View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

An $O(n \log n)$-Time Algorithm for the k-Center Problem in Trees

Haitao Wang
Department of Computer Science, Utah State University
Logan, UT 84322, USA
haitao.wang@usu.edu

Jingru Zhang
Department of Computer Science, Marshall University
Huntington, WV 25705, USA
jingru.zhang@marshall.edu

Abstract

We consider a classical k-center problem in trees. Let T be a tree of n vertices and every vertex has a nonnegative weight. The problem is to find k centers on the edges of T such that the maximum weighted distance from all vertices to their closest centers is minimized. Megiddo and Tamir (SIAM J. Comput., 1983) gave an algorithm that can solve the problem in $O(n \log^2 n)$ time by using Cole's parametric search. Since then it has been open for over three decades whether the problem can be solved in $O(n \log n)$ time. In this paper, we present an $O(n \log n)$ time algorithm for the problem and thus settle the open problem affirmatively.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Theory of computation → Computational geometry

Keywords and phrases k-center, trees, facility locations

Digital Object Identifier 10.4230/LIPIcs.SoCG.2018.72

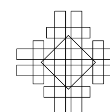
Related Version A full version of this paper is available at `https://arxiv.org/abs/1705.02752`

1 Introduction

In this paper, we study a classical k-center problem in trees. Let T be a tree of n vertices. Each edge $e(u, v)$ connecting two vertices u and v has a positive length $d(u, v)$, and we consider the edge as a line segment of length $d(u, v)$ so that we can talk about "points" on the edge. For any two points p and q of T, there is a unique path in T from p to q, denoted by $\pi(p, q)$, and by slightly abusing the notation, we use $d(p, q)$ to denote the length of $\pi(p, q)$. Each vertex v of T is associated with a weight $w(v) \geq 0$. The *k-center problem* is to compute a set Q of k points on T, called *centers*, such that the maximum weighted distance from all vertices of T to their closest centers is minimized, or formally, the value $\max_{v \in V(T)} \min_{q \in Q}\{w(v) \cdot d(v, q)\}$ is minimized, where $V(T)$ is the vertex set of T. Note that each center can be in the interior of an edge of T.

Kariv and Hakimi [20] first gave an $O(n^2 \log n)$ time algorithm for the problem. Jeger and Kariv [19] proposed an $O(kn \log n)$ time algorithm. Megiddo and Tamir [25] solved the problem in $O(n \log^2 n \log \log n)$ time, and the running time can be reduced to $O(n \log^2 n)$ by Cole's parametric search [12]. Some progress has been made very recently by Banik et al. [3] for small values of k, where an $O(n \log n + k \log^2 n \log(n/k))$-time algorithm and another $O(n \log n + k^2 \log^2(n/k))$-time algorithm were given.

Since Megiddo and Tamir's work [25], it has been open whether the problem can be solved in $O(n \log n)$ time. In this paper, we settle this three-decade long open problem

© Haitao Wang and Jingru Zhang;
licensed under Creative Commons License CC-BY
34th International Symposium on Computational Geometry (SoCG 2018).
Editors: Bettina Speckmann and Csaba D. Tóth; Article No. 72; pp. 72:1–72:15


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

affirmatively by presenting an $O(n \log n)$-time algorithm. Note that the previous $O(n \log^2 n)$-time algorithm [12, 25] and the first algorithm in [3] both rely on Cole's parametric search, which involves a large constant in the time complexity due to the AKS sorting network [2]. Our algorithm, however, avoids Cole's parametric search.

If each center is required to be located a vertex of T, then we call it the *discrete* case. The previously best-known algorithm for this case runs in $O(n \log^2 n)$ time [26]. Our techniques also solve the discrete case in $O(n \log n)$ time.

Related work. Many variations of the k-center problem have been studied. If $k = 1$, then the problem is solvable in $O(n)$ time [23]. If T is a path, the k-center problem was already solved in $O(n \log n)$ time [9, 12, 25], and Bhattacharya and Shi [4] also gave an algorithm whose running time is linear in n but exponential in k.

For the unweighted case where the vertices of T have the same weight, an $O(n^2 \log n)$-time algorithm was given in [8] for the k-center problem. Later, Megiddo et al. [26] solved the problem in $O(n \log^2 n)$ time, and the algorithm was improved to $O(n \log n)$ time [17]. Finally, Frederickson [16] solved the problem in $O(n)$ time. The above four papers also solve the discrete case and the following problem version in the same running times: All points of T are considered as demand points and the centers are required to be at vertices of T. Further, if all points of T are demand points and centers can be any points of T, Megiddo and Tamir solved the problem in $O(n \log^3 n)$ time [25], and the running time can be reduced to $O(n \log^2 n)$ by applying Cole's parametric search [12].

As related problems, Frederickson [15] presented $O(n)$-time algorithms for the following tree partitioning problems: remove k edges from T such that the maximum (resp., minimum) total weight of all connected subtrees is minimized (resp., maximized).

Finding k centers in a general graph is NP-hard [20]. The geometric version of the problem in the plane is also NP-hard [24], i.e., finding k centers for n demanding points. Some special cases, however, are solvable in polynomial time. For example, if $k = 1$, then the problem can be solved in $O(n)$ time [23], and if $k = 2$, it can be solved in $O(n \log^2 n \log^2 \log n)$ time [7] (also refer to [1] for a faster randomized algorithm). If we require all centers to be on a given line, then the problem of finding k centers can be solved in polynomial time [5, 21, 28]. Recently, problems on uncertain data have been studied extensively and some k-center problem variations on uncertain data were also considered, e.g., [13, 18, 27, 29–31].

Our approach. We discuss our approach for the non-discrete problem, and that for the discrete case is similar (and even simpler). Let λ^* be the optimal objective value, i.e., $\lambda^* = \max_{v \in V(T)} \min_{q \in Q}\{w(v) \cdot d(v, q)\}$ for an optimal solution Q. A *feasibility test* is to determine whether $\lambda \geq \lambda^*$ for a given value λ, and if yes, we call λ a *feasible value.* Given any λ, the feasibility test can be done in $O(n)$ time [20].

Our algorithm follows an algorithmic scheme in [16] for the unweighted case, which is similar to that in [15] for the tree partition problems. However, a big difference is that three schemes were proposed in [15, 16] to gradually solve the problems in $O(n)$ time, while our approach only follows the first scheme and this significantly simplifies the algorithm. One reason the first scheme is sufficient to us is that our algorithm runs in $O(n \log n)$ time, which has a logarithmic factor more than the feasibility test algorithm. In contrast, most efforts of the last two schemes of [15, 16] are to reduce the running time of the algorithms to $O(n)$, which is the same as their corresponding feasibility test algorithms.

More specifically, our algorithm consists of two phases. The first phase will gather information so that each feasibility test can be done faster in sub-linear time. By using the

faster feasibility test algorithm, the second phase computes the optimal objective value λ^*. As in [16], we also use a stem-partition of the tree T. In addition to a matrix searching algorithm [14], we utilize some other techniques, such as 2D sublist LP queries [10] and line arrangements searching [11], etc.

