An Algorithm for Finding a k-Median in a Directed Tree

Antoine Vigneron * *Lixin Gao* † *Mordecai J. Golin* ‡
Giuseppe F. Italiano § *Bo Li* ¶

Abstract

We consider the problem of finding a *k-median* in a directed tree. We present an algorithm that computes a k-median in $O(Pk^2)$ time where k is the number of resources to be placed and P is the path length of the tree. In the case of a balanced tree, this implies $O(k^2 n \log n)$ time, in a random tree $O(k^2 n^{3/2})$, while in the worst case $O(k^2 n^2)$. Our method employs dynamic programming and uses $O(nk)$ space, while the best known algorithms for undirected trees require $O(n^2 k)$ space.

Keywords: Algorithms, Analysis of Algorithms, Combinatorial Problems, Dynamic Programming, k-Median Problem.

---

*École Polytechnique, Paris, France. Most of this work was done while visiting the Hong Kong University of Science & Technology. Research partially supported by HK RGC CERG Grant HKUST652/95E. Email: `antoine.vigneron@polytechnique.org`.

†Dept of Computer Science, Smith College. Email: `gao@cs.smith.edu`. Research supported in part by NCR-9977555 and NSF CAREER Award grant ANI-9875513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

‡Dept of Computer Science, Hong Kong UST, Kowloon, Hong Kong. Research partially supported by HK RGC CERG Grants HKUST652/95E and 6082/97E. Email: `golin@cs.ust.hk`. URL: `http://www.cs.ust.hk/~golin`.

§Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Italy. Research partially supported by Italian Ministry of University and Scientific and Technological Research under the Project "Algorithms for Large Data Sets: Science and Engineering". Most of this work was done while visiting the Hong Kong University of Science & Technology. Email: `italiano@info.uniroma2.it`. URL: `http://www.info.uniroma2.it/~italiano`.

¶Dept of Computer Science, Hong Kong UST, Kowloon, Hong Kong. Email: `bli@cs.ust.hk`.

1 Introduction

Let G be a network, with weights on nodes and lengths on edges. The network also contains resources, or service centers, on some of the nodes. The *cost* of servicing a request from node $v \in G$ is the length of the shortest path from v to a service center. The cost of the network is the sum over all nodes of the weight of the node times the cost of servicing the node. The k-median problem is to find the placement of k resources in nodes of G that achieves the smallest cost over all placements of k resources. Kariv and Hakimi [1] proved that this problem is NP-hard in the case of general graphs, and proposed an $O(n^2k^2)$ algorithm for undirected trees, where n is the number of tree nodes and k the number of resources. Auletta *et al.* [2] studied the same problem on a line, and found an $O(nk)$ algorithm for this special case. Recently, Li *et al.* [3] proposed an $O(n^2k)$ algorithm for the same problem on a *directed* line.

In this paper, we consider the k-median problem on a *directed* tree, where edges are directed from child to parent. Our motivation is the optimal placement of cache proxies in a computer network [3]. A node v makes a service request, e.g., an http request. This request moves up the tree towards the root, stopping when a proxy is found; if this location is near v then the request can be satisfied almost immediately. This is similar to web caching, which is currently commonly used. In this application, the weight of a node represents the frequency of the requests coming up from that node, the distance between a child and parent nodes is related to several parameters for the corresponding link (e.g., distance, traffic, congestion) between the two nodes and the cost function to be minimized is the overall traffic in the network.

The main contribution of this paper is an algorithm for computing the k-median on a directed tree in $O(Pk^2)$ worst-case time and $O(nk)$ space, where P is the *path length* of the tree. The path length P is a well known quantity in combinatorics, and is defined as the sum over the whole tree of the number of ancestors for each tree node, i.e., the number of edges on the path from the node to the root. For a balanced binary tree with n nodes P is $\Theta(n \log n)$, for random general trees P is $\Theta(n\sqrt{n}\,)$ [4, p. 245] and in the worst case $P = \Theta(n^2)$.

