CARDINALITY CONSTRAINED FACILITY LOCATION PROBLEMS IN TREES

by

Robert Radu Benkoczi

B.A.Sc., "Politehnica" University of Timişoara, Romania, 1995

M.A.Sc., "Politehnica" University of Timişoara, Romania, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY

May 2004

APPROVAL

Name: Robert Radu Benkoczi

Degree: Doctor of Philosophy

Title of thesis: Cardinality constrained facility location problems in trees

Examining Committee: Dr. Ramesh Krishnamurti
Chair

Dr. Binay Bhattacharya, Senior Supervisor

Dr. Pavol Hell, Supervisor

Dr. Tom Shermer, SFU Examiner

Dr. Arie Tamir, External Examiner
Department of Operations Research
School of Mathematical Sciences
Tel-Aviv University,
Ramat Aviv, Tel-Aviv 69978, Israel

Date Approved: _____

Abstract

Operations Research is the application of scientific methods, especially mathematical and statistical ones, to problems of making decisions. From the huge variety of real life applications, this thesis focuses on a particular class of problems for which the placement of certain resources is in question. These tasks are referred collectively as facility location problems. This dissertation is about algorithms to solve a fundamental problem in facility location, the k-median problem.

The mathematical object used here in modeling the resources and their interactions with the environment is a tree. Many other formulations are used in practice with the k-median problem, but the case of trees is special because, (i) the formulation is very simple, (ii) problems can be solved efficiently, (iii) efficient algorithms for problems in trees can be used to derive approximate solutions for general networks (Tamir [102]), (iv) and efficient algorithms for k-median problems in trees could lead to specific k-median algorithms for classes of graphs less studied, such as the graphs with bounded tree-width. Using simple techniques from computational geometry, we give the first k-median algorithm sub-quadratic in the size of the tree when k is fixed, for arbitrary trees.

In the introduction, we give an overview of the main results known about the k-median problem in general. The main ideas behind our approach are also illustrated. In Chapter 2 we present a decomposition of trees that is central to our methods. In Chapter 3 we describe our approach for solving the k-median problem in trees and we give simplified algorithms for three particular cases, the 3-median problem, the k-median problem in directed trees, and the k-median problem in balanced binary trees. The following two chapters discuss two generalizations of the k-median problem, the k-median problem with positive and negative weights and the collection depots problem.

For Carmen who waited so long

"When would you use it? pray, sir, tell me that."

—*The Two Gentlemen of Verona,* W. Shakespeare

Acknowledgments

This thesis wouldn't have existed if it weren't for the help and support of those around me. I would like to thank first my family and most of all, my beautiful, kind, and wise Carmen who has helped my in every imaginable way. Only I know how vital is her love, her friendship, her unmistakable laughter.

I am indebted to Sara and Peter, our special friends who welcomed me in their house for such a long time. They have touched the lives of many international students with their kindness and care, and they definitely have touched ours. To Moonmoon who has always fed us good food and has surrounded us with warmth and affection. To Dada who believed that I can write a thesis in one month and a half even when I doubted that.

I am thankful for David's friendship, David, the big one from Québec. For Ben's infinite bag of linux and emacs tips, for Riz's jokes, for Qiaosheng's ideas, for Snezana. For all my friends that I played squash with, I went to the movies with, I ate lunch and dinner with. For Jenifer who proof-read my thesis, for Jane and Ian who always encouraged me.

I must thank Professor Tom Shermer from whom I took the Graph Drawing course in my first year at SFU and from whom I learned how to read a paper effectively; for his guidance and suggestions in my thesis proposal and defence. Professor Pavol Hell for his wonderful graduate course on perfect graphs. Professor Arie Tamir who came for my defence all the way from Israel at a time when he was most busy with teaching and exams; for his ideas, observations, and corrections that have improved the quality of this thesis and opened numerous directions for future research.

Finally, thanks are due to my advisor, Professor Binay Bhattacharya without whom I would not be where I am today. Through his guidance and later on through his friendship, he made the years I spent in graduate school be some of my best ones. Thank you all, for adding to my strength and confidence as I continue on the road ahead.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

To organize an economic process so as to maximize efficiency is a fundamental problem that industrial engineers and economists face every day across the world. Many different types of problems exist depending on the activities being optimized. There are the so called *facility location* problems where the goal is to compute an optimal placement for certain objects, production plants or other facilities, relative to the position of a set of existing objects, warehouses or clients of some sort, with which the former must interact. The interaction comes at a price which depends on the distance between objects. Other problems seek an optimal scheduling of resources given a set of constraints like in crew rostering for airline companies. A large number of optimization problems arise in inventory and production management, quality assurance, *etc.*, or even in activities indirectly linked to production such as optimal routing of data in computer networks, communication protocols, and so on.

The methods used in solving all these problems are diverse, spanning different fields from computer science, applied mathematics, biology, and physics. In this thesis we focus on a specific facility location problem called the k-median problem in trees. We propose a set of simple techniques inspired from computational geometry which are quite powerful in the setting of optimization problems in trees. For the k-median problem, we have designed the first algorithm in almost ten years which is different from the usual dynamic programming approach by Kariv and Hakimi [67] and Tamir [99]. The 1996 paper of Tamir [99] considers a more general k-median objective function which accommodates set-up costs, and proves a tighter bound on the running time, but the algorithm is essentially similar to that of Kariv and Hakimi. Results similar to ours were described in the Ph.D. thesis of Rahul Shah [92], for several optimisation problems related to k-median. The problems solved by Shah use a

more general cost function than our k-median problem, but they are not constrained by the cardinality of the set of facilities as in our case. In general, cardinality constraints are more difficult to handle.

We argue that our techniques can be used on other optimization problems with success. As example, we have studied two generalizations of the k-median problem that were recently proposed to the operational research community, the mixed obnoxious facility location problem and the collection depots problem. In both cases, we obtain algorithms with improved running time.