In the following, in Section 2, we review some previous techniques that will be used later in our algorithm. In Section 3, we describe our techniques for dealing with a so-called "stem". We finally solve the k-center problem on T in Section 4. Due to the space limit, many details (including the algorithm for the discrete case) are omitted but can be found in the full paper.

2 Preliminaries

2.1 The feasibility test FTEST0

Given any value λ, the feasibility test is to determine whether λ is feasible, i.e., whether $\lambda \geq \lambda^*$. We say that a vertex v of T is *covered* (under λ) by a center q if $w(v) \cdot d(v, q) \leq \lambda$. Note that λ is feasible if and only if we can place k centers in T such that all vertices are covered. In the following we describe a linear-time feasibility test algorithm, which is essentially the same as the one in [20] although our description is much simpler.

We pick a vertex of T as the root, denoted by γ. For each vertex v, we use $T(v)$ to denote the subtree of T rooted at v. Following a post-order traversal on T, we place centers in a bottom-up and greedy manner. For each vertex v, we maintain two values $sup(v)$ and $dem(v)$, where $sup(v)$ is the distance from v to the closest center that has been placed in $T(v)$, and $dem(v)$ is the maximum distance from v such that if we place a center q within such a distance from v then all uncovered vertices of $T(v)$ can be covered by q. We also maintain a variable *count* to record the number of centers that have been placed so far.

Initially, $count = 0$, and for each vertex v, $sup(v) = \infty$ and $dem(v) = \frac{\lambda}{w(v)}$. Following a post-order traversal on T, suppose vertex v is being visited. For each child u of v, we update $sup(v)$ and $dem(v)$ as follows. If $sup(u) \leq dem(u)$, then we can use the center of $T(u)$ closest to u to cover the uncovered vertices of $T(u)$, and thus we reset $sup(v) = \min\{sup(v), sup(u) + d(u, v)\}$. Note that since u connects v by an edge, $d(v, u)$ is the length of the edge. Otherwise, if $dem(u) < d(u, v)$, then we place a center on the edge $e(u, v)$ at distance $dem(u)$ from u, so we update $count = count + 1$ and $sup(v) = \min\{sup(v), d(u, v) - dem(u)\}$. Otherwise (i.e., $dem(u) \geq d(u, v)$), we update $dem(v) = \min\{dem(v), dem(u) - d(u, v)\}$.

After the root γ is visited, if $sup(\gamma) > dem(\gamma)$, then we place a center at γ and update $count = count + 1$. Finally, λ is feasible if and only if $count \leq k$. The algorithm runs in $O(n)$ time. We use *FTEST0* to refer to the algorithm. To solve the k-center problem, the key is to compute λ^*, after which we can find k centers by applying *FTEST0* with $\lambda = \lambda^*$.

Remark. The algorithm *FTEST0* actually partitions T into at most k disjoint connected subtrees such that the vertices in each subtree is covered by the same center that is located in the subtree. We will make use of this observation later.

2.2 A matrix searching algorithm

We review an algorithm *MSEARCH*, which was proposed in [14] and was widely used, e.g., [15–17]. A matrix is *sorted* if elements in every row and every column are in nonincreasing order. Given a set of sorted matrices, a searching range (λ_1, λ_2) such that λ_2 is feasible and λ_1 is not, and a stopping count c, *MSEARCH* will produce a sequence of values one at a time for feasibility tests, and after each test, some elements in the matrices will be discarded.

Suppose a value λ is produced. If $\lambda \notin (\lambda_1, \lambda_2)$, we do not need to test λ. If λ is feasible, then λ_2 is updated to λ; otherwise, λ_1 is updated to λ. *MSEARCH* will stop once the number of remaining elements in all matrices is at most c. Lemma 1 is proved in [14] and we slightly change the statement to accommodate our need.

▶ **Lemma 1** ([14–17]). *Let $\mathcal{M}$ be a set of N sorted matrices $\{M_1, M_2, \ldots, M_N\}$ such that M_j is of dimension $m_j \times n_j$ with $m_j \leq n_j$, and $\sum_{j=1}^{N} m_j = m$. Let $c \geq 0$. The number of feasibility tests needed by* MSEARCH *to discard all but at most c of the elements is $O(\max\{\log \max_j\{n_j\}, \log(\frac{m}{c+1})\})$, and the total time of* MSEARCH *exclusive of feasibility tests is $O(\kappa \cdot \sum_{j=1}^{N} m_j \log(\frac{2n_j}{m_j}))$, where $O(\kappa)$ is the time for evaluating each matrix element (i.e., the number of matrix elements that need to be evaluated is $O(\sum_{j=1}^{N} m_j \log(\frac{2n_j}{m_j})))$.*

2.3 The 2D sublist LP queries

Let $H = \{h_1, h_2, \ldots, h_m\}$ be a set of m upper half-planes in the plane. Given two indices i and j with $1 \leq i \leq j \leq m$, a *2D sublist LP query* asks for the lowest point in the common intersection of $h_i, h_{i+1}, \ldots, h_j$. The *line-constrained* version of the query is: Given a vertical line l and two indices i and j with $1 \leq i \leq j \leq m$, the query asks for the lowest point on l in the common intersection of $h_i, h_{i+1}, \ldots, h_j$. Lemma 2 was proved in [10] (i.e., Lemma 8 and the discussion after it; the query algorithm for the line-constrained version is used as a procedure in the proof of Lemma 8).

▶ **Lemma 2** ([10]). *We can build a data structure for H in $O(m \log m)$ time such that each 2D sublist LP query can be answered in $O(\log^2 m)$ time, and each line-constrained query can be answered in $O(\log m)$ time.*

2.4 Line arrangement searching

Let L be a set of m lines in the plane. Denote by $\mathcal{A}(L)$ the arrangement of the lines of L, and let $y(v)$ be the y-coordinate of each vertex v of $\mathcal{A}(L)$. Let $v_1(L)$ be the lowest vertex of $\mathcal{A}(L)$ whose y-coordinate is a feasible value, and let $v_2(L)$ be the highest vertex of $\mathcal{A}(L)$ whose y-coordinate is smaller than that of $v_1(L)$. Hence, $y(v_2(L)) < \lambda^* \leq y(v_1(L))$, and $\mathcal{A}(L)$ does not have a vertex v with $y(v_2(L)) < y(v) < y(v_1(L))$. Lemma 3 was proved in [11].

▶ **Lemma 3** ([11]). *Both vertices $v_1(L)$ and $v_2(L)$ can be computed in $O((m+\tau)\log m)$ time, where τ is the time for a feasibility test.*

Remark. Alternatively, we can use Cole's parametric search [12] to compute the two vertices. First, we sort the lines of L by their intersections with the horizontal line $y = \lambda^*$, and this can be done in $O((m + \tau)\log m)$ time by Cole's parametric search [12]. Then, $v_1(L)$ and $v_2(L)$ can be found in additional $O(m)$ time because each of them is an intersection of two adjacent lines in the above sorted order. The line arrangement searching technique, which modified the slope selection algorithms [6, 22], avoids Cole's parametric search [12].