We mention that a non-trivial adaptation of the algorithm by Kariv and Hakimi [1] could achieve the same time bounds as our algorithm but using $O(Pk)$ space (details are in the appendix). It is not clear, however, how to reduce their space usage. We believe this to be an important consideration in practice, as our experience shows that space clearly becomes the main bottleneck for the applicability of these algorithms. We implemented our algorithm in C and tested it on random trees. On an Indigo 2 Silicon Graphics workstation with 64MB of RAM, we were able to solve problem sizes of $n = 2,000$ nodes and $k = 100$

proxies in about 1 minute and $n = 5,000$ nodes and $k = 100$ proxies in about 4 minutes. (However, the case of $n = 10,000$ nodes and $k = 100$ proxies required over 4 hours of computing time due to memory (swap) problems.) A preliminary version of our algorithm, requiring $O(Pk) = O(n^2 k)$ space, failed to solve instances of even less than $n = 1,000$ nodes and $k = 100$ proxies on the same machine.

2 Notations and Preliminaries

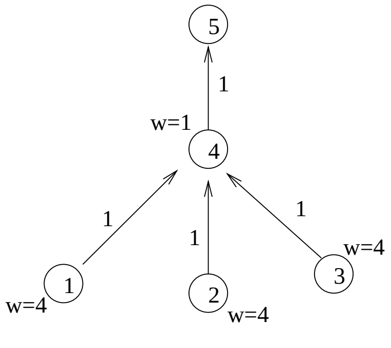
Let T be a rooted tree, all of whose edges are directed upwards towards root u_0. For u a node of T we denote by T_u the subtree rooted at u (note that $T = T_{u_0}$). We also denote by $|V|$ the cardinality of a set V, and by $|G|$ the number of nodes of a graph G. Each tree node u has weight $w(u)$ and each edge (u, v) a length $d(u, v)$. If v is an ancestor of u, then we extend $d(u, v)$ to denote the sum of the lengths on the directed path from u to v. (We remark that our algorithm still works even if $d()$ and $w()$ are arbitrary functions on an ordered ring.) Let $V \subset T_u$ be a set of nodes, which we call *proxies*, such that $u \in V$. If v is a tree node, let v' be the closest ancestor of v belonging to V. We define the cost of T_u associated with V as follows:

$$cost(T_u, V) = \sum_{v \in T_u} w(v) d(v, v')$$

The minimum cost of placing k proxies in T_u is

$$\|T_u\|_k = \min_{|V|=k, u \in V} \{cost(T_u, V)\}.$$

$V_{u,k}$ is set to be such that $\|T_u\|_k = cost(T_u, V_{u,k})$. Figure 1 illustrates a tree and its associated values.



$k = 1:$	$V_{5,1} = \{5\}$	$\|T_5\|_1 = 25$
$k = 2:$	$V_{5,2} = \{5, 4\}$	$\|T_5\|_2 = 12$
$k = 3:$	$V_{5,3} = \{5, 4, 3\}$	$\|T_5\|_3 = 8$
$k = 4:$	$V_{5,4} = \{5, 3, 2, 1\}$	$\|T_5\|_4 = 1$
$k = 5:$	$V_{5,5} = \{5, 4, 3, 2, 1\}$	$\|T_5\|_5 = 0$

Figure 1. A tree and its associated values.

Our algorithm uses dynamic programming, and it is based upon a careful decomposition of the input tree. In particular, the tree is split into three parts: a right part, which is assumed to be empty of proxies, so that its contribution to the total cost can be easily computed; a middle part, which is a subtree whose cost has already been computed; and a left part whose contribution is computed recursively. We now provide the details of this decomposition.

2.1 Ordering and Splitting the Tree

We start by ordering the tree nodes according to a postorder traversal of the tree, as shown in Figure 2. In the remainder of our description we will identify each node of T with its postorder numbering, i.e., a distinct integer in the closed interval $[1, n]$.