In the following sections, we will discuss the importance of the results obtained and the ideas that have motivated our present work. We will start by defining the k-median problem and the two generalizations studied, then we will introduce part of the notation used throughout the thesis. We will briefly review some of the properties of the k-median problem that illustrate its hardness, and we will conclude, in Section 1.4, with an overview of the techniques used to obtain our results. Among these, an important feature is preprocessing the input tree into a data structure called the *spine decomposition* which is then used to guide the computation of several relevant functions associated with subtrees of the given tree. The structure of the spine decomposition, its properties, and the construction algorithms are detailed in Chapter 2.

1.1 Definition of problems covered

Facility location problems deal with the placement of one or more facilities with respect to a set of clients so that certain economic criteria are satisfied. In most situations, the intention is to minimize the cost associated with the service flow between facilities and clients, the cost being dependent on distance. Sometimes this cost is viewed as profit, in which case the goal is to maximize it. In practice, such problems occur in urban planning, industrial and computer engineering, data mining, vehicle routing, financial planning, *etc.* A few examples of location problems are the optimal placement of emergency stations in sparse rural areas, location of toxic waste recycling depots in dense urban areas, location of plants in a transportation system to minimize production costs, VLSI design, clustering applications in data mining, *etc.*

The k-median problem is a particular type of facility location problem among several variants. Depending on the function used as cost, we have:

- k-median (or min-sum) problems. The goal is to place k facilities to minimize the total service cost for all clients.
- k-center (or min-max) problems. The goal is to locate k facilities to minimize the service cost of the most expensive client.
- k-cover problem: One needs to select k facilities to maximize the total profit, where each client can only be served by a facility placed within a certain service radius.
- k-facility p-centrum problem. This is a generalization of median and center problems in which the optimum set of k facilities is to minimize the sum of the p largest service distances.
- uncapacitated facility location (UFL) problem. The question is to find the location of an appropriate number of facilities that minimize the total service cost and the total cost of opening facilities.
- capacitated facility location (CFL) problem. It is the same as UFL except that each facility can serve only a given number of clients.

All these variants but the last two impose a constraint on the number of facilities that are allowed to be located. In the case of UFL and CFL, a cost is paid to locate a facility at any given location, and thus one cannot chose to open an arbitrarily large number of facilities in the optimal solution.

The setting in which these problems are formulated is also important. There are problems defined in the plane where the clients are modeled as points or other geometric objects and the cost depends on various distance metrics which may also consider obstacles. Other instances are defined in graphs where vertices represent clients and potential facility locations. In a more general setting, the edges of a graph can be viewed as an infinite set of locations where facilities and sometimes clients can be placed. This model is called *network*, to distinguish it from a graph. The service cost depends on the length of the shortest path between locations in the network. The following paragraphs introduce the basic concepts used in this thesis.

Our notation follows the style of Reinhard Diestel's book on graphs [35]. A directed graph $G = (V, E)$ consists of two sets, V called the set of vertices of G, and a set of ordered pairs of elements from V, $E \subset V \times V$, called the set of edges of G. We denote the cardinality

of V by n. The set of vertices and edges of a graph G is referred to as $V(G)$ and $E(G)$ respectively. We may not always distinguish between a graph and its set of vertices or edges. For example we might speak of a vertex $v \in G$ and an edge $e \in G$. Given an edge $uv \in E$, we call u its source and v its sink. An undirected graph $G = (V, E)$ is a graph where the elements of E are unordered, *i.e.* $uv \in E \Rightarrow vu \in E$. In this case u and v are called adjacent vertices or neighbors. A vertex $v \in e$ is called incident to edge e; then e is an edge at v.

A path $\pi(v_1, v_n)$ is the graph

$$\begin{aligned}\pi(v_1, v_n) =&\big(\{v_1, v_2, \dots v_n\}, \{v_1v_2, v_2v_3, \dots v_{n-1}v_n\}\big)\\ =&\{v_1, v_2, \dots v_n\} \text{ (to relax the notation, we may omit the edge set)},\end{aligned}$$

where v_1, v_2, ... v_n are distinct. A cycle on the same set of vertices is graph

$$\pi(v_1, v_n) \cup \{v_nv_1\},$$

where "$\cup$" denotes the union of two graphs as the union of their vertex and edge sets. Here, only the edge set of the second graph is relevant. A tree is a connected, undirected graph that has no cycle as subgraph. A rooted tree is a tree $T = (V, E)$ with a special vertex r_T called the root and a function $p : V \to V$ called the parent function so that,
– $\forall uv \in E$, $u = p(v)$ or $v = p(u)$ but not both, and
– $p(r_T)$ is undefined.
Vertex v is also called the child of $p(v)$. A tree is called a binary tree if every vertex has at most two children. A vertex that has no children is called a leaf. If $u, v \in V$ are such that $u = p \circ \ldots \circ p(v)$, then u is an ancestor of v and v is a descendant of u. Here, operation $\circ$ denotes the composition of two functions. In particular, r_T is an ancestor for all vertices in the tree. Consider all paths that start at root r_T and end at a leaf. The number of vertices of the longest path (the path with the largest number of vertices) defines the height of the tree.

In our problems, graphs are augmented with two functions that associate a weight with each vertex and a length with each edge,

$$w : V \to \mathbb{R}_+, \text{ and} \qquad\qquad l : E \to \mathbb{R}_+.$$

The length $l(\pi)$ of a path π is the total length of all the edges in the path. The length function defines a distance in the graph as the length of the shortest path between two

vertices. If $\mathcal{P}(u,v) = \{\pi(u \dots v) | \pi(u,v) \subseteq G\}$ is the set of all paths in G between vertices u and v, then the distance $d(u,v)$ between u and v is

$$d(u,v) = \min_{\pi \in \mathcal{P}(u,v)} l(\pi).$$

We define the network N_G of graph G as an extension of G where every edge $uv \in G$ corresponds to a closed interval $[uv] \in N_G$. A point (or location) x can be chosen on edge $[uv]$ in which case we can refer to the edges $[ux]$ and $[xv]$. We can extend the edge length function on the network too, such that $l(uv) = l([uv]) = l([ux]) + l([xv])$. Between any two locations x and y in the network, we define as distance the length of the shortest path between x and y. In the remainder of this thesis we might not distinguish between a graph and its associated network. It will generally be clear from the context which object we are referring to.