We will often talk about some problems in the plane $\mathbb{R}^2$, and if the context is clear, for any point $p \in \mathbb{R}^2$, we use $x(p)$ and $y(p)$ to denote its x- and y-coordinates, respectively.

3 The algorithms for stems

In this section, we first define *stems*, which are similar in spirit to those proposed in [16] for the unweighted case. Then, we will present two algorithms for solving the k-center problem on a stem, and both techniques will be used later for solving the problem in the tree T.

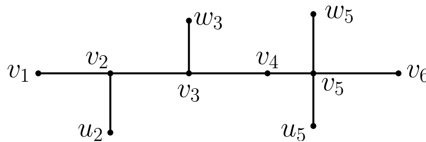


Figure 1 Illustrating a stem.

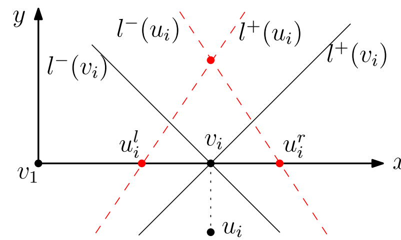


Figure 2 Illustrating the definitions of the lines defined by a backbone vertex v_i and its thorn vertex u_i.

Let $\widehat{P}$ be a path of m vertices, denoted by $v_1, v_2, \ldots, v_m$, sorted from left to right. For each vertex v_i, other than its incident edges in $\widehat{P}$, v_i has at most two additional edges connecting two vertices u_i and w_i that are not in $\widehat{P}$. Either vertex may not exist. Let P denote the union of $\widehat{P}$ and the above additional edges (e.g., see Fig. 1). For any two points p and q on P, we still use $\pi(p, q)$ to denote the unique path between p and q in P, and use $d(p, q)$ to denote the length of the path. With respect to a range (λ_1, λ_2), we call P a *stem* if the following holds: For each $i \in [1, m]$, if u_i exists, then $w(u_i) \cdot d(u_i, v_i) \leq \lambda_1$; if w_i exists, then $w(w_i) \cdot d(w_i, v_i) \geq \lambda_2$.

Following the terminology in [16], we call $e(v_i, u_i)$ a *thorn* and $e(v_i, w_i)$ a *twig*. Each u_i is called a *thorn vertex* and each w_i is called a *twig vertex*. $\widehat{P}$ is called the *backbone* of P, and each vertex of $\widehat{P}$ is called a *backbone vertex*. We define m as the *length* of P. The total number of vertices of P is at most $3m$.

Remark. Our algorithm in Section 4 will produce stems P as defined above, where $\widehat{P}$ is a path in T and all vertices of P are also vertices of T. However, each thorn $e(u_i, v_i)$ may not be an original edge of T, but it corresponds to the path between u_i and v_i in T in the sense that the length of $e(u_i, v_i)$ is equal to the distance between u_i and v_i in T. This is also the case for each twig $e(w_i, v_i)$. Our algorithm in Section 4 will maintain a range (λ_1, λ_2) such that λ_1 is not feasible and λ_2 is feasible, i.e., $\lambda^* \in (\lambda_1, \lambda_2]$. Since any feasibility test will be made to a value $\lambda \in (\lambda_1, \lambda_2)$, the above definitions on thorns and twigs imply the following: For each thorn vertex u_i, we can place a center on the backbone to cover it (under λ), and for each twig vertex w_i, we need to place a center on the edge $e(w_i, v_i) \setminus \{v_i\}$ to cover it.

In the sequel we give two different techniques for solving the k-center problem on the stem P. In fact, in our algorithm for the k-center problem on T in Section 4, we use these techniques to process a stem P, rather than directly solve the k-center problem on P. Let λ^* temporarily refer to the optimal objective value of the k-center problem on P in the rest of this section, and we assume $\lambda^* \in (\lambda_1, \lambda_2]$.

3.1 The first algorithm

This algorithm is motivated by the following easy observation: there exist two vertices v and v' in P such that a center q is located in the path $\pi(v, v')$ and $w(v) \cdot d(q, v) = w(v') \cdot d(q, v') = \lambda^*$.

We assume that all backbone vertices of P are in the x-axis of an xy-coordinate system $\mathbb{R}^2$ where v_1 is at the origin and each v_i has x-coordinate $d(v_1, v_i)$. Each v_i defines two lines $l^+(v_i)$ and $l^-(v_i)$ both containing v_i and with slopes $w(v_i)$ and $-w(v_i)$, respectively (e.g., see Fig. 2). Each thorn u_i also defines two lines $l^+(u_i)$ and $l^-(u_i)$ as follows. Define u_i^l (resp., u_i^r) to be the point in $\mathbb{R}^2$ on the x-axis with x-coordinate $d(v_1, v_i) - d(u_i, v_i)$ (resp., $d(v_1, v_i) + d(u_i, v_i)$). Hence, u_i^l (resp., u_i^r) is to the left (resp., right) of v_i with distance $d(u_i, v_i)$ from v_i. Define $l^+(u_i)$ to be the line through u_i^l with slope $w(u_i)$ and $l^-(u_i)$ to be the

line through u_i^r with slope $-w(u_i)$. Note that $l^+(u_i)$ and $l^-(u_i)$ intersect at the point whose x-coordinate is the same as that of v_i and whose y-coordinate is equal to $w(u_i) \cdot d(u_i, v_i)$. For each twig vertex w_i, we define points w_i^l and w_i^r, and lines $l^+(w_i)$ and $l^-(w_i)$, in the same way as those for u_i.

Consider a point q on the backbone of P to the right side of v_i. It can be verified that the weighted distance $w(v_i) \cdot d(v_i, q)$ from v_i to q is exactly equal to the y-coordinate of the intersection between $l^+(v_i)$ and the vertical line through q. If q is on the left side of v_i, we have a similar observation for $l^-(v_i)$. This is also true for u_i and w_i.

Let L denote the set of the lines in $\mathbb{R}^2$ defined by all vertices of P. Note that $|L| \leq 6m$. Based on the above observation, λ^* is equal to the y-coordinate of a vertex of the line arrangement $\mathcal{A}(L)$ of L. More precisely, λ^* is equal to the y-coordinate of $v_1(L)$, as defined in Section 2. By Lemma 3, we can compute λ^* in $O((m + \tau) \log m)$ time.

3.2 The second algorithm

This algorithm relies on the algorithm *MSEARCH*. We first form a set of sorted matrices.

For each $i \in [1, m]$, we define the two lines $l_i^+(v_i)$ and $l_i^-(v_i)$ in $\mathbb{R}^2$ as above in Section 3.1. If u_i exists, then we also define $l_i^+(u_i)$ and $l_i^-(u_i)$ as before; otherwise, both $l_i^+(u_i)$ and $l_i^-(u_i)$ refer to the x-axis. Let $h_{4(i-1)+j}$, $1 \leq j \leq 4$, denote respectively the four upper half-planes bounded by the above four lines (their index order is arbitrary). In this way, we have a set $H = \{h_1, h_2 \ldots, h_{4m}\}$ of $4m$ upper half-planes.

