:* (a) The difference $f^u(\alpha) - f^v(\alpha) = (\omega_u - \omega_v)\alpha + (b_u - b_v)$ is a linear function, so it can change sign at most once.
(b) As in (a), we have $f^u(\alpha) - f^v(\alpha) = (\omega_u - \omega_v)\alpha + (b_u - b_v) > 0$. Since $v \in T_u$, $\omega_u > \omega_v$, so $f^u(\alpha) - f^v(\alpha)$ is an increasing function. □

We now introduce some more terminology. If a pair $\{u, v\}$ is a maximizer and $u$ is better than $v$ then we call $u$ the *primary node*, and $v$ the *secondary node.* The ties can be broken arbitrarily. We will also say that $v$ is the *partner* of $u$. A node $u$ in a maximizer is *fresh* if $u$ did not belong to any earlier maximizer. Nodes that are not fresh are called *used.* A node $u$ is *dead* at $\alpha$ if $u$ has a proper ancestor that was used in the past; otherwise $u$ is called *live.*

**Lemma 3** *Maximizers have the following properties:*
(a) *Only live nodes can belong to a maximizer.*
(b) *Any node can be a primary node only once. More precisely, if $u$ is a primary node of a maximizer at $\alpha$ but not at $\beta > \alpha$, then $u$ is not a primary node for maximizers at any $\gamma > \beta$.*
(c) *If $\{u, v\}$ is the maximizer at $\alpha$, where $u$ is the primary node and its partner $v$ is fresh, then for all $\beta > \alpha$ at which $u$ is still primary, the partner of $u$ is fresh.*
(d) *During the time when $u$ is a primary node, $u$ can have at most two used partners.*

*Proof:* (a) Suppose that $u \in T_v$ and $\{v, w\}$ is the maximizer at $\alpha$. Since $v, w$ are independent, so are $u, w$, and thus $v$ is better than $u$ at $\alpha$. By Lemma 2.b, $v$ will also be better than $u$ at any $\beta > \alpha$. Take any node $t$ independent of $v$. Then $t$ is independent of $u$ as well, and the pair $\{t, v\}$ is better than $\{t, u\}$. So $u$ cannot be in a maximizer at time $\beta$.

(b) It suffices to show that the tree weight of a primary maximizer increases monotonically. Assume that at a time $\alpha$ the primary node switches from $u$ to $p$. Clearly, $p \notin T_u$. If $u \in T_p$ then $\omega_p > \omega_u$, and we are done. So from now on we can assume that $u$ and $p$ are independent.

Let $v$ be the partner of $u$. Node $p$ is alive before $\alpha$ and independent of $u$. Thus $u$ must be better than $p$ before $\alpha$, since otherwise $\{p, u\}$ would be better than $\{u, v\}$ before $\alpha$. By a similar argument, $p$ must be better than $u$ after $\alpha$. Using these two facts, and simple algebra, we get $\omega_p \geq \omega_u$.

(c) Suppose that $u$ remains the primary node till time $\beta > \alpha$, and let its partner at $\beta$ be $w$. Until time $\alpha$, both $v$ and $w$ must be alive, so they are independent, by part (a). At time $\alpha$, $v$ is better than $w$ and, at time $\beta$, $w$ is better than $v$. So $v$ was better than $w$ at all times before $\alpha$, by Lemma 2.a. Thus, since $v$ was fresh till $\alpha$, so was $w$, since, if $w$ were used, $v$ would be used, contradiction.

(d) Towards contradiction, let $v, w, t$ be the partners of $u$ at times $\alpha < \beta < \gamma$, respectively, and suppose that they all were previously used. Then $u, v, t$ must be independent. Node $t$ is better than $v, w$ at $\gamma$, but worse than $v, w$, respectively, at $\alpha$ and $\beta$. So $t$ is worse than $v$ and $w$ at all times before $\alpha$. Then $t$ could not have appeared in any maximizer before $\alpha$. $\Box$

**Lemma 4** *Function IndGain($\alpha$) has at most $3n$ maximizers.*

*Proof:* The proof is by amortized analysis. At any given time, with each node $u$ we associate its potential $\phi(u)$ as follows. If $u$ is fresh then $\phi(u) = 3$, if $u$ was used as a secondary node but not as primary then $\phi(u) = 2$, otherwise $\phi(u) = 0$. The potential function is then defined as $\Phi = \sum_u \phi(u)$. So initially $\Phi = 3n$.