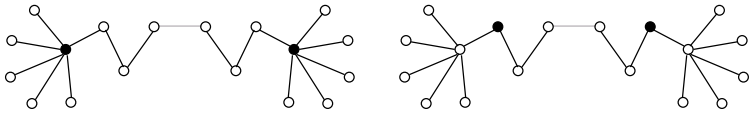
Before we formulate the k-median and k-center problems in graphs, we define two sets S and D of points from N_G, informally the set of supply respectively demand locations. Set S contains all locations in N_G where a facility can be placed, and D is the set of client locations. Most frequently $D = V$. Using the supply and demand sets above, the k-median and k-center problems are defined as to compute the following values,

$$\text{for } k\text{-median:} \qquad \min_{\substack{F_k \subseteq S \\ |F_k| = k}} \Big\{ \sum_{v \in D} w(v) \min_{x \in F_k} d(v,x) \Big\}$$

$$\text{for } k\text{-center:} \qquad \min_{\substack{F_k \subseteq S \\ |F_k| = k}} \Big\{ \max_{v \in D} w(v) \min_{x \in F_k} d(v,x) \Big\}.$$

The k-median problem is also referred to as the min-sum problem and the k-center as the min-max. Set F_k contains the locations at which the k facilities are placed and the cost of providing service from a facility to a client is modeled by the weighted distance between the two locations.

Sets S and D allow us to treat the k-median and k-center problems in a more general fashion by substituting S and D with either V (we restrict the supply or demand set to the vertices of the graph) or N_G (we allow supply or demand locations to sit on the edges of the network). We can use the positional notation of Tamir and Zemel [103] to refer to various versions of the k-median and k-center problems. The notation consists of a triple of symbols, for example $\underset{(S)}{V} / \underset{(D)}{V} / k$, which identifies the supply set, demand set, and number of facilities.



(a) (b)

Figure 1.1: Example of optimal solution for locating two facilities in a tree with unit vertex weights and unit edge lengths; the facilities are shown as dark dots. Case (a) 2-median. Case (b) 2-center

It should be noted that for the k-median problem, versions $\underset{(S)}{V}/\underset{(D)}{V}/k$ and $\underset{(S)}{N_G}/\underset{(D)}{V}/k$ are equivalent. An optimal k-median solution always exists in which all facilities are located at the vertices of the network even if they are allowed to sit on edges. This property, referred to as the vertex optimality of the k-median, was shown by Hakimi [51] and is reviewed in Section 1.2. Evidently, this is no longer true for the k-center problem (consider $k = 1$ and the simple graph $(\{u, v\}, \{uv\})$).

This thesis describes new algorithms for the k-median problem in rooted trees [11]. Three special cases are also considered, namely the 3-median problem in trees, the k-median problem in balanced binary trees, and the k-median problem in directed trees [13]. In these special cases, we propose simpler algorithms with a tighter upper bound on the running time than our general k-median algorithm. The class of balanced binary trees consists of trees with height logarithmic in the size of the tree as n tends to infinity. In directed trees, any facility is allowed to serve only clients for which it is an ancestor. The k-median problem for directed trees was introduced by Li *et al.* [74] as a mathematical model for optimizing the placement of web proxies to minimize average latency. These results are described in Chapter 3.

As mentioned earlier, we can use similar techniques to solve other facility location problems in trees. In this thesis we choose two generalizations of the k-median and k-center problem in trees, the mixed obnoxious median problem and the collection depots median location problems. In the following section we define these two types of problems rigorously and identify the instances solved by our approach.

1.1.1 Generalizations of the classical facility location problems

The mixed obnoxious median problem is a combination between the classic median location problem and the so called obnoxious facility location problem. As the name suggests, obnoxious facility location deals with the placement of a set of facilities that have an undesirable effect on clients and that should be placed as far away as possible. One can view such a problem as a usual k-median instance in which all clients are assigned a negative weight. The mixed obnoxious k-median facility location problem is then a k-median problem where some clients have positive weight while others have negative weight. The idea of mixing clients with positive and negative weight appeared in the work of Burkard *et al.* [22] and it reflects more accurately the practical situations encountered in real life. For example, the location of an industrial waste collection center is obnoxious for the residents in the area but desirable for the industries using it. In this case, one can assign negative weights to the locations containing residential complexes and positive weights to those with industrial designation.

The presence of vertices with negative weight adds to the complexity of the k-median problem. In fact, there are two possible interpretations of the objective function to be minimized,

$$\min_{\substack{F_k \subseteq S \\ |F_k|=k}} \Big\{ \sum_{v \in D} w(v) \min_{x \in F_k} d(v,x) \Big\} \quad \text{and} \tag{1.1}$$

$$\min_{\substack{F_k \subseteq S \\ |F_k|=k}} \Big\{ \sum_{v \in D} \min_{x \in F_k} w(v) d(v,x) \Big\}. \tag{1.2}$$

In the first version, every client in the demand set is served by the closest facility, irrespective of the sign of its weight. We refer to this problem as the WMD k-median because of the order in which $w(v)$, min, and $d(v,x)$ appear in (1.1). In the second version of the objective function, called for obvious reasons MWD, a client is served by the farthest facility if it has a negative weight. At a first glance it seems that problem WMD is identical to the classic k-median with positive weights, but in fact it is much more difficult. Burkard *et al.* [21] showed that if problem $\underset{(S)}{N_G} / \underset{(D)}{V} /k$ is considered, the optimal median set might not be a subset of the vertices of the tree. In contrast, problem MWD is easier, and the vertex optimality property holds. Burkard *et al.* solved the 1-median problem for trees and *cacti* in linear time [22]. A cactus is a graph that can have cycles, but any two cycles have at most one vertex in common. In a following paper [21], Burkard *et al.* considered the 2-median

problem in trees for both WMD and MWD variants, for which they proposed algorithms respectively cubic and quadratic in the size of the tree.