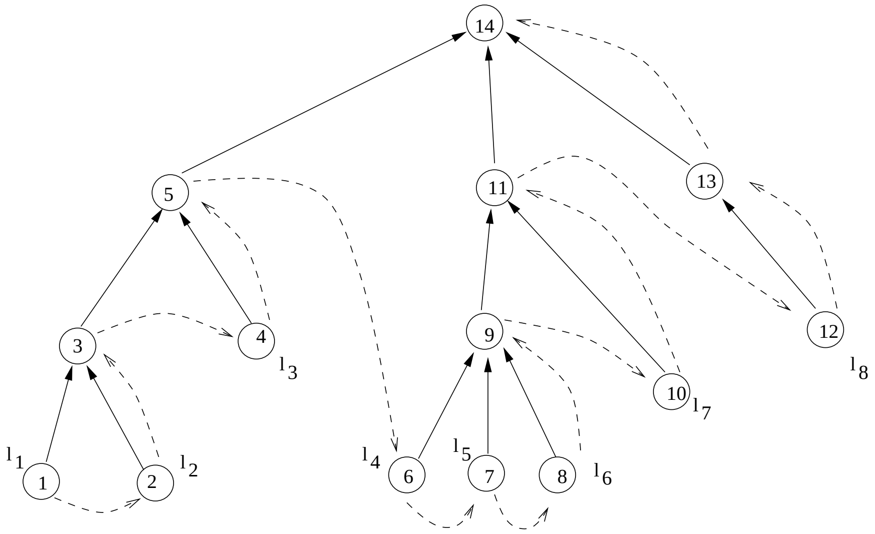


Figure 2. Ordering the nodes.

Let $l_1 < l_2 < ... < l_r$ be the set of leaves of T. Denote by m_v the lowest numbered node of T_v. Notice that m_v must be a leaf; in particular it is the leftmost leaf of T_v. Let $l_i = m_v$: if $i \neq 1$ we define $m'_v = l_{i-1}$ (i.e., the leaf immediately preceding m_v in the ordering). As an example, in Figure 2, $m_{11} = 6$ and $m'_{11} = 4$. It can be easily proved that for all vertices $v \in T,\ T_v = [m_v, v]$.

Let $v \in T_u$. Our dynamic programming algorithm works with the following decomposition of the tree, as illustrated in Figure 3. Note that '[' and ']' represent closed interval boundaries, '(' and ')', open ones:

$$L_{u,v} = [m_u, m_v), \qquad T_v = [m_v, v], \qquad R_{u,v} = (v, u].$$

We remark that $L_{u,v}$ depends only on m_v and not v. In particular $L_{u,v} = L_{u,m_v}$. We define the following extra costs associated with these three parts:

. $\|R_{u,v}\|_1 = \sum_{x \in R_{u,v}} d(x, u)w(x)$ is the cost of $R_{u,v}$, assuming that u is its only proxy.