We consider the interval when $u$ is a primary node. At the beginning, we have $\phi(u) \geq 2$ and at the end $\phi(u) = 0$. By Lemma 3.d, $u$ will have at most two used partners. The corresponding maximizers can be paid for by the decrease of $\phi(u)$. In each other maximizer, $u$ will have a fresh partner whose potential will decrease from 3 to 2. So overall, during the time when $u$ is primary, the number of maximizers can be paid for by the decrease of $\Phi$. Since $\Phi$ is non-negative, the lemma follows. $\Box$

The lemma above implies our main result of this section.

**Theorem 3** *For $k = 2$, function Cost($\cdot$) has $O(n)$ minimizers.*

*Proof:* We can write the gain function as $Gain(\cdot) = \max\{IndGain(\cdot) + AncGain(\cdot)\}$, where $AncGain(\alpha)$ is the maximum gain for ancestral pairs of facilities in $T^\alpha$. By Lemma 4, function $IndGain(\cdot)$ has at most $3n$ maximizers. In the ancestral case, for each $u$ we can determine one $v \in T_u$ that maximizes the gain over all pairs that contain $u$ as the ancestor node. This implies that $AncGain(\cdot)$ has at most $n$ maximizers. Therefore $Gain(\cdot)$ has at most $4n$ maximizers, and the relationship between $Gain(\cdot)$ and $Cost(\cdot)$ implies that $Cost(\cdot)$ has at most $4n$ minimizers. $\Box$

## 4 Independent Facilities

Suppose that instead of any set of $k$ facilities, we wish to find a set of $k$ *independent* facilities that minimizes cost. We show that this problem can be solved in linear time for any fixed $k$.

**Algorithm** IndDP. The algorithm works by simple bottom-up dynamic programming. For each node $x$ and each $j$ we compute $IndGain_x^j$, the maximum gain resulting from placing up to $j$ independent facilities in $T_x$. If $x$ is a leaf, set $IndGain_x^0 = 0$ and $IndGain_x^j = w_x d_x$ for $j > 0$. Suppose that $x$ is not a leaf and $y, z$ are the sons of $x$. The recursive formulas for the gain functions are as follows:

$$IndGain_x^j \quad = \quad \max \left\{ \begin{array}{l} \max_{p+q=j} \left\{ IndGain_y^p + IndGain_z^q \right\} \\ \omega_x d_x \end{array} \right.$$

Algorithm IndDP clearly runs in time $O(k^2 n)$. The correctness is easy to justify: When we place independent facilities at $x$, either we place just one facility at $x$, or the facilities are split between $T_y$ and $T_z$. The gains from the subtrees of $x$ are additive.

# 5 A Faster Algorithm for $k = 2$

We now improve the upper bound for $k = 2$ to $O(n \log n)$. We have two subcases: the independent case, when the two facilities are independent, and the ancestral case, when one facility is an ancestor of the other. The independent case has been solved in Section 4, so we only consider the ancestral case.

**The ancestral case.** In the ancestral case, our task is to find $u$ and $v \in T_u$ that maximize the gain resulting from placing two facilities at $u, v$:

$$Gain(u, v) \quad = \quad d_u \omega_u + (d_v - d_u)\omega_v. \tag{1}$$

The algorithm for the ancestral case is based on the divide-and-conquer approach. At each step we make a recursive call to some tree $S$ to compute an ancestral pair $u, v$ that maximizes (1 among all pairs in $S$. (To avoid possible confusion, we emphasize that this pair is optimal with respect to the gain function for the whole tree $T$, not $S$.) This tree $S$ will always be a subtree of the input tree $T$, although in general it is not of the form $T_x$.