In this thesis, we improve the known results on the 2-median WMD [10] and 2-median MWD problems [12]. For the general case when $k > 2$, no algorithms have been proposed to date. It is very likely that a similar dynamic programming approach gives a polynomial algorithm in the general case too, however we do not tackle this problem here. The components that are served by the same facility interact in a more complex way than in the case of positive weights, and solving all issues is not straightforward. Details are given in Chapter 4.

The second generalization of the k-median and k-center problems has a vehicle routing flavor. We are given a set of locations where one of two types of objects are already placed, clients or collection depots. The facilities house a number of vehicles used for serving clients. We are asked to find a placement for one or more such facilities so that the following scenario is optimized.

- To serve a client, a vehicle starts at the facility, visits the client (to collect garbage, for example), then stops at a collection depot (to dump the garbage), and finally returns to the originating facility.
- All clients must be served.
- The objective function to optimize is:
  - Center problem: to minimize the cost (weighted distance) of the most expensive tour.
  - Median problem: to minimize the total cost (weighted distance) of all tours.

Consider tree T whose vertices are now given as the union of two sets not necessarily disjoint, C the set of client locations, and D the set of collection depots. Let $C = \{c_1, c_2 \dots c_{n_C}\}$ and $D = \{\delta_1, \delta_2 \dots \delta_{n_D}\}$, and $n = |C \cup D|$. Every client is associated with a positive weight $w(c_i)$. Given the location y of a facility in N_T, we denote by $r(y, c_i)$ the weighted distance of the route serving client c_i,

$$r(y, c_i) = w(c_i)\Big(d(y, c_i) + \min_{\delta \in D} \{d(c_i, \delta) + d(\delta, y)\}\Big).$$

Then, the k-median and k-center collection depots problems are expressed by

$$\min_{\substack{F_k \subseteq N_T \\ |F_k|=k}} \left\{ \sum_{c_i \in C} \min_{y \in F_k} r(y,c) \right\} \text{ and } \qquad \min_{\substack{F_k \subseteq N_T \\ |F_k|=k}} \left\{ \max_{c_i \in C} \min_{y \in F_k} r(y,c) \right\} \text{ respectively.}$$

This problem is the round-trip collection depots problem from [104] in which the set of depots is available to all clients.

Most of the results published regarding the collection depots problems characterize the optimal solution in planar settings, graphs, or trees [37, 16, 15]. A few heuristics for the median problem in general graphs also exist. For the center problem, Tamir and Halman [104] give several approximation algorithms for the metric problem, and exact algorithms for 1-center in graphs and k-center in trees. In this thesis, we look at 1-median and k-median problems in trees. Our results are discussed in Chapter 5.

1.2 Brief characterization of the problems

This section describes in a condensed format a few results that are indicative of the hardness of k-median and k-center problems in graphs. Since the classical k-median and k-center facility location problems are special cases of collection depots and mixed obnoxious location problems, one can conclude that the complexity of the later problems is at least as that of the former.

1.2.1 Complexity of k-median and k-center

This section reviews the complexity results for two important problems in facility location. Notions regarding complexity theory are not summarized, but can be found in the books by Meyr *et al.* [81], Bovet and Crescenzi [19], Garey and Johnson [42], or Papadimitrou [85].

Formulating the k-median as a decision problem, Kariv and Hakimi [67] showed that it is NP-complete even when the input graph is planar with unit edge lengths and with a maximum degree equal to 3. The proof uses a simple transformation from the dominating set problem which was shown to be NP-complete by Garey and Johnson in [42]. For the geometric version where clients are points in the plane and facilities are to be located anywhere so that to minimize the total Euclidean or rectilinear distance, the k-median and k-center are still NP-complete. This was shown by Megiddo and Supowit in [80].

Definition 1.1 (k-median (k-center) decision problem). Given a graph $G = (V, E)$, a positive integer k, and a real value c, does there exist a set of vertices $V_k \subset V$ with $|V_k| \leq k$ such that the cost of the k-median (k-center) problem with V_k as facilities is no more than c?

It is obvious that both k-median and k-center are in the class NP of problems because for a set of facilities given as a certificate, it is easy to compute in polynomial time the cost of the k-median or k-center and compare it with c. We consider this fact established and we will not mention it again in our following discussion about computational complexity.

Definition 1.2 (dominating set problem). Given a graph $G = (V, E)$ and a positive integer k, does there exist a set $V_k \subset V$ with $|V_k| \leq k$ such that each vertex of G is either in V_k or is adjacent to a vertex in V_k? IF the answer is "yes", V_k is called a dominating set.

Theorem 1.1 (Kariv and Hakimi [67]). *The k-median problem is NP-hard even in the case of a planar graph of maximum vertex degree 3, whose edges have unit length.*

Proof idea. Assume all edges of the graph have unit length. A dominating set for G of cardinality k exists if and only if the cost of the optimal k-median of G is $n - k$, where $n = |V|$. In that case, the set of medians is also a dominating set for G. □

A similar result can be established for the center problem, the reduction being done again from the dominating set. Both types of center problems, $\underset{(S)}{V}/\underset{(D)}{V}/k$ called the discrete k-center and $\underset{(S)}{N_G}/\underset{(D)}{V}/k$ known as the absolute problem, are NP-complete.