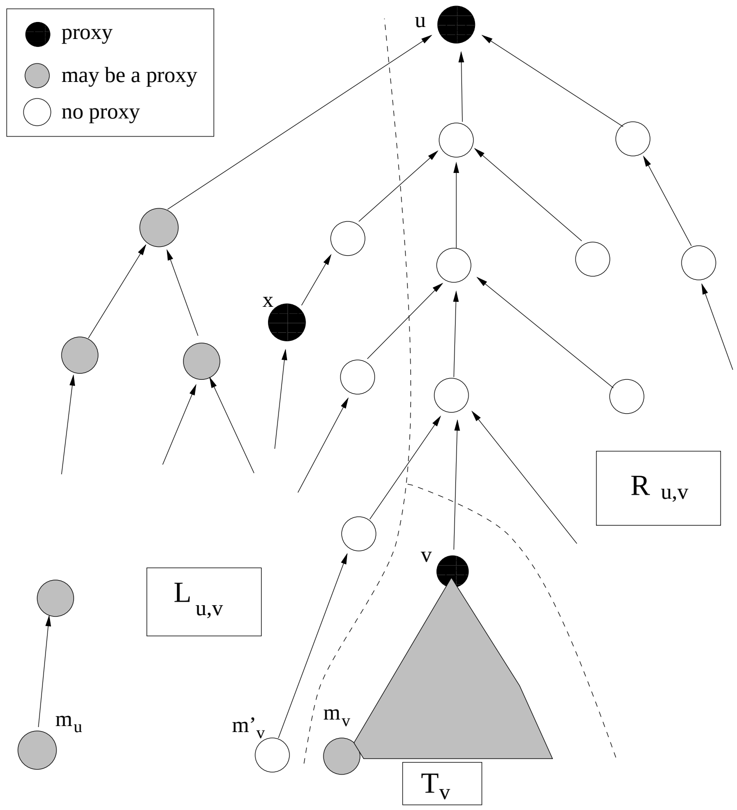


Figure 3. The decomposition of the tree into three parts.

- $\|T_{u,v}\|_0 = \sum_{x \in T_v} d(x,u)w(x)$ is the cost of T_v, assuming that T_v contains no proxies and that v is the proxy servicing all of T_v.
- $\|L_{u,v}\|_t$ is the min cost of $L_{u,v}$, assuming that there are t proxies in $L_{u,v}$ and that u is the only proxy in $R_{u,v}$ (if $m_u = m_v$ then $L_{u,v} = \emptyset$ and $\|L_{u,v}\|_t = 0$).

Note that $\|L_{u,v}\|_t$ is well defined (as there is no directed path from any node of $L_{u,v}$ to any node of T_v) and thus the cost of $L_{u,v}$ does not depend on the proxies of T_v.

2.2 Dynamic Programming Recurrences

In this section we derive the recurrence relations upon which the correctness of our algorithm hinges.

Lemma 2.1. *Let u be any vertex in T and t any positive integer, $1 < t \leq |T_u|$. Then*

$$\|T_u\|_{t>1} = \min_{\substack{v \in T_u - \{u\} \\ 1 \leq t' < t \\ t - |L_{u,v}| - 1 \leq t' \leq |T_v|}} \{\|T_v\|_{t'} + \|L_{u,v}\|_{t-t'-1} + \|R_{u,v}\|_1\}$$

Proof: Assume that we know v, the highest numbered proxy in T_u different from u. Then the minimum cost of placing t proxies in T_u is found by assuming that T_v contains t' proxies, $L_{u,v}$ contains $t - t' - 1$ proxies, calculating the best possible cost under this assumption and then minimizing over t. Note that each of the two sets must contain an optimal proxy placement so the minimum cost of placing t proxies in T_u is

$$\min_{\substack{1 \le t' < t \\ t - |L_{u,v}| - 1 \le t' \le |T_v|}} \{\|T_v\|_{t'} + \|L_{u,v}\|_{t-t'-1} + \|R_{u,v}\|_1\}.$$

Each set can not contain more proxies than its size so $t' \le |T_v|$ and $t - t' - 1 \le |L_{u,v}|$ Since we have no a-priori information about v we must then minimize over all possible values of v. $\square$

Lemma 2.2. *Let u, v be any vertices in T such that u is an ancestor of v and let t be any positive integer smaller than $|L_{u,v}|$. Then*

$$\|L_{u,v}\|_{t>0} = \min_{\substack{x \in L_{u,v} \\ 1 \le t' \le t \\ t - |L_{u,x}| \le t' \le |T_x|}} \{\|T_x\|_{t'} + \|L_{u,x}\|_{t-t'} + \|R_{u,x}\|_1 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0\}$$