Let $x$ be the root of $S$. It is well-known – and easy to see – that $S$ always has a node $y$ such that $\frac{1}{3} \cdot |S| \leq |S_y| \leq \frac{2}{3} \cdot |S|$. We call this node $y$ a *centroid* of $S$. Using $y$, we can partition $S$ into disjoint subtrees $S_y$ and $S - S_y$. This gives rise to three possible locations of the optimal pair $u, v$: (i) $u, v \in S_y$, (ii) $u, v \in S - S_y$, or (iii) $u \in path(x, y)$ and $v \in S_y$, where $path(x, y)$ denotes the path from $x$ to $y$ in $S$ (See Figure 1), excluding $y$. Recursively, we can find optimal pairs in $S_y$ and $S - S_y$.

We now show how to solve the third case, that is, how to find an optimal pair that is split between $path(x, y)$ and $S_y$. Denote by $P = path(x, y)$ and $Q = S_y$ the lists of all ancestors (excluding $y$) and predecessor of $y$, respectively, ordered by decreasing tree weight $\omega$. Note that $P$ is then also ordered by increasing depth. (This is not necessarily true for $Q$, though.)

**Lemma 5** *Let $P = (p_1, p_2, \ldots, p_a)$ be a list of nodes ordered by increasing depth and $Q = (q_1, q_2, \ldots, q_b)$ a list of nodes ordered by decreasing tree weight. Then function $Gain(\cdot, \cdot)$ has the Monge property on $P \times Q$, that is, for all $i < j$ and $g < h$ we have*

$$Gain(p_i, q_g) + Gain(p_j, q_h) \quad \geq \quad Gain(p_i, q_h) + Gain(p_j, q_g)$$

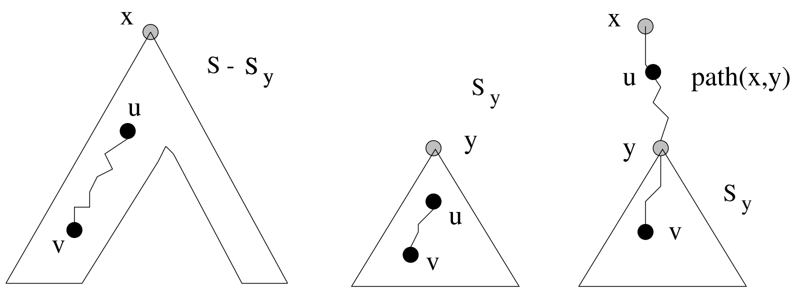


Figure 1: Three possible location of an optimal ancestral pair in a partitioned tree $S$.

*Proof:* Substituting, we get

$$\begin{aligned} Gain(p_i, q_g) + Gain(p_j, q_h) &= d_{p_i}\omega_{p_i} + [d_{q_g} - d_{p_i}]\omega)q_g \ + d_{p_j}\omega_{p_j} + [d_{q_h} - d_{p_j}]\omega_{q_h} \\ &= d_{p_i}\omega_{p_i} + [d_{q_h} - d_{p_i}]\omega_{q_h} \ + d_{p_j}\omega_{p_j} + [d_{q_g} - d_{p_j}]\omega_{q_g} \\ &\qquad + [d_{p_j} - d_{p_i}][\omega_{q_g} - \omega_{q_h}] \\ &\geq Gain(p_i, q_h) + Gain(p_j, q_g), \end{aligned}$$

because $d_{p_j} \geq d_{p_i}$ and $\omega_{q_g} > \omega_{q_h}$. $\Box$

**Procedure** AncSplitPair$(P, Q)$. The procedure receives on input two lists of vertices $P$ and $Q$ that satisfy the conditions of Lemma 5. It outputs a pair $\{u, v\}$, for $u \in P$ and $v \in Q$, that maximizes $Gain(u, v)$. To find $u$ and $v$, we simply apply the linear-time algorithm called SMAWK (see [1]) that computes maxima of functions with the Monge property.