Theorem 1.2 (Kariv and Hakimi [66]). *The discrete k-center problem is NP-hard even in the case of a planar graph of maximum vertex degree 3 with unit edge length and vertex weight.*

Proof idea. The proof follows immediately from the observation that a dominating set of size less than k exists if and only if the radius of the optimal k-center solution is at most 1. □

Theorem 1.3 (Megiddo and Tamir [79]). *The $\underset{(S)}{N_G}/\underset{(D)}{N_G}/k$ center problem is NP-hard even in the case of a planar graph of maximum vertex degree 4 with unit edge length and vertex weight.*

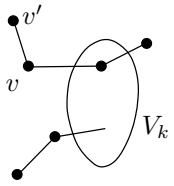


Figure 1.2: Proving the NP-completeness of the continuous k-center problem

Proof idea. Consider graph $G = (V, E)$ for which the dominating set problem must be answered. From G construct graph $G' = (V', E')$ by adding a vertex v' for every vertex $v \in V$ and making v' adjacent only to v (see Figure 1.2). Every edge in G' has length 1. Then, one has to show that a dominating set of size k exists in G if and only if the optimal k-center for G' has a cost of at most 2. Clearly, a dominating set of size k in G determines directly in G' a k-center of cost exactly 2. For the converse, consider a set of k centers whose radius is at most 2. Then, the longest path that must be covered by any center is only of type $(u'uvv')$. Therefore, moving centers to their closest vertex from V will not increase the k-center cost over 2 and would directly give a dominating set of size k for G. □

Both k-median and k-center problems are no longer NP-hard when either k is considered constant, or if the input are trees, interval graphs, or circular arc graphs. If k is constant, a polynomial brute-force algorithm can look at all possible subsets V_k of cardinality k of vertices (there are $\binom{k}{n} \in O(n^k)$ possibilities), can compute the objective function for each placement, and can retain the one with minimum cost. For the special types of graphs mentioned above, the distance function has specific properties that yield polynomial time algorithms for most location problems. In the following section some of these properties are reviewed.

1.2.2 Vertex optimality of k-median

An important property of the optimal solution for k-median is that the facilities may be placed only at the vertices of the network, even if locating facilities on edges is allowed. This was proved by Hakimi [51].

The argument comes from the way distance to a vertex varies from a point moving continuously on an edge. Consider network edge $e = [uv]$ and a function $f_e : [0, l(e)] \to [uv]$ that defines the moving point on e such that $d(u, f_e(\alpha)) = \alpha$. Let x be an arbitrary vertex of G. Then the distance from x to the moving point $d(x, f_e(\alpha))$ is a concave function of α

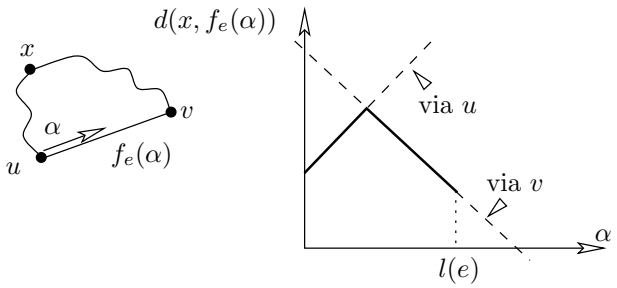


Figure 1.3: Concavity of the network distance from a moving point on an edge to any given location on G

(see also [70] for a good exposition of the properties of network distance in general graphs and trees). Concavity of the distance is caused by the presence of two possible routes that could be taken by the shortest path from $f_e(\alpha)$ to reach x, one through u, the other through v (see Figure 1.3). Since the objective function of the k-median problem is a summation of weighted distances with positive weights, facilities located at the vertices of the network can achieve the minimum median cost.

1.2.3 Approximation results

Because of the NP-completeness of k-median and k-center problems in general graphs, a significant part of the research has been focused on finding good approximation algorithms. Unfortunately, Lin and Vitter [75, 76] showed that approximating k-median is as hard as approximating dominating set and set cover, and therefore, it is unlikely that there exist constant factor approximation algorithms for both problems. More details regarding the approximability of set cover can be found in the work of Feige [38] and Meyr *et al.* [81].

When the service cost between pairs of locations satisfies the triangle inequality, constant factor approximations are possible. Several constant approximation algorithms were recently published. Table 1.1 gathers some of these results. There, n represents the number of vertices, m the number of edges, L the value of the largest number from input, and $p \leq k$ is a positive integer chosen by the user.

For the k-center problem, the situation is similar. If the pairwise service cost does not satisfy the triangle inequality, Hochbaum [59] showed that constant factor approximations are not possible. When the triangle inequality is satisfied, an approximation factor of 2 is the best one can hope for both in unweighted [46, 60, 63, 86] and weighted [87, 86] cases.

Approx. factor	Run. time	Reference
6	$O(n^2 \log(n(L + \log n)))$	Jain, Vazirani [64]
4	$O(n^3)$	Charikar, Guha [23]
$12 + o(1)$	$O(m)$	Thorup [106]
$3 + \frac{2}{p}$	$O(n^p)$	Arya *et al.* [6]
$\epsilon > 0$ (planar)	$O(n^{O(1+\frac{1}{\epsilon})})$	Arora *et al.* [5]

Table 1.1: Constant factor approximation algorithms for the metric k-median problem

1.2.4 Final remarks

The previous paragraphs give an overview of k-median and k-center problems in general. It is interesting to notice that in terms of their complexity, both problems are the same. However k-median seems to be easier to solve because the candidate set of optimal facilities for k-median is just the vertex set, whereas for k-center, this set is more complex. Surprisingly, the algorithmic results, including the performance of the approximation algorithms, seem to indicate otherwise. Perhaps this is due to the duality between center and cover problems which is exploited in many algorithms for the center problem. A similar duality between cover and median problems can be formulated too, but at the expense of increasing the size of the cover problem quadratically [68]. Recently, Demaine *et al.* [34] proved that the unweighted k-center problem on planar graphs is fixed parameter tractable [36] when parameterized by the number of facilities k and the cost (radius) of the solution. This means that there is an algorithm with a running time that depends on two factors. One factor is polynomial in the size of the input graph and the other is exponential in k and the value of the radius. As a result, the k-center problem can be solved efficiently on large planar graphs provided the number of facilities and the value of the radius are small. It is not known yet whether the k-median problem has a similar behaviours or not.