Proof: We must compute the best placement of t proxies that minimizes the cost of $L_{u,v}$. Assume we know x, the highest numbered proxy in $L_{u,v}$. Then $L_{u,v}$ is split into three parts, as illustrated in Figure 4. The middle part is T_x which we assume contains t' proxies, so $t' \le |T_x|$. This leaves $t - t'$ proxies in the left part, $L_{u,x}$, so $t - t' \le |L_{u,x}|$. The right part is $R_{u,x} - R_{u,v} - T_{u,v}$; since it is empty of proxies, its contribution to the cost is $\|R_{u,x}\|_1 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0$. In order to find the contribution of the middle and the left parts, we must minimize over t' the sum $\|T_x\|_{t'} + \|L_{u,x}\|_{t-t'}$. Finally, we must minimize this value over all choices of x. $\square$

Taking into account the definitions of $L_{u,v}$ and $L_{u,x}$ the statements in Lemmas 2.1 and 2.2 can actually be written slightly differently:

$$\|T_u\|_{t>1} = \min_{\substack{m_u \le v < u \\ 1 \le t' < t \\ t - |L_{u,v}| - 1 \le t' \le |T_v|}} \{\|T_v\|_{t'} + \|L_{u,m_v}\|_{t-t'-1} + \|R_{u,v}\|_1\} \tag{1}$$

$$\|L_{u,v}\|_{t>0} = \min_{\substack{m_u \le x < m_v \\ 1 \le t' \le t \\ t - |L_{u,x}| \le t' \le |T_x|}} \{\|T_x\|_{t'} + \|L_{u,m_x}\|_{t-t'} + \|R_{u,x}\|_1 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0\} \tag{2}$$

We now derive another recurrence relation that allows us to compute $\|L_{u,v}\|_t$ more efficiently. It effectively permits reducing the range of x over which $\|L_{u,v}\|_t$ must be minimized.

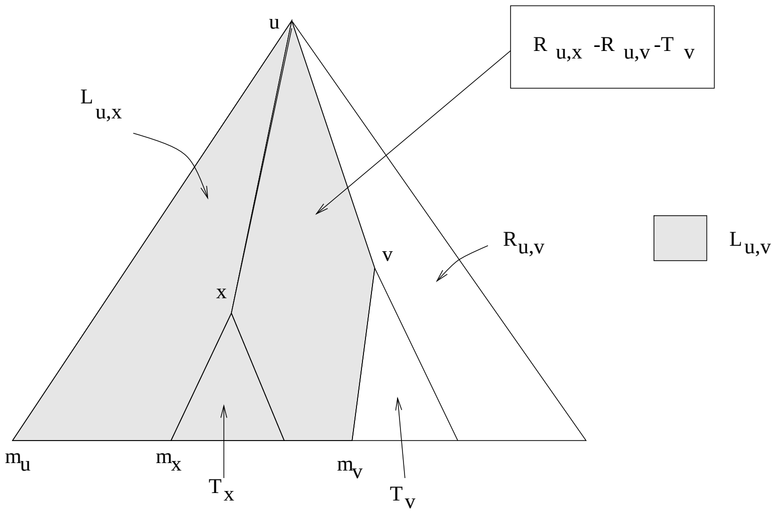


Figure 4. Computing $\|L_{u,v}\|_t$ as in Lemma 2.2.

Theorem 2.3. *Let u be any vertex in T, let $v \in T_u$ such that $m_v \neq m_u$. Then*

$$\|L_{u,v}\|_t = \min\{A_{u,v,t}, B_{u,v,t}\} \tag{3}$$

where

$$A_{u,v,t} = \|L_{u,m'_v}\|_t + \|R_{u,m'_v}\|_1 + \|T_{u,m'_v}\|_0 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0,$$

$$B_{u,v,t} = \min_{\substack{m'_v \leq x < m_v \\ 1 \leq t' \leq t \\ t-|L_{u,x}| \leq t' \leq |T_x|}} \{\|T_x\|_{t'} + \|L_{u,x}\|_{t-t'} + \|R_{u,x}\|_1 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0\}.$$