For any i and j with $1 \leq i \leq j \leq m$, we define $\alpha(i, j)$ as the y-coordinate of the lowest point in the common intersection of the upper half-planes of H from $h_{4(i-1)+1}$ to h_{4j}, i.e., all upper half-planes defined by u_t and v_t for $t \in [i, j]$. Observe that if we use one center to cover all backbone and thorn vertices u_t and v_t for $t \in [i, j]$, then $\alpha(i, j)$ is equal to the optimal objective value of this one-center problem.

We define a matrix M of dimension $m \times m$: For any i and j in $[1, m]$, if $i + j \leq m + 1$, then $M[i, j] = \alpha[i, m + 1 - j]$; otherwise, $M[i, j] = 0$.

For each twig w_i, we define two arrays A_i^r and A_i^l of at most m elements each as follows. Let $h^+(w_i)$ and $h^-(w_i)$ denote respectively the upper half-planes bounded by the lines $l^+(w_i)$ and $l^-(w_i)$ defined in Section 3.1. The array A_i^r is defined on the vertices of P on the right side of v_i, as follows. For each $j \in [1, m - i + 1]$, if we use a single center to cover w_i and all vertices u_t and v_t for $t \in [i, m + 1 - j]$, then $A_i^r[j]$ is defined to be the optimal objective value of this one-center problem, which is equal to the y-coordinate of the lowest point in the common intersection of $h^+(w_i)$ and the upper half-planes of H from $h_{4(i-1)+1}$ to $h_{4(m+1-j)}$. Symmetrically, array A_i^l is defined on the left side of v_i. Specifically, for each $j \in [1, i]$, if we use one center to cover w_i and all vertices u_t and v_t for $t \in [j, i]$, then $A^l[j]$ is defined to be the optimal objective value, which is equal to the y-coordinate of the lowest point in the common intersection of $h^-(w_i)$ and the upper half-planes of H from $h_{4(j-1)+1}$ to h_{4i}.

Let $\mathcal{M}$ be the set of the matrices M and A_i^r and A_i^l for all $1 \leq i \leq m$. The following lemma implies that we can apply *MSEARCH* on $\mathcal{M}$ to compute λ^*.

► **Lemma 4.** *Each matrix of $\mathcal{M}$ is sorted, and λ^* is an element of a matrix in $\mathcal{M}$.*

Proof. Refer to the full paper for the proof that every matrix of $\mathcal{M}$ is sorted. In the following, we show that λ^* must be an element of one of these matrices. We only sketch the proof.

Imagine that we apply algorithm *FTEST0* on $\lambda = \lambda^*$ and the stem P by considering P as a tree with root v_m. Then, *FTEST0* will compute at most k centers in P. *FTEST0* actually partitions P into at most k disjoint connected subtrees such that the vertices in each subtree is covered by the same center that is located in the subtree. Further, there must be a subtree

P_1 that has a center q and two vertices v' and v such that $w(v) \cdot d(v,q) = w(v') \cdot d(v',q) = \lambda^*$. Since P_1 is connected and both v and v' are in P_1, the path $\pi(v,v')$ is also in P_1.

Depending on whether one of v and v' is a twig vertex, there are two cases.

If neither vertex is a twig vertex, then we claim that all thorn vertices connecting to the backbone vertices of $\pi(v,v')$ are covered by the center q (see the full paper for the proof of the claim). If v is a backbone vertex, then let i be its index, i.e., $v = v_i$; otherwise, v is a thorn vertex and let i be the index such that v connects the backbone vertex v_i. Similarly, define j for v'. Without loss of generality, assume $i \leq j$. The above claim implies that λ^* is equal to the y-coordinate of the lowest point in the common intersection of all upper half-planes defined by the backbone vertices v_t and thorn vertices u_t for all $t \in [i,j]$, and thus, $\lambda^* = \alpha(i,j)$, which is equal to $M[i, m+1-j]$. Therefore, λ^* is in the matrix M.

Next, we consider the case where at least one of v and v' is a twig vertex. For each twig vertex w_i of P, by definition, $w(w_i) \cdot d(w_i, v_i) \geq \lambda_2$, and since $\lambda^* \leq \lambda_2$, the twig $e(w_i, v_i)$ must contain a center. Because both v and v' are covered by q, only one of them is a twig vertex. Without loss of generality, we assume that v is a twig vertex, say, w_i. If v' is a backbone vertex, then let j be its index; otherwise, v' is a thorn vertex and let j be the index such that v' connects the backbone vertex v_j. Without loss of generality, we assume $i \leq j$.

By the same argument as the above, all thorn vertices u_t with $t \in [i,j]$ are covered by q. This implies that λ^* is the y-coordinate of the lowest point in the common intersection of $h^+(w_i)$ and all upper half-planes defined by the backbone vertices v_t and thorn vertices u_t for all $t \in [i,j]$. Thus, $\lambda^* = A_i^r[m+1-j]$. Therefore, λ^* is in the array A_i^r. ◀

Note that $\mathcal{M}$ consists of a matrix M of dimension $m \times m$ and $2m$ arrays of lengths at most m. With the help of the 2D sublist LP query data structure in Lemma 2, the following lemma shows that the matrices of $\mathcal{M}$ can be implicitly formed in $O(m \log m)$ time.

▶ **Lemma 5.** *With $O(m \log m)$ time preprocessing, each matrix element of $\mathcal{M}$ can be evaluated in $O(\log^2 m)$ time.*

Proof. We build a data structure of Lemma 2 on the upper half-planes of H in $O(m \log m)$ time. Then, each element of M can be obtained in $O(\log^2 m)$ time by a 2D sublist LP query.

Now consider an array A_i^l. Given any index j, to compute $A_i^l[j]$, recall that $A_i^l[j]$ is equal to the y-coordinate of the lowest point p^* of the common intersection of the upper half-plane $h^+(w_i)$ and those in H', where H' is the set of the upper half-planes of H from $h_{4(j-1)+1}$ to h_{4i}. The lowest point p' of the common intersection of the upper half-planes of H' can be computed in $O(\log^2 m)$ time by a 2D sublist LP query with query indices $4(j-1)+1$ and $4i$. Computing p^* can also be done in $O(\log^2 m)$ time by slightly modifying the query algorithm for computing p'. We briefly discuss it below and the interested reader should refer to [10] for details (the proof of Lemma 8 and the discussion after the lemma).

The query algorithm for computing p' is similar in spirit to the linear-time algorithm for the 2D linear programming problem in [23]. It is a binary search algorithm. In each iteration, the algorithm computes the highest intersection p'' between a vertical line l and the bounding lines of the half-planes of H', and based on the local information at the intersection, the algorithm will determine which side to proceed for the search. For computing p^*, we need to incorporate the additional half-plane $h^+(w_i)$. To this end, in each iteration of the binary search, after we compute the highest intersection p'', we compare it with the intersection of l and the bounding line of $h^+(w_i)$ and update the highest intersection if needed. This costs only constant extra time for each iteration. Therefore, computing p^* takes $O(\log^2 m)$ time.