1.3 Thesis motivation

At the beginning of this thesis we mentioned that our results relate to the k-median facility location problem for trees and two generalizations, the collection depots and the mixed obnoxious facility location problems. Our choice of problems is motivated both by practical and theoretical concerns.

From a theoretical point of view, we were are able to design a different algorithm for

a fundamental problem in optimization, the k-median problem in trees. As mentioned by Tamir *et al.*[102], k-median algorithms for tree networks are useful in deriving approximations for k-median problems in general graphs. In addition, we believe that our result also advances the research towards finding ways to exploit the special structure present in graphs that have not yet been considered for the k-median and other facility location problems. An example are graphs with bounded tree-width (partial q-trees) [89] which have a configuration that resembles a tree to some extent. Many optimization problems have been successfully solved on such graphs by exploiting this resemblance, however few results in the same direction are known for the k-median problem. Some contributions concern optimization problems in partial q-trees that are related to k-median. These might be used as starting points for research on median problems in such special graphs. We mention here the work of Gurevich and Stockmeyer [49] who designed an algorithm for the continuous minimum cover problem whose running time is proportional to the number of edges of the graph and exponential in the value of a parameter that measures how different is the given graph from a tree. In partial 2-trees, Hassin and Tamir [55] give an $O(n^4)$ algorithm to solve the uncapacitated facility location problem and an $O(n \log^3 n)$ for selecting the k-th longest path. The later procedure can be used in an algorithm to solve the k-center problem. Granot and Skorin-Kapov [47] look again at uncapacitated facility location problems and propose an $O(n^{q+2})$ algorithm for partial q-trees. They also study a more general distance function in a graph called p-cable distance determined by the total length of p vertex disjoint paths between two vertices. Chaudhuri and Zaroliagis [25] consider shortest path queries in partial q-trees, and give an algorithm that answers any query in $\alpha(n)$ time after linear-time pre-processing ($\alpha(n)$ is the inverse Ackermann function). Designing efficient k-median algorithms for partial q-trees might also reveal information regarding the parameterized complexity of the k-median problem, an issue neglected until now. Parameterized complexity was pioneered by Downey and Fellows [36] to supplement the results of classical complexity theory. The motivation behind parameterized complexity comes from the observation that certain problems deemed hard by classical complexity theory can actually be efficiently solved, even for large instances, as long as some other parameters are small in size. An example of such a situation is given by Integer Programming, which can be solved in polynomial time if the number of variables is fixed (see Lenstra [72]), or by Linear Programming solved in linear-time if the dimension is fixed (see Megiddo [78]).

Our algorithm is also suitable in practice. It is not too difficult to implement since it

deals mostly with the recursive computation of cost functions and the data structures used are standard. In addition, our algorithm is flexible enough that it can also be implemented with other decompositions of trees, even with the centroid decomposition although the performance is affected in this case. Unfortunately, the bound on the running time of our algorithm is exponential in k, whereas the classic k-median problem has a complexity of $O(kn^2)$ for variable k. However, a conjecture by Chrobak *et al.* [28] states that the size of the cost functions handled by our algorithm is linear in the number of vertices of the subtrees for which the functions are defined. If true, this might lead to designing a sub-quadratic algorithm that is much more efficient, with a performance perhaps not even exponential in k for weighted arbitrary trees. There is strong indication that the conjecture is true. Tamir [101] mentioned an unpublished result by Rahul Shah who designed a sub-quadratic k-median algorithm for un-weighted balanced binary trees with a running-time of $O(k^2 n \log n)$ where k is variable.

1.4 Our collection of techniques

The k-median problem has been a subject of study for several decades. In general, it is NP-hard (see Section 1.2.1), but when the input graph is a tree, polynomial time algorithms are possible. Kariv and Hakimi in their seminal paper on facility location [67] gave an $O(k^2 n^2)$ algorithm based on dynamic programming. Hsu proposed a different algorithm with complexity $O(kn^3)$ [62], but later on Tamir [99] explained that the dynamic programming algorithm has a tighter bound of $O(kn^2)$ on the running time.

When $k \leq 2$ more efficient algorithms exist. For the 1-median a linear time algorithm was proposed by Goldman [44] in 1971. A characterization of the 1-median was given independently by Sabidussi [91], Zelinka [110], Kang and Ault [65], and Kariv and Hakimi [67]. They showed that the optimal 1-median of a vertex weighted tree coincides with the w-centroid of the tree. The w-centroid is a vertex $v \in V$ such that the total weight of the subtree rooted at any of the neighbors of v has no more than half the total weight of the tree. This property turned out to be very important for the algorithms that tackled the case $k = 2$ later on.

For the 2-median Gavish and Sridhar [43] designed a method with a running time of $O(n \log n)$ based on a complex data structure of Sleator and Tarjan [97] for maintaining a collection of disjoint trees. A simpler algorithm with the same complexity was recently

proposed by Breton [20]. It uses a tree decomposition data structure that allows updates of the optimal 1-median solution of any subtree of the original tree in logarithmic time. Breton [20] also showed that a linear time algorithm for the 2-median is possible if the tree edges are of the same length or if the vertex weights are sorted. To achieve such a performance, additional properties of the relative placement of the two optimal medians in a tree were used. These properties had been studied by Sherali and Nordai [96], and Mirchandani and Oudjit [83]. Auletta *et al.* [7] proposed a linear time algorithm for the 2-median problem in a tree in general, but unfortunately their analysis was flawed. It seems that the quest to find a linear time algorithm for the 2-median problem in general is very difficult so perhaps a running time of $O(n \log n)$ is the best one can hope for. However, for any other values of k, no specific algorithms are known.