Proof: Recall that

$$\|L_{u,m'_v}\|_t = \min_{\substack{m_u \leq x < m'_v \\ 1 \leq t' \leq t \\ t-|L_{u,x}| \leq t' \leq |T_x|}} \{\|T_x\|_{t'} + \|L_{u,x}\|_{t-t'} + \|R_{u,x}\|_1 - \|R_{u,m'_v}\|_1 - \|T_{u,m'_v}\|_0\}.$$

In the definition of $A_{u,v,t}$ replace $\|L_{u,m'_v}\|_t$ by the right hand side of this equation. After some algebra, we obtain:

$$A_{u,v,t} = \min_{\substack{m_u \leq x < m'_v \\ 1 \leq t' \leq t \\ t-|L_{u,x}| \leq t' \leq |T_x|}} \{\|T_x\|_{t'} + \|L_{u,x}\|_{t-t'} + \|R_{u,x}\|_1 - \|R_{u,v}\|_1 - \|T_{u,v}\|_0\}.$$

The proof of the theorem then follows directly from (2). $\square$

3 The Algorithm

We now present an algorithm that computes $\|T_{u_0}\|_k$ and $V_{u_0,k}$ in $O(Pk^2)$ time and $O(nk)$ space, where n is the number of nodes in T_{u_0}, and P is the *path length* of tree T, which is defined as the sum over T of the number of ancestors of each node. Our algorithm can be divided into two phases. In the first phase, we compute $\|T_u\|_t$ for each node u and $1 \le t \le k$ via dynamic programming. In the second phase, we compute the optimal set of proxies $V_{u_0,k}$. We start by describing the first phase.

L1: Order the nodes of the tree; $\forall v$ compute m_v and m'_v, as defined in Section 2.1.

L2: For $u = 1$ to n do

- L3: *Initialization*: For all $v \in T_u$, compute $\|T_v\|_1$, $\|T_{u,v}\|_0$, $\|R_{u,v}\|_1$ and $\|L_{u,v}\|_0$.
- L4: For $t = 2$ to k do
  - L5: For $v = m_u + 1$ to $u - 1$ do
    - L6: If $v = m_v$ compute $\|L_{u,v}\|_{t-1}$ using (3)
  - L7: Compute $\|T_u\|_t$ using (1).
- L8: Release the memory used, except for $\|T_v\|_t, v \in [1..u], t \in [1..k]$.

A couple of remarks are in order. First, notice that in line L6 whenever $t - 1 > |L_{u,v}|$, it is not necessary to compute $\|L_{u,v}\|_{t-1}$ (since it is not defined). Second, in line L7 it is not necessary to compute $\|T_u\|_t$ for $t > |T_u|$.

Theorem 3.1. *The First Phase requires $O(Pk^2)$ time and $O(nk)$ space.*

Proof: We first analyze the worst-case running time. The first step (L1) can be straightforwardly implemented in linear time. Step (L3) can be done in $O(|T_u|)$ time using dynamic programming. Line (L7) is $O(k|T_u|)$. The loop (L5) essentially enumerates all the possible values of m_v in T_u following the order we defined in Section 2.1 and calculates $\|L_{u,m_v}\|_{t-1}$. By recurrence (3), this computation takes $O((m_v - m'_v)t)$. Let the leaves of T_u be $l_a < l_{a+1} < ... < l_b$. Then the running time of loop (L5) is $O(t\sum_{i=a+1}^{b}(l_i - l_{i-1})) = O(t(l_b - l_a)) < O(t(u - m_u))$ and therefore loop (L5) requires overall $O(t|T_u|)$ time. As loop (L4) requires overall $O(k^2|T_u|)$ time, the total running time of the algorithm is $O(k^2 \sum_{u \in T} |T_u|) = O(Pk^2)$, where P is the path length of T.