**Procedure** AncPair$(S)$. The input $S$ is a tree with root $x$. The procedure produces two outputs: the optimal ancestral pair from $S$, and the list of all nodes of $S$ sorted by decreasing tree weight. If $S = \{x\}$, return $\{x, x\}$ and a one-element list $(x)$. If $|S| > 1$, we proceed as follows:

0. Find a centroid $y$ of $S$.
1. Call AncPair$(S_y)$ to compute the best pair $\{u_1, v_1\}$ in $S_y$ and the list $Q_1$ of the nodes in $S_y$ sorted by decreasing tree weight.
2. Call AncPair$(S - S_y)$ to compute the best pair $\{u_2, v_2\}$ in $S - S_y$ and the list $Q_2$ of the nodes in $S - S_y$ sorted by decreasing tree weight.
3. Call AncSplitPair$(P, Q_1)$, where $P = path(x, y)$, to compute the best pair $\{u_3, v_3\}$, where $u_3 \in path(x, y)$ and $v_3 \in S_y$.
4. Return the best of the three pairs $\{u_i, v_i\}$, $i = 1, 2, 3$.
5. Merge lists $Q_1$ and $Q_2$ and return the resulting sorted list.

**Algorithm** Fast2Median. Given a weighted tree $T$, we use Algorithm IndDP with $k = 2$ to find the best independent pair. Then we use Procedure AncPair to find the best ancestral pair. We output the better of these two.

**Theorem 4** *Algorithm* Fast2Median *computes* 2*-medians in directed trees in time* $O(n \log n)$.

*Proof:* The correctness of the algorithm is obvious. The independent case takes time $O(n)$. So it is sufficient to show that AncPair runs in time $O(n \log n)$.

Consider one recursive call of AncPair($S$), with $|S| = m$ and $|S_y| = m'$. The running time of AncPair($S$) satisfies the recurrence:

$$t(m) \quad = \quad t(m') + t(m - m') + O(m), \quad \text{where } m/3 \leq m \leq 2m/3.$$

since computing $y$, merging sorted sequences, and the call to AncSplitPair all take time $O(m)$. This immediately implies that the running time of AncPair is $t(n) = O(n \log n)$. $\square$

# 6 An Algorithm for $k = 3$

In this section we improve the time complexity for $k = 3$ to $O(n \log^2 n)$. We start by describing some auxiliary procedures.

**Best-descendant function.** First, we solve the following task. For each $u$, we want to compute $v \in T_u$ that maximizes $Gain(u, v) = d_u \omega_u + (d_v - d_u)\omega_v$. We refer to this $v$ as the *best descendant* of $u$ and denote it $BestDesc(u)$.

We compute the function $BestDesc(\cdot)$ by essentially the same algorithm as the one for solving the ancestral case of the 2-median problem, AncPair. Referring to the description of AncPair in Section 5, the following changes need to be made. The recursive call to AncPair($S$), instead of outputting just one optimal pair, computes $BestDesc(u)$ for all $u \in S$. As before, it also produces a list of nodes in $S$ sorted according to decreasing tree weight. After the two recursive calls to $S_y$ and $S - S_y$, we call a modified version of AncSplitPair($P, Q$) that, for each $u \in P$ computes an optimal $v \in Q$ that maximizes $Gain(u, v)$. This, again, is done using the SMAWK algorithm. For $u \in S_x - path(x, y)$ this gives us the desired values of $BestDesc(u)$. For $u \in path(x, y)$, $BestDesc(u)$ is the better of the nodes produced by the recursive call to AncPair($S - S_y$) and by AncSplitPair($P, Q_1$). The running time of this modification of AncPair is also $O(n \log n)$.

**Forbidden path.** We need another, rather technical procedure, similar to the one given above for computing $BestDesc(\cdot)$. In this case, the input is some tree $U$ with root $x$. ($U$ will be a subtree of $T$, but in general it will not have the form $T_x$.) We are also given a node $z \in U$. The goal is to compute, for each $u \in path(x, z)$ the best descendant $v$ of $u$ in $U - path(x, z)$ (the one that maximizes $Gain(u, v)$). We call $path(x, z)$ the *forbidden path.* Think of $path(x, z)$ as the "black sheep" of the family $S$. For each "black sheep" $u$ we want to compute a best descendant that is not a "black sheep."