Computing the elements of arrays A_i^r can be done similarly. The lemma thus follows. ◀

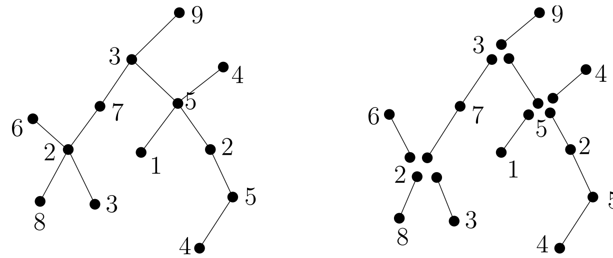


Figure 3 Left: the tree T where the numbers are the vertex weights. Right: the path partition of T.

By applying algorithm *MSEARCH* on $\mathcal{M}$ with stopping count $c = 0$ and $\kappa = O(\log^2 m)$, according to Lemma 1, *MSEARCH* produces $O(\log m)$ values for feasibility tests, and the total time exclusive of feasibility tests is $O(m \log^3 m)$ because we need to evaluate $O(m \log m)$ matrix elements of $\mathcal{M}$. Hence, the total time for computing λ^* is $O(m \log^3 m + \tau \cdot \log m)$.

Remark. Clearly, the first algorithm is better than the second one. However, later when we use the techniques of the second algorithm, m is often bounded by $O(\log^2 n)$ and thus $\log^3 m = O(\log n)$. In fact, we use the techniques of the second algorithm mainly because we need to set the stopping count c to some non-zero value.

4 Solving the k-center problem on T

In this section, we present our algorithm for solving the k-center problem on T. We will focus on computing the optimal objective value λ^*. Frederickson [15] proposed a *path-partition* of T, which is a partition of the edges of T into paths where a vertex v is an endpoint of a path if and only if the degree of v in T is not equal to 2 (e.g., see Fig. 3). A path in a partition-partition of T is called a *leaf-path* if it contains a leaf of T.

As in [16], we generalize the path-partition to stem-partition as follows. During the course of our algorithm, a range $(\lambda_1, \lambda_2]$ that contains λ^* will be maintained and T will be modified by removing some edges and adding some thorns and twigs. At any point in our algorithm, let T' be T with all thorns and twigs removed. A *stem* of T is a path in the path-partition of T', along with all thorns and twigs that connect to vertices in the path. A *stem-partition* of T is to partition T into stems according to a path-partition of T'. A stem in a stem-partition of T is called a *leaf-stem* if it contains a leaf of T that is a backbone vertex of the stem.

Our algorithm follows the first algorithmic scheme in [16]. There are two main phases: Phase 1 and Phase 2. Let $r = \log^2 n$. Phase 1 gathers information so that the feasibility test can be made in sublinear $O(\frac{n}{r} \log^3 r)$ time. Phase 2 computes λ^* by using the faster feasibility test. If T has more than $2n/r$ leaves, then there is an additional phase, called Phase 0, which reduces the problem to a tree with at most $2n/r$ leaves. In the following, we consider the general case where T has more than $2n/r$ leaves.

4.1 The preprocessing and computing the vertex ranks

We first perform some preprocessing. Recall that γ is the root of T. We compute the distances $d(v, \gamma)$ for all vertices v in $O(n)$ time. Then, if u is an ancestor of v, $d(u, v) = d(\gamma, v) - d(\gamma, u)$, which can be computed in $O(1)$ time. In the following, whenever we need to compute a distance $d(u, v)$, it is always the case that one of u and v is an ancestor of the other, and thus $d(u, v)$ can be obtained in $O(1)$ time.

Next, we compute a "rank" $rank(v)$ for each vertex v of T. These ranks will facilitate our algorithm later. For each vertex v, we define a point $p(v)$ on the x-axis with x-coordinate equal to $d(\gamma, v)$ in an xy-coordinate system $\mathbb{R}^2$, and define $l(v)$ as the line through $p(v)$ with slope equal to $-w(v)$. Let L be the set of these n lines. Consider the line arrangement $\mathcal{A}(L)$ of L. Let $v_1(L)$ and $v_2(L)$ be the vertices as defined in Section 2. By Lemma 3, both vertices can be computed in $O(n \log n)$ time. Let l be a horizontal line strictly between $v_1(L)$ and $v_2(L)$. We sort all lines of L by their intersections with l from left to right, and for each vertex v, we define $rank(v) = i$ if there are $i-1$ lines before $l(v)$ in the above order. By the definitions of $v_1(L)$ and $v_2(L)$, the above order of L is also an order of L sorted by their intersections with the horizontal line $y = \lambda^*$.

4.2 Phase 0

Recall that T has more than $2n/r$ leaves. In this section, we reduce the problem to the problem of placing centers in a tree with at most $2n/r$ leaves. Our algorithm will maintain a range $(\lambda_1, \lambda_2]$ that contains λ^*. Initially, $\lambda_1 = y(v_2(L))$, the y-coordinate of $v_2(L)$, which is already computed in the preprocessing, and $\lambda_2 = y(v_1(L))$. We form a stem-partition of T, which is actually a path-partition since there are no thorns and twigs initially, and this can be done in $O(n)$ time.

Recall that $r = \log^2 n$. While there are more than $2n/r$ leaves in T, we do the following.

Recall that the length of a stem is defined as the number of backbone vertices. Let S be the set of all leaf-stems of T whose lengths are at most r. Let n' be the number of all backbone vertices on the leaf-stems of S. For each leaf-stem of S, we form matrices by Lemma 5. Let $\mathcal{M}$ denote the collection of matrices for all leaf-stems of S. We call *MSEARCH* on $\mathcal{M}$, with stopping count $c = n'/(2r)$, by using the feasibility test algorithm *FTEST0*. After *MSEARCH* stops, we have an updated range (λ_1, λ_2) and matrix elements of $\mathcal{M}$ in (λ_1, λ_2) are called *active* values. Since $c = n'/(2r)$, at most $n'/(2r)$ active values of $\mathcal{M}$ remain, and thus at most $n'/(2r)$ leaf-stems of S have active values.

For each leaf-stem $P \in S$ without active values, we perform the following *post-processing procedure*. The backbone vertex of P closest to the root is called the *top vertex*. We place centers on P, subtract their number from k, and replace P by either a thorn or a twig connected to the top vertex (P is thus removed from T except the top vertex), such that solving the k-center problem on the modified T also solves the problem on the original T. The post-processing procedure can be implemented in $O(m)$ time, where m is the length of P. The details are given below.

The post-processing procedure on P. Let z be the top vertex of P. We run *FTEST0* on P with z as the root and $\lambda = \lambda'$ that is an arbitrary value in (λ_1, λ_2). After z is finally processed, depending on whether $sup(z) \leq dem(z)$, we do the following.