We next analyze the space usage. The program needs to store all values $\|T_u\|_t$, using $O(nk)$ space in total. For fixed current u and each v, $\|T_{u,v}\|_0$,$\|R_{u,v}\|_1$ and $\|L_{u,v}\|_0$ have to be stored, using $O(n)$ space. We also need to store $\|L_{u,v}\|_t$ for current u and all the possible values of t and v, using $O(nk)$. □

In the second phase, we recursively compute $V_{u_0,k}$ using the values of $\|T_u\|_t$ calculated and stored during the first phase. We use $(x_{u,v,t}, c_{u,v,t})$ to denote the pair (x, t') that achieves the minimum of $\|L_{u,v}\|_t$ in recurrence (3).

The second phase is implemented by a recursive procedure $Proxies(u, t)$, which returns $V_{u,t}$. We now describe how this procedure works. If $t = 1$ it simply returns $\{u\}$. Otherwise, $t > 1$ and it searches for the highest numbered proxy, according to the order given in Section 2.1. We call this node $N_{u,1}$ and denote the number of proxies in $T_{N_{u,1}}$ by $C_{u,1}$ (see Figure 5). Both $N_{u,1}$ and $C_{u,1}$ can be computed using recurrence (1). Next, we define $N_{u,2}$ as being the highest node of $V_{u,t}$ that does not belong to $T_{N_{u,1}} \cup \{u\}$, and $C_{u,2}$ as the number of proxies of $T_{N_{u,2}}$. We compute the values $(N_{u,i}, C_{u,i}), i \geq 1$ until we reach $C_{u,1} + C_{u,2} + ... + C_{u,h} = t - 1$. Finally, $Proxies$ returns $\{u\} \cup Proxies(C_{u,1}, N_{u,1}) \cup Proxies(C_{u,2}, N_{u,2}) \cup \ldots \cup Proxies(C_{u,h}, N_{u,h})$.

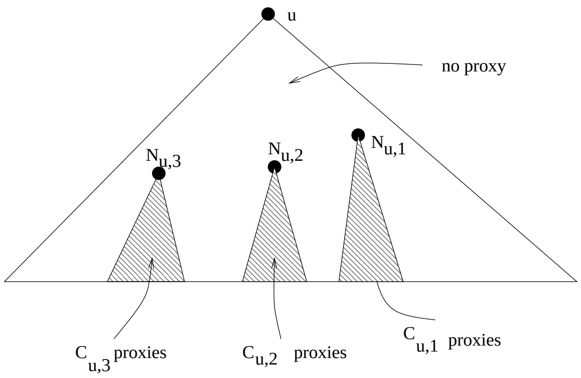


Figure 5. The Second Phase.

The algorithm pseudocode follows. Notice that lines (L2)–(L5) are analogous to the first phase. We further assume that the values of $\|T_u\|_t$, m_v and m'_v computed during the first phase are still available.

$Proxies(u, t)$

L1: If $t = 1$ then return $\{u\}$;

L2: *Initialization*: For each $v \in T_u$, compute $\|T_v\|_1$, $\|T_{u,v}\|_0$,$\|R_{u,v}\|_1$ and $\|L_{u,v}\|_0$.

L3: For $t = 2$ to k do

L4: For $v = m_u + 1$ to $u - 1$ do

L5: If $v = m_v$ compute $\|L_{u,v}\|_{t-1}$,$x_{u,v,t-1}$ and $c_{u,v,t-1}$ using (3).

L6: Compute $\|C_{u,1}\|$ and $\|N_{u,1}\|$ using (1).

L7: $i \leftarrow 2$; $t \leftarrow t - C_{u,1} - 1$;

L8: if $t = 0$ goto L12:

L9: $(C_{u,i}, N_{u,i}) = (x_{u,N_{u,i-1},t}, c_{u,N_{u,i-1},t})$

L10: $t \leftarrow t - C_{u,i}$; $i \leftarrow i + 1$

L11: goto L8:

L12: Release memory except for $\|T_{u,t}\|, \forall u, \forall t$ and $(C_{u,i}, N_{u,i}), \forall i$.