We accomplish this task with a recursive procedure PathBestDesc($U, z$) based on the centroid partitions. We find a centroid $y$ of $U$ and we partition $U$ into $U_y$ and $U - U_y$. We now have two cases (see Figure 2).

Suppose that $y$ and $z$ are independent, and denote by $s$ their lowest common ancestor. We first make a recursive call to PathBestDesc($U - U_y, z$). This will give us the correct descendant for each $u \in path(x, z) - path(x, s)$. Then for all $u \in path(x, s)$ we compute the best descendant in $U_y$ using the SMAWK algorithm, and we pick the better of the two choices.

The other case is when $z \in U_y$. We first make a recursive call to PathBestDesc($U_y, z$). This will give us the correct descendant for each $u \in path(y, z)$. Next, we make a recursive call to

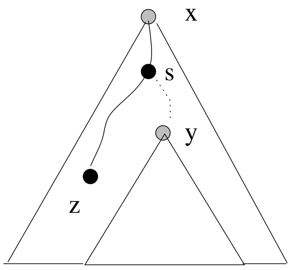


Figure 2: Two cases in computing descendants with a forbidden path.

PathBestDesc$(U - U_y, y')$, where $y'$ is the father of $y$. Then, for all $u \in path(x, y')$ we compute the best ancestor in $U_y - path(y, z)$ using the SMAWK algorithm. (Removing $path(y, z)$ does not affect the algorithm, we just need to change the second input list). This gives us up to two candidate descendants for all $u \in path(x, y')$ and we pick the better one.

Overall, if $|U| = m$, the above procedure in time $O(m \log m)$ for each $u \in path(x, z)$ will compute its best descendant that is not no $path(x, z)$.

**The algorithm.** We consider separately four cases corresponding to different genealogical relations between the facilities and chooses the optimal solution. We have four cases (see Figure 3):

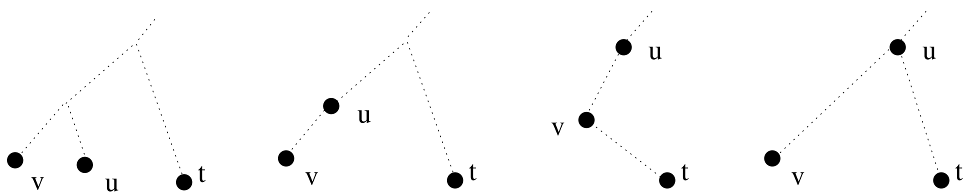


Figure 3: Four genealogical relations between three facilities.

*Independent:* All three facilities are independent.

*Partially independent:* A pair of nodes $u, v$, with $u$ being an ancestor of $v$, and one node $t$ independent of $u$ and $v$.

*Ancestral:* Three nodes $u, v, t$ on a path from root to a leaf.

*Partially ancestral:* A node $u$ with two independent descendants $v, t$.

To solve the independent case we can simply apply Algorithm IndDP from Section 4 with $k = 3$. We now describe how to solve the remaining three cases.

**Partially independent case.** To solve this case, we use the function $BestDesc(\cdot)$. We perform dynamic programming in a bottom-up order, similarly to the independent case from Section 4. For each node $x$ we compute three values: an optimal singleton in $T_x$, and optimal ancestral pair in $T_x$, and an optimal partially independent triple in $T_x$. Suppose we are now at $x$. The optimal singleton in $T_x$ is either $x$, or the optimal singleton in $T_y$, or the optimal singleton in $T_z$. The optimal ancestral pair is either $\{x, v\}$, for $v = BestDesc(x)$, or the optimal pair from $T_y$, or the optimal from $T_z$. The optimal partially independent triple is obtained by combining the optimal pair from one subtree with the optimal singleton from the other and adding up their gain values. This can be done in two ways, and we choose the better of the two triples.