If $sup(z) \leq dem(z)$, then let q be the last center that has been placed. In this case, all vertices of P are covered and z is covered by q. According to algorithm *FTEST0* and as discussed in the proof of Lemma 4, q covers a connected subtree of vertices, and let $V(q)$ denote the set of these vertices excluding z. Note that $V(q)$ can be easily identified during *FTEST0*. Let k' be the number of centers excluding q that have been placed on P. Since $\lambda' \in (\lambda_1, \lambda_2)$ and the matrices formed based on P do not have any active values, we have the following *key observation:* if we run *FTEST0* with any $\lambda \in (\lambda_1, \lambda_2)$, the algorithm will also cover all vertices of $P \setminus (V(q) \cup \{z\})$ with k' centers and cover vertices of $V(q) \cup \{z\}$ with one center. Indeed, this is true because the way we form matrices for P is consistent with *FTEST0*, as discussed in the proof of Lemma 4. In this case, we replace P by attaching a

twig $e(u,z)$ to z with length equal to $d(u,z)$, where u is a vertex of $V(q)$ with the following property: For any $\lambda \in (\lambda_1, \lambda_2)$, if we place a center q' on the path $\pi(u,z)$ at distance $\lambda/w(u)$ from u, then q' will cover all vertices of $V(q)$ under λ, i.e., u "dominates" all other vertices of $V(q)$ and thus it is sufficient to keep u (since λ_2 is feasible, any subsequent feasibility test in the algorithm will use $\lambda \in (\lambda_1, \lambda_2)$). The following lemma shows that u is the vertex of $V(q)$ with the largest rank. The proof of the lemma is omitted.

▶ **Lemma 6.** *Let u be the vertex of $V(q)$ with the largest rank. For any $\lambda \in (\lambda_1, \lambda_2)$, the following holds.*
1. $\frac{\lambda}{w(u)} \leq d(u,z)$.
2. *If q' is the point on the path $\pi(u,z)$ with distance $\frac{\lambda}{w(u)}$ from u, then q' covers all vertices of $V(q)$ under λ, i.e., $w(v) \cdot d(v,q') \leq \lambda$ for all $v \in V(q)$.*

Due to the preprocessing in Section 4.1, we can find u from $V(q)$ in $O(m)$ time. This finishes our post-processing procedure for the case $sup(z) \leq dem(z)$. Since $\frac{\lambda}{w(u)} \leq d(u,z)$ for any $\lambda \in (\lambda_1, \lambda_2)$, we have $w(u) \cdot d(u,z) \geq \lambda_2$, and thus, $e(u,z)$ is indeed a twig.

Next, we consider the other case $sup(z) > dem(z)$. In this case, P has some vertices other than z that are not covered yet, and we would need to place a center at z to cover them. Let V be the set of all uncovered vertices other than z, and V can be identified during *FTEST0*. In this case, we replace P by attaching a thorn $e(u,z)$ to z with length equal to $d(u,z)$, where u is a vertex of V with the following property: For any $\lambda \in (\lambda_1, \lambda_2)$, if there is a center q outside P covering u *through* z (by "through", we mean that $\pi(q,u)$ contains z) under distance λ, then q also covers all other vertices of V (intuitively u "dominates" all other vertices of V). Since later we will place centers outside P to cover the vertices of V through z under some $\lambda \in (\lambda_1, \lambda_2)$, it is sufficient to maintain u. The following lemma shows that u is the vertex of V with the largest rank. The proof is omitted.

▶ **Lemma 7.** *Let u be the vertex of V with the largest rank. Then, for any center q outside P that covers u through z under any $\lambda \in (\lambda_1, \lambda_2)$, q also covers all other vertices of V.*

Since a center at z would cover u, it holds that $w(u) \cdot d(u,z) \leq \lambda$ for any $\lambda \in (\lambda_1, \lambda_2)$, which implies that $w(u) \cdot d(u,z) \leq \lambda_1$. Thus, $e(u,z)$ is indeed a thorn.

The above replaces P by attaching to z either a thorn or a twig. We perform the following additional processing.

Suppose z is attached by a thorn $e(z,u)$. If z already has another thorn $e(z,u')$, then we discard one of u' and u whose rank is smaller, because any center that covers the remaining vertex will cover the discarded one as well. This makes sure that z has at most one thorn.

Suppose z is attached by a twig $e(z,u)$. If z already has another twig $e(z,u')$, then we can discard one of u and u' whose rank is *larger* (and subtract 1 from k). The reason is the following. Without loss of generality, assume $rank(u) < rank(u')$. Since both $e(z,u)$ and $e(z,u')$ are twigs, if we apply *FTEST0* on any $\lambda \in (\lambda_1, \lambda_2)$, then the algorithm will place a center q on $e(z,u)$ with distance $\lambda/w(u)$ from u and place a center q' on $e(z,u')$ with distance $\lambda/w(u')$ from u'. As $rank(u) < rank(u')$, we have Lemma 8.

▶ **Lemma 8.** $d(q,z) \leq d(q',z)$.

Lemma 8 tells that any vertex covered by q' in the subsequent algorithm will also be covered by q. Thus, it suffices to maintain the twig $e(z,u)$. Since we need to place a center at $e(z,u')$, we subtract 1 from k after removing $e(z,u')$. Hence, z has at most one twig.

This finishes the post-processing procedure for P. Due to the preprocessing in Section 4.1, the running time of the procedure is $O(m)$.

Let T be the modified tree after the post-processing on each stem P without active values. If T still has more than $2n/r$ leaves, then we repeat the above. The algorithm stops once T has at most $2n/r$ leaves. This finishes Phase 0. The following lemma gives the time analysis.

▶ **Lemma 9.** *Phase 0 runs in $O(n(\log\log n)^3)$ time.*

Proof. We first argue that the number of iterations of the while loop is $O(\log r)$.

We consider an iteration of the while loop. Suppose the number of leaf-stems in T, denoted by m, is at least $2n/r$. Then, at most n/r leaf-stems are of length larger than r. Hence, at least half of the leaf-stems are of length at most r. Thus, $|S| \geq m/2$. Recall that n' is the total number of backbone vertices in all leaf-stems of S. Because at most $n'/(2r)$ leaf-stems have active values after *MSEARCH*, at least $|S| - n'/(2r) \geq m/2 - n'/(2r) \geq m/2 - n/(2r) \geq m/2 - m/4 = m/4$ leaf-stems will be removed. Note that removing two such leaf-stems may make an interior vertex become a new leaf in the modified tree. Hence, the tree resulting at the end of each iteration will have at most 7/8 of the leaf-stems of the tree at the beginning of the iteration. Therefore, the number of iterations of the while loop needed to reduce the number of leaf-stems to at most $2n/r$ is $O(\log r)$.