L13: Return $\{u\} \cup Proxies(N_{u,i-1}, C_{u,i-1}) \cup Proxies(N_{u,i-2}, C_{u,i-2}) \cup \ldots \cup Proxies(N_{u,1}, C_{u,1})$

Theorem 3.2. *The Second Phase requires $O(Pk^2)$ time and $O(nk)$ space.*

Proof: The first part (lines (L2)–(L6)) is executed as in the first phase, and thus requires at most $O(Pk^2)$ time by Theorem 3.1. Loop (L8)–(L11) is executed at most k times, since each computation of $(N_{u,i}, C_{u,i})$ implies a recursive call to *Proxies* on line (L13). The running time of lines (L8)–(L11) is $O(nk)$ (due to the cost of computing recurrence (3)), and thus the overall cost of (L8)–(L13) is $O(nk^2)$.

As for the space usage, in the second phase we need to store the same variables stored in the first phase plus the arrays $(N_{u,i}, C_{u,i})$ and $(c_{u,v,t}, x_{u,v,t})$. Since we only need to store these values for the current u, it uses overall $O(nk)$ space. □

References

[1] O. Kariv, S.L. Hakimi. "An Algorithmic Approach to Network Location Problems. II: The p-Medians". *SIAM. J. Appl. Math.*, 37 (1979), 539–560.

[2] V. Auletta, D. Parente, G. Persiano. "Placing Resources on a Growing Line". *Journal of Algorithms,* 26 (1998), 87–100.

[3] B. Li, X. Deng, M.J. Golin, K. Sohraby. "On The Optimal Placement of Web Proxies on the Internet", 8th IFIP Conference on High Performance Networking (HPN'98). (September 1998).

[4] Robert Sedgewick and Philippe Flajolet, *An Introduction to the Analysis of Algorithms,* Addison-Wesley, Reading, Mass.. (1996).

Appendix: Another Algorithm

In this appendix we briefly describe the adaptation, mentioned in the introduction, of Kariv and Hakimi's [1] algorithm that achieves the same time complexity of $O(Pk^2)$ but requires $O(Pk)$ space.

A node u is said to *cover* the subtree T_v if it is the closest ancestor of v that hosts a proxy. $C_{u,v,t}$ denotes the contribution of T_v to the overall cost knowing both that u covers T_v and that T_v contains t proxies. This quantity is computed for every triple (u, v, t) such that $t < k$ and u is an ancestor of v. This can be done by a postorder traversal of the tree according to v as in the previous algorithm.

Suppose $v_1, v_2, ...v_i$ are the sons of v. Then a straightforward derivation yields:

$$C_{u,v,t} = \min(A_{u,v,t}, B_{v,t}) \tag{4}$$

where

$$A_{u,v,t} = \min_{t_1+t_2+...+t_i=t}(C_{u,v_1,t_1} + C_{u,v_2,t_2} + ... + C_{u,v_i,t_i}) \tag{5}$$

and

$$B_{v,t} = \min_{t_1+t_2+...+t_i=t-1}(C_{v,v_1,t_1} + C_{v,v_2,t_2} + ... + C_{v,v_i,t_i}) \tag{6}$$

The minimizations in (5) and (6) can be performed in $O(it)$ time by recursively computing the contribution of $T_{v_1} \cup T_{v_2} ... \cup T_{v_j}$ for increasing j (since this quantity can be obtained in $O(t)$ time when the contribution of $T_{v_1} \cup T_{v_2} ... \cup T_{v_{j-1}}$ is known). Performing the minimizations by doing a postorder traversal of the tree by v, and then for each v running over all $u \in T_v$ and all t, yields a $O(Pk^2)$ running time.