**Ancestral case.** The totally ancestral case is solved, in essence, by applying the ancestral case of the 2-median algorithm twice. We first compute function $BestDesc(\cdot)$. With each $u$ and $v \in T_u$ we can associate a modified gain value:

$$Gain^*(u, v) \quad = \quad d_u\omega_u + (d_v - d_u)\omega_v + G(v) \tag{2}$$

where $G(v) = (d_t - d_v)\omega_t$, for $t = BestDesc(v)$. Calculations similar to the proof of Lemma 5 show that $Gain^*(u, v)$ has the Monge property. The maximum of $Gain^*(u, v)$ is equal to the maximum gain of an ancestral triple $\{u, v, t\}$. Thus we can solve this case by running the procedure $\mathsf{AncPair}(T)$ with the modified gain function $Gain^*(u, v)$ instead of $Gain(u, v)$.

**Partially ancestral case.** This is the most difficult case. The algorithm is again based on divide-and-conquer using the centroid tree partition. As before, a given tree $S$ is partitioned into two subtrees $S_y$ and $S - S_y$. There are four possibilities for the location of the optimal partially ancestral triple, see Figure 4. The first two cases are solved recursively. The third case is solved using the depth-based dynamic programming approach from Section 3. The last case is hardest. In this case $u \in path(x, y)$, one of its descendants $v$ is in $S - S_y - path(x, y)$ and the other descendant $t$ is in $S_y$. The idea is that we can compute $v$ and $t$ independently of each other. We compute $t$ using the SMAWK algorithm. To compute $v$, we call the procedure $\mathsf{PathBestDesc}(S - S_y, y)$ that finds the best descendant that does not lie on $path(x, y)$.

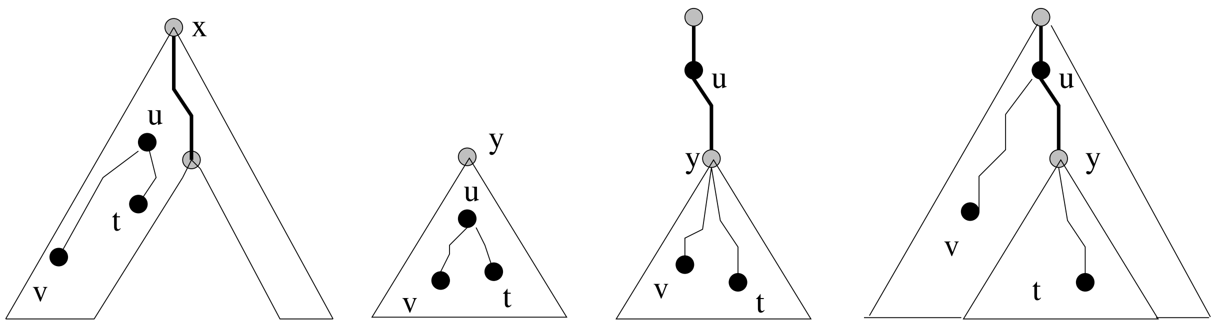


Figure 4: Four possible location of an optimal partially ancestral triple in a partitioned tree $S$.

We now describe the whole procedure in more detail. For a tree $S$ with root $x$, let $Gain^1_S(\alpha)$ denote the optimal gain of having one facility in $S$ given that there is another facility located at distance $\alpha$ above $x$. Similarly, let $IndGain^2_S(\alpha)$ denote the optimal gain of having two independent facilities in $S$ given that there is another facility located at distance $\alpha$ above $x$. (We slightly overload the notation introduced in Section 3 to avoid clutter).

**Procedure** PartAncTriple($S$)**.** The input $S$ is a tree with root $x$. The procedure produces three outputs. The first output is an optimal partially ancestral triple $\{u, v, t\}$ in $S$ (or nil if no such triple exists). The second output is the function $Gain^1_S(\cdot)$, and the third output is the function $IndGain^2_S(\cdot)$. Both functions are piece-wise linear functions, as explained in Section 3. For both functions, with each segment, we store the node (or the pair of nodes) that corresponds to this segment.

If $|S| = \{x\}$, return nil. If $|S| > 1$, we proceed as follows:

0. Find a centroid $y$ of $S$. Let $S' = S - S_y$.
1. Call PartAncTriple($S_y$). Let $\{u_1, v_1, t_1\}$ be the optimal triple in $S_y$.
2. Call PartAncTriple($S'$). Let $\{u_2, v_2, t_2\}$ be the optimal triple in $S'$.
3. We divide this step into three sub-steps.