We proceed to analyze the running time of Phase 0. In each iteration of the while loop, we call *MSEARCH* on the matrices for all leaf-stems of S. Since the length of each stem P of S is at most r, there are $O(r)$ matrices formed for P. We perform the preprocessing of Lemma 5 on the matrices, so that each matrix element can be evaluated in $O(\log^2 r)$ time. The total time of the preprocessing on stems of S is $O(n' \log r)$. Since $\mathcal{M}$ has $O(n')$ matrices and the stopping account c is $n'/(2r)$, each call to *MSEARCH* produces $O(\log r)$ values for feasibility tests in $O(n' \log^3 r)$ time (i.e., $O(n' \log r)$ matrix elements will be evaluated). For each leaf-stem without active values, the post-processing time for it is $O(r)$. Hence, the total post-processing time in each iteration is $O(n')$.

Since there are $O(\log r)$ iterations, the total number of feasibility tests is $O(\log^2 r)$, and thus the overall time for all feasibility tests in Phase 0 is $O(n \log^2 r)$. On the other hand, after each iteration, at most $n'/(2r)$ leaf-stems of S have active values and other leaf-stems of S will be deleted. Since the length of each leaf-stem of S is at most r, the leaf-stems with active values have at most $n'/2$ backbone vertices, and thus at least $n'/2$ backbone vertices will be deleted in each iteration. Therefore, the total sum of such n' in all iterations is $O(n)$. Hence, the total time for the preprocessing of Lemma 5 is $O(n \log r)$, the total time for *MSEARCH* is $O(n \log^3 r)$, and the total post-processing time for leaf-stems without active values is $O(n)$.

In summary, the overall time of Phase 0 (excluding the preprocessing in Section 4.1) is $O(n \log^3 r)$, which is $O(n(\log\log n)^3)$ since $r = \log^2 n$. ◀

4.3 Phase 1

We assume that the tree T now has at most $2n/r$ leaves and we want to place k centers in T to cover all vertices. Note that T may have some thorns and twigs. The main purpose of this phase is to gather information so that each feasibility test can be done in sublinear time, and specifically, $O(n/r \log^3 r)$ time. Recall that we have a range $(\lambda_1, \lambda_2]$ that contains λ^*.

We first form a stem-partition for T. Then, we further partition the stems into substems, each of length at most r, such that the lowest backbone vertex v in a substem is the highest backbone vertex in the next lower substem (if v has a thorn or/and a twig, then they are included in the upper substem). So this results in a partition of edges. Let S be the set of all substems. Let T_c be the tree in which each node represents a substem of S and node μ in T_c is the parent of node ν if the highest backbone vertex of the substem for ν is the lowest

backbone vertex of the substem for μ, and we call T_c the *stem tree*. As in [15, 16], since T has at most $2n/r$ leaves, $|S| = O(n/r)$ and the number of nodes of T_c is $O(n/r)$.

For each substem $P \in S$, we compute the set L_P of lines as in Section 3.1. Let L be the set of all the lines for all substems of S. We define the lines of L in the same xy-coordinate system $\mathbb{R}^2$. Clearly, $|L| = O(n)$. Consider the line arrangement $\mathcal{A}(L)$. Define vertices $v_1(L)$ and $v_2(L)$ of $\mathcal{A}(L)$ as in Section 2. With Lemma 3 and *FTEST0*, both vertices can be computed in $O(n \log n)$ time. We update $\lambda_1 = \max\{\lambda_1, y(v_2(L))\}$ and $\lambda_2 = \min\{\lambda_2, y(v_1(L))\}$. Hence, we still have $\lambda^* \in (\lambda_1, \lambda_2]$. We again call the values in (λ_1, λ_2) *active* values.

For each substem $P \in S$, observe that each element of the matrices formed based on P in Section 3.2 is equal to the y-coordinate of the intersection of two lines of L_P, and thus is equal to the y-coordinate of a vertex of $\mathcal{A}(L)$. By the definitions of $v_1(L)$ and $v_2(L)$, no matrix element of P is active.

In the future algorithm, we will only need to test feasibilities for values $\lambda \in (\lambda_1, \lambda_2)$. We compute a data structure on each substem P of S, so that it will help make the feasibility test faster. We have the following lemma and use *FTEST1* to denote the feasibility test algorithm in the lemma. The lemma proof is omitted.

▶ **Lemma 10.** *After $O(n \log n)$ time preprocessing, each feasibility test can be done in $O(n/r \cdot \log^3 r)$ time.*

4.4 Phase 2

In this phase, we will finally compute the optimal objective value λ^*, using the faster feasibility test *FTEST1*. Recall that we have computed a range $(\lambda_1, \lambda_2]$ that contains λ^* after Phase 1.

We first form a stem-partition of T. While there is more than one leaf-stem, we do the following. Let S be the set of all leaf-stems. For each stem $P \in S$, we compute the set of lines as in Section 3.1, and let L be the set of the lines for all stems of S. With Lemma 3 and *FTEST1*, we compute the two vertices $v_1(L)$ and $v_2(L)$ of the arrangement $\mathcal{A}(L)$ as defined in Section 2. We update $\lambda_1 = \max\{\lambda_1, y(v_2(L))\}$ and $\lambda_2 = \min\{\lambda_2, y(v_1(L))\}$. As discussed in Phase 1, each stem P of S does not have any active values (in the matrices defined by P). Next, for each stem P of S, we perform the post-processing procedure as in Section 4.2, i.e., place centers on P, subtract their number from k, and replace P by attaching a twig or a thorn to its top vertex. Let T be the modified tree.

After the while loop, T is a single stem. Then, we apply above algorithm on the only stem T, and the obtained value λ_2 is λ^*. The running time of Phase 2 is bounded by $O(n \log n)$, which is proved in the following theorem.

▶ **Theorem 11.** *The k-center problem on T can be solved in $O(n \log n)$ time.*

Proof. As discussed before, Phases 0 and 1 run in $O(n \log n)$ time. We focus on Phase 2.

First of all, as in [16], the number of iterations of the while loop is $O(\log n)$ because the number of leaf-stems is halved after each iteration. In each iteration, let n' denote the total number of backbone vertices of all leaf-stems in S. Hence, $|L| = O(n')$. Thus, the call to Lemma 3 with *FTEST1* takes $O((n' + n/r \cdot \log^3 r) \log n')$ time. The total time of the post-processing procedure for all leaf-stems of S is $O(n')$. Since all leaf-stems of S will be removed in the iteration, the total sum of all such n' is $O(n)$ in Phase 2. Therefore, the total time of the algorithm in Lemma 3 in Phase 2 is $O(n \log n + n/r \cdot \log^3 r \log^2 n)$, which is $O(n \log n)$ since $r = \log^2 n$. Also, the overall time for the post-processing procedure in Phase 2 is $O(n)$. Therefore, the total time of Phase 2 is $O(n \log n)$. ◀

The pseudocode in Algorithm 1 summarizes the overall algorithm.

Algorithm 1: The k-center algorithm.