In contrast, sub-quadratic algorithms for the uncapacitated facility location and for the minimum cost coverage problems in trees were recently proposed by Shah and Farach-Colton [93]. Their solution is based on a modified dynamic programming formulation called *undiscretized dynamic programming*, which first appeared in a paper by Shah, Langerman, and Lodha [94], for solving the problem of placing filters in a multi-cast tree. Unfortunately, the authors of [93] were unable to extend their approach to solving the k-median problem because they could not handle the cardinality constraint on facilities. To illustrate the idea behind undiscretized dynamic programming, we will review the basics of the k-median algorithm as formulated by Tamir [99]. Then we will describe how we avoided the difficulties of Shah and Farach-Colton in our sub-quadratic algorithm for the k-median.

1.4.1 Dynamic programming

Before we proceed with our description of the algorithm, we need to introduce some terminology used throughout the thesis. We always work with a binary tree $T = (V, E)$ rooted at vertex r_T and having positive vertex weights and edge lengths, as defined in Section 1.1. If the tree is not binary, it can be transformed into a binary one by adding a linear number of vertices and edges with zero weight as in Tamir [99].

Let $M = \{m_1, m_2, \ldots m_k\} \subseteq V$ be a set of k medians in T. Every vertex v is closest to one or more vertices from M. Let $V_i \subseteq V$ be the set of vertices closest to a particular $m_i \in M$, ties being broken in such a way that V_i stays connected,

$$V_i = \big\{ v \in V : d(v, m_i) = \min_{m_j \in M} d(v, m_j) \big\}.$$

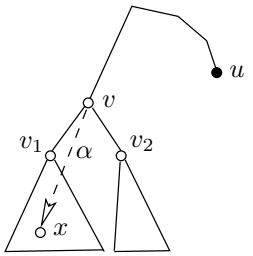


Figure 1.4: Recursive computation of a classic dynamic programming cost function

We say that any vertex $v \in V_i$ is *covered*, or *served* by m_i. Sets V_i partition the tree in k connected components separated from each other by $k-1$ edges called *split edges*. Moreover, m_i is the 1-median of the subtree induced by V_i. Thus the problem of finding an optimal set of k medians can also be formulated as the task to find an appropriate set of $k-1$ split edges.

The basic idea behind dynamic programming algorithms for optimization problems in trees is to find an optimal solution by solving subproblems defined on subtrees of the input tree. The subproblems have a trivial solution for inputs of small size (for example if the subtree contains only one vertex) and a relatively simple solution based on the result returned recursively on smaller components. Each subproblem must be constrained by the global solution through some choice made outside the subtree. The collection of solution values for the sub-problems can be viewed as a set of functions called "cost functions". Such methods are efficient on trees because the interaction between a connected subtree and the rest of the graph takes place only through an edge. For example, for the k-median algorithm in Tamir [99], the influence of the global solution on the subproblem is determined by the choice of the facility closest to the root of the subtree.

More precisely, Tamir defines two cost functions for each vertex $v \in T$. Recall from Section 1.1 that $p(v)$ represents the parent of vertex v. We denote by $T(v)$ the component of T containing v obtained by removing edge $vp(v)$. If $v = r_T$, then $T(r_T) = T$. $T(v)$ is in fact the subtree rooted at v induced by v and all descendants of v. The functions are:

- Function $G(v, \alpha, p)$ which returns the optimal cost in $T(v)$ provided that at least one facility is placed in $T(v)$ at no more than distance α from reference vertex v, and at most p facilities are placed in $T(v)$.
- Function $F(v, u, p)$ which returns the optimal cost in $T(v)$ if at most p facilities are placed in $T(v)$ and the closest facility in $T \setminus T(v)$ is at vertex u.

Clearly, the optimal k-median solution is returned by $G(r_T, \infty, k)$. The functions are computed recursively from the value of F and G associated with the children of v,

$$F(v,u,p) = \min\Big\{G\big(v, d(u,v), p\big), \\ w(v)d(v,u) + \min_{0\le j\le p}\big\{F(v_1,u,j) + F(v_2,u,p-j)\big\}\Big\}, \tag{1.3}$$

$$G(v,\alpha,p) = \min\Big\{G(v,\alpha^-,p), \\ w(v)\alpha + \min_{0\le j\le p}\big\{G\big(v_1,\alpha - d(v,v_1),j\big) + F(v_2,x,p-j)\big\}\Big\}. \tag{1.4}$$

Equations (1.3) and (1.4) illustrate only one case of the computation. They are given here as an example (see Figure 1.4).

In (1.3), we decide whether new vertex v is better served by facility u outside, in which case we use functions F at v_1 and v_2, or whether it should be served by some facility within $T(v)$, in which case we use function G at the children nodes. For reasons that are perhaps not obvious now but will be explained at the beginning of Chapter 2, we should point out that (1.3) is correct because reference vertex v in function G is the vertex of $T(v)$ that is closest to any vertex from $T \setminus T(v)$.

For (1.4) we consider the distances from v to every vertex in T sorted in increasing order. Then α^- is the largest distance smaller than α in this ordering, and *w.l.o.g.* $x \in T(v_1)$ is the vertex corresponding to distance α. Equation (1.4) determines the location of the facility closest to v within $T(v)$. In this expression, we choose between placing a facility at x using the cost of functions G and F at v_1 and v_2, and placing it somewhere closer to v , by using the value of function G at v.

With this choice of dynamic programming cost functions, there is little hope for getting a sub-quadratic algorithm because for every parameter v there is a number linear in n of functions F and G to be calculated. However, in computing the optimal cost returned by function F in a subtree, we do not have to take into account the closest median outside the subtree but only the distance to it. Hence, we can replace discrete parameter u from $F(v,u,p)$ with a continuous parameter α that represents the distance from v to u, the closest external facility. In this way, we are working with a piecewise linear continuous function whose complexity depends only on $|T(v)|$. This is the idea exploited by the *undiscretized dynamic programming* algorithms. The term was used for the first time by Langerman *et al.* [94] for a problem related to the k-median, the problem of placing filters in a multi-cast tree, where the use of continuous cost functions allowed them an $O(n \log n)$ solution.