   (a) Call PathBestDesc($S', y$) to compute, for each $u \in path(x, y)$ the best descendant $v$ of $u$ in $S' - path(x, y)$.

   (b) For each $u \in path(x, y)$ compute its best descendant $t \in S_y$. This again can be done using the SMAWK algorithm.

   (c) For each $u \in path(x, y)$ form a partially ancestral triple $\{u, v, t\}$ by using nodes $v$ and $t$ from the previous two steps. Among these triples, let $\{u_3, v_3, t_3\}$ be the triple that has the maximum gain value.
4. For each $u \in path(x, y)$ compute $IndGain_{S_y}(d_y - d_u)$. Let $u_4$ be the node that achieves this maximum, and $v_4, t_4$ the independent nodes in $S_y$ that correspond to the line segment that contains $\alpha = d_y - d_u$. Form a partially ancestral triple $\{u_4, v_4, t_4\}$.
5. Return the best of the triples $\{u_i, v_i, t_i\}$, $i = 1, 2, 3, 4$.
6. Compute and return function $Gain^1_S(\cdot)$. To determine $Gain^1_S(\cdot)$, we use the gain functions computed by the recursive calls in Step 1 and 2. We take $Gain^1_S(\alpha)$ to be the maximum of $Gain^1_{S_y}(\alpha + d_{yx})$ and $Gain^1_{S'}(\alpha)$.
7. Compute and return function $IndGain^2_S(\cdot)$. To determine $IndGain^2_S(\alpha)$, we compute the maximum of three functions that correspond to three possible locations of the independent facilities $v, t$ in $S$: $v, t \in S_y$, $v, t \in S'$, and $v \in S_y$ and $t \in S'$. Thus the first function is $IndGain^2_{S_y}(\alpha + d_{yx})$, the second function is $IndGain^2_{S'}(\alpha + d_{yx})$, and the third function is $Gain^1_{S_y}(\alpha + d_{yx}) + Gain^1_{S'}(\alpha)$. All functions needed for these computations were computed by the recursive calls.

**Algorithm** Fast3Median**.** Given a weighted tree $T$, we compute separately the optimal triples for each of the four genealogical patterns in Figure 3. The independent case was solved in Section 4. The algorithms for the other three cases were described above.

**Theorem 5** *Algorithm* Fast3Median *computes* 3*-medians in directed trees in time* $O(n \log^2 n)$.

*Proof:* In the independent case, algorithm IndDP takes time $O(n)$. To solve the partially independent case, we compute $BestDesc(\cdot)$ in time $O(n \log n)$ and the remaining computation takes time $O(n)$. The ancestral case takes time $O(n \log n)$.

We now consider the partially ancestral case, Procedure PartAncTriple. Consider a recursive call to a tree $S$. Let $m = |S|$ and $m' = |S_y|$. The recursive calls take time $t(m')$ and $t(m - m')$. The invocation of PathBestDesc$(S', y)$ in Step 3 executes in time $O(m \log m)$. The rest of the computation in this step requires only time $O(m)$. Steps 4-7 take time $O(m)$. So, overall, the running time satisfies the recurrence:

$$t(m) \quad = \quad t(m') + t(m - m') + O(m \log m), \quad \text{where } m/3 \leq m \leq 2m/3.$$

This immediately implies that the running time of PartAncTriple is $t(n) = O(n \log n)$. $\Box$

# 7 Final Comments

We have presented an $O(n \log n)$-time algorithm for 2-medians and an $O(n \log^2 n)$-time algorithm for 3-medians in directed trees, improving the $O(n^2)$ upper bound for these cases. Many open problems remain.

Can the 2-median problem be solved in linear time? The answer is positive if the tree is a path. However, for general trees, we remain skeptical. The hard case appears to be a tree with a long path from the root $r$ to a node $y$ and a "bushy" subtree $T_y$. If $T_y$ is sorted by decreasing tree weight, we can apply the SMAWK algorithm to get a solution in linear time. So the problem appears to reduce to the question of whether sorting is necessary.