Input: A tree T and an integer k
Output: The optimal objective values λ^* and k centers in T

1 Perform the preprocessing in Section 4.1 and compute the "ranks" for all vertices of T;
/* Phase 0 */
2 $r \leftarrow \log^2 n$;
3 Form a stem-partition of T;
4 **while do**
5 there are more than $2n/r$ leaves in T
6 Let S be the set of all leaf-stems of lengths at most r;
7 Form the set $\mathcal{M}$ of matrices for all leaf-stems of S by Lemma 5;
8 Let n' be the total number of all backbone vertices of the leaf-stems of S;
9 Call *MSEARCH* on $\mathcal{M}$ with stopping count $c = n'/(2r)$, using *FTEST0*;
10 **for do**
11 each leaf-stem P of S with no active values
12 Perform the post-processing on P, i.e., place centers on P, subtract their number from k, replace P by a thorn or a twig, and modify the stem-partition of T;
/* Phase 1 */
13 For a stem-partition of T, and for each stem, partition it into substems of lengths at most r;
14 Let S be the set of all substems, and form the stem-tree T_c;
15 Compute the set L of lines for all stems of S in the way discussed in Section 3.1;
16 Compute the two vertices $v_1(L)$ and $v_2(L)$ of $\mathcal{A}(L)$ by Lemma 3 and *FTEST0*, and update λ_1 and λ_2;
17 **for do**
18 each substem P of S
19 Compute the data structure for the faster feasibility test *FTEST1*;
/* Phase 2 */
20 Form a stem-partition of T;
21 **while do**
22 there is more than one leaf-stem in T
23 Compute the set L of lines for all stems of S in the way discussed in Section 3.1;
24 Compute the two vertices $v_1(L)$ and $v_2(L)$ of $\mathcal{A}(L)$ by Lemma 3 and *FTEST1*, and update λ_1 and λ_2;
25 **for do**
26 each leaf-stem of S
27 Perform the post-processing on P, i.e., place centers on P, subtract their number from k, replace P by a thorn or a twig, and modify the stem-partition of T;
28 Compute the set L of the lines for the only leaf-stem T;
29 Compute the two vertices $v_1(L)$ and $v_2(L)$ of $\mathcal{A}(L)$ by Lemma 3 and *FTEST1*, and update λ_1 and λ_2;
30 $\lambda^* = \lambda_2$;
31 Apply *FTEST0* on $\lambda = \lambda^*$ to find k centers in the original tree T;

References

1. P.K. Agarwal and J.M. Phillips. An efficient algorithm for 2D Euclidean 2-center with outliers. In *Proceedings of the 16th Annual European Conference on Algorithms(ESA)*, pages 64–75, 2008.
2. M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1983.
3. A. Banik, B. Bhattacharya, S. Das, T. Kameda, and Z. Song. The p-center problem in tree networks revisited. In *Proc. of the 15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 6:1–6:15, 2016.
4. B. Bhattacharya and Q. Shi. Optimal algorithms for the weighted p-center problems on the real line for small p. In *Proc. of the 10th International Workshop on Algorithms and Data Structures*, pages 529–540, 2007.
5. P. Brass, C. Knauer, H.-S. Na, C.-S. Shin, and A. Vigneron. The aligned k-center problem. *International Journal of Computational Geometry and Applications*, 21:157–178, 2011.
6. H Brönnimann and B. Chazelle. Optimal slope selection via cuttings. *Computational Geometry: Theory and Applications*, 10(1):23–29, 1998.
7. T.M. Chan. More planar two-center algorithms. *Computational Geometry: Theory and Applications*, 13:189–198, 1999.
8. R. Chandrasekaran and A. Tamir. Polynomially bounded algorithms for locating p-centers on a tree. *Mathematical Programming*, 22(1):304–315, 1982.
9. D.Z. Chen, J. Li, and H. Wang. Efficient algorithms for the one-dimensional k-center problem. *Theoretical Computer Science*, 592:135–142, 2015.
10. D.Z. Chen and H. Wang. Approximating points by a piecewise linear function. *Algorithmica*, 88:682–713, 2013.
11. D.Z. Chen and H. Wang. A note on searching line arrangements and applications. *Information Processing Letters*, 113:518–521, 2013.
12. R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *Journal of the ACM*, 34(1):200–208, 1987.
13. G. Cormode and A. McGregor. Approximation algorithms for clustering uncertain data. In *Proc. of the 27t Symposium on Principles of Database Systems (PODS)*, pages 191–200, 2008.
14. G. Frederickson and D. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984.
15. G.N. Frederickson. Optimal algorithms for tree partitioning. In *Proc. of the 2nd Annual ACM-SIAM Symposium of Discrete Algorithms (SODA)*, pages 168–177, 1991.
16. G.N. Frederickson. Parametric search and locating supply centers in trees. In *Proc. of the 2nd International Workshop on Algorithms and Data Structures (WADS)*, pages 299–319, 1991.
17. G.N. Frederickson and D.B. Johnson. Finding kth paths and p-centers by generating and searching good data structures. *Journal of Algorithms*, 4(1):61–80, 1983.
18. L. Huang and J. Li. Stochastic k-center and j-flat-center problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 110–129, 2017.
19. M. Jeger and O. Kariv. Algorithms for finding P-centers on a weighted tree (for relatively small P). *Networks*, 15(3):381–389, 1985.
20. O. Kariv and S.L. Hakimi. An algorithmic approach to network location problems. I: The p-centers. *SIAM Journal on Applied Mathematics*, 37(3):513–538, 1979.
21. A. Karmakar, S. Das, S.C. Nandy, and B.K. Bhattacharya. Some variations on constrained minimum enclosing circle problem. *Journal of Combinatorial Optimization*, 25(2):176–190, 2013.

22 M. Katz and M. Sharir. Optimal slope selection via expanders. *Information Processing Letters*, 47(3):115–122, 1993.
23 N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.
24 N. Megiddo and K.J. Supowit. On the complexity of some common geometric location problems. *SIAM Journal on Comuting*, 13:182–196, 1984.
25 N. Megiddo and A. Tamir. New results on the complexity of p-centre problems. *SIAM Journal on Computing*, 12(4):751–758, 1983.
26 N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $O(n \log^2 n)$ algorithm for the k-th longest path in a tree with applications to location problems. *SIAM Journal on Computing*, 10:328–337, 1981.
27 H. Wang and J. Zhang. One-dimensional k-center on uncertain data. *Theoretical Computer Science*, 602:114–124, 2015.
28 H. Wang and J. Zhang. Line-constrained k-median, k-means, and k-center problems in the plane. *International Journal of Computational Geometry and Applications*, 26:185–210, 2016.
29 H. Wang and J. Zhang. A note on computing the center of uncertain data on the real line. *Operations Research Letters*, 44:370–373, 2016.
30 H. Wang and J. Zhang. Computing the center of uncertain points on tree networks. *Algorithmica*, 609:32–48, 2017.
31 H. Wang and J. Zhang. Covering uncertain points in a tree. In *Proc. of the 15th Algorithms and Data Structures Symposium (WADS)*, pages 557–568, 2017.