What is the complexity of the problem for general $k$? We conjecture that it can be solved in time $O(c_k n \log^a n)$, where $a$ is a small constant and $c_k$ is a constant that depends only on $k$. An analogous question can be asked for the undirected case. We believe that our approach from Section 6 can be used to get an $O(n \log^2 n)$-time algorithm for this problem in case $k = 3$.

The linearity conjecture remains open for $k \geq 3$. We believe that getting good estimates on the number of minimizers of the cost function would help to get better upper bounds on the complexity of the $k$-median problem. The linearity conjecture is closely related to some problems in combinatorial geometry. For example, if $T$ is a star, that is, a tree consisting only of a root with $n - 1$ sons, then the problem is equivalent to counting the number of sets consisting of $k$ top lines in an arrangement of $n - 1$ lines, which is known to be $O(kn)$. We suspect that nearly linear upper bounds can be obtained using Davenport-Schinzel sequences [7, 17], but so far we were able to apply this method only for $k = 2$.

# References


[1] A. Aggarwal, M.M. Klawe, S. Moran, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for euclidean k-medians and related problems. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 106–113, 1998.

[3] V. Auletta, D. Parente, and G. Persiano. Dynamic and static algorithms for optimal placement of resources in a tree. *Theoretical Computer Science*, 165:441–461, 1996.

[4] V. Auletta, D. Parente, and G. Persiano. Placing resources on a growing line. *Journal of Algorithms*, 26:87–100, 1998.

[5] M. Charikar and S. Guha. Improved combinatorial algorithms for facility location and $k$-median problems. In *Proc. 40th Symposium on Foundations of Computer Science (FOCS'99)*, pages 378–388, 1999.

[6] M. Charikar, S. Guha, E. Tardos, and D. Shmoys. A constant-factor approximation algorithm for the k-median problem. In *Proc. 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, 1999.

[7] H. Davenport and A. Schinzel. A combinatorial problem connected with differential equations. *American J. Math.*, 87:684–694, 1965.

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. W.H. Freeman and Co., 1979.

[9] R. Gavish and S. Sridhar. Computing the 2-median on tree networks in $O(n \log n)$ time. *Networks*, 26:305–317, 1995.

[10] M.H. Halldorson, K. Iwano, N. Katoh, and T. Tokuyama. Finding subsets maximizing minimum structures. In *Proc. 6th Annual Symposium on Discrete Algorithms (SODA'95)*, pages 150–157, 1995.

[11] R. Hassin and A. Tamir. Improved complexity bounds for location problems on the real line. *Operation Research Letters*, 10:395–402, 1991.

[12] W.L. Hsu. The distance-domination numbers of trees. *Operation Research Letters*, 1:96–100, 1982.

[13] O. Kariv and S.L. Hakimi. An algorithmic approach to network location problems II: The p-medians. *SIAM Journal on Applied Mathematics*, 37:539–560, 1979.

[14] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. *Journal of Algorithms*, 37:146–188, 2000.

[15] B. Li, X. Deng, M. Golin, and K. Sohraby. On the optimal placement of web proxies on the internet: linear topology. In *Proc. 8th IFIP Conference on High Peformance Netwworking (HPN'98)*, pages 00–00, 1998.

[16] B. Li, M.J. Golin, G.F. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *IEEE InfoComm'99*, pages 1282–1290, 1999.

[17] M. Sharir and P.K. Agarwal. *Davenport-Schinzel sequences and their geometric applications.* Cambridge University Press, 1995.

[18] A. Tamir. An $O(pn^2)$ algorithm for the $p$-median and related problems on tree graphs. *Operations Research Letters*, 19:59–64, 1996.

[19] A. Vigneron, L. Gao, M. Golin, G. Italiano, and B. Li. An algorithm for finding a $k$-median in a directed tree. *Information Processing Letters*, 74:81–88, 2000.

[20] G. Woeginger. Monge strikes again: optimal placement of web proxies in the internet. *Operations Research Letters*, 27:93–96, 2000.