Other facility location problems on trees, such as the uncapacitated facility location problem (UFL), were approached from the same angle by Shah *et al.* [93] who designed $O(n \log n)$ algorithms for UFL and minimum cost coverage problems. They used the properties of the continuous cost functions which are either convex or concave and devised a data structure to handle them efficiently. With their data structure, Shah *et al.* could implement the recursive computation of cost functions in time linear in the size of the smallest function involved in the procedure. This is the reason why the total running time for computing and using cost functions is sub-quadratic in n. Unfortunately their approach could not solve the k-median problem because cost functions use an additional parameter specifying the number of facilities.

To avoid the difficulties of Shah *et al.*, we propose a different way to obtain a sub-quadratic algorithm. Let $f(|T(v)|)$ denote the complexity (number of linear pieces) of $F(v, r, p)$. The total size of all functions F computed becomes $O(kf(n)h)$, where h represents the height of the tree[1]. If we can prove that $f(n)$ is sub-quadratic and if we apply our algorithm on the class of trees with logarithmic height, then the total size of all cost functions computed is sub-quadratic. To accommodate the class of trees with linear height too, we decompose the input tree into a family of nested subtrees called components, that partition the tree recursively with logarithmic depth. The type of decomposition we employ is called *spine decomposition* and is described in detail in Chapter 2. Then, we associate the cost functions with the components of the decomposition rather than with subtrees $T(v)$. The upper bound on the total size of cost functions becomes $O(kf(n) \log n)$ because the recursive depth of the spine decomposition is $O(\log n)$. Key to our algorithm is to show that $f(n)$ is sub-quadratic in n. The same idea of using continuous functions allowed us to improve the algorithms for positive/negative 2-median problem in the WMD formulation.

Working with continuous functions and decompositions of trees are not the only common traits of our algorithms. The structure of our decomposition is such that it clearly represents parts of the tree (paths called spines) through which different components interact. In the same time, this structure restricts somewhat the type of information that can be efficiently pre-computed. In order to make use of the data available, we need to carefully process it during the computation phase too. We realized that by representing data in two dimensional space (cost vs. distance), we can employ simple computational geometry algorithms to

[1] Assume $f(a) + f(b) \leq f(a + b)$.

achieve the desired outcome. The idea was previously used by Hassin and Tamir [56] to solve facility location problems on the line, and independently by Auletta *et al.* [8] who revisited Hassin and Tamir's k-median problem in a path. Auletta's algorithm is in essence the same, except that their interpretation is not geometric and thus somewhat more difficult to follow.

In Chapter 2 we describe and analyse the properties of the decomposition used in almost all algorithms discussed in the thesis. Then, in Chapter 3, we start by presenting the framework of our undiscretized dynamic programming algorithm for the k-median problem in trees. We identify a sub-problem more difficult to address and solve it for (i) directed trees, (ii) balanced trees, (iii) arbitrary trees but for $k = 3$, and (iv) arbitrary trees for any fixed value of k. In Chapter 4, we study the 2-median problem in trees when vertices may have negative weights as well. We give new algorithms for both MWD and WMD objective functions. In Chapter 5 we consider another generalization of the k-median problem, the collection depots median problems. We give an algorithm for solving the 1-median case in a tree and show that it is possible to adapt Tamir's k-median algorithm to include costs that depend not on weighted distance but on weighted round-trip trip distance. Finally, we conclude in Chapter 6 with a summary of our results and directions for future research.

Chapter 2

The spine decomposition of trees

We argued in Section 1.4.1 that we need a decomposition that partitions input tree T into recursive connected components such that the depth of the recursion is $O(\log n)$ where n is the number of vertices in the tree. In this chapter we describe the decomposition used in most of the algorithms presented here. The description is problem independent. The bounds on storage space and running time from this chapter refer only to the data structures internal to the decomposition and to their initialization procedures.

First, we define what we mean by a decomposition of a tree. Note that in this thesis, we might refer to a decomposition of a tree calling it also a "tree decomposition". Unless stated otherwise, the term tree decomposition has the semantics described by the definition below and does not refer to the notion introduced by Robertson and Seymour [89] in the context of tree decompositions of graphs. It is important to define the height or depth of a decomposition which is a parameter that directly influences the running time of the

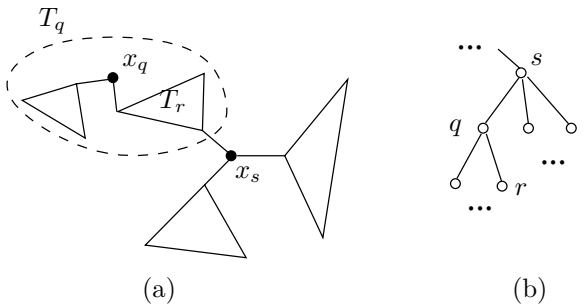


Figure 2.1: Part of a tree (a) and its centroid decomposition (b)

algorithms based on the decomposition.

Definition 2.1 (Decomposition of a tree). A decomposition of tree T is a set of subtrees of T denoted $D(T)$ that satisfies the following two properties:
P.1) $T \in D(T)$, and
P.2) $\forall T_1, T_2 \in D(T)$ either T_1 and T_2 are disjoint, or one is strictly contained in the other.
An element of $D(T)$ is called a component of the decomposition.

Definition 2.2 (Height of a decomposition). The height or depth of a decomposition $D(T)$ is the maximum cardinality of a subset of components $H \subseteq D(T)$ whose elements strictly contain one another,

$$H = \{T_1, T_2, \ldots T_k : T_1 \subset T_2 \subset \ldots \subset T_k\}.$$

In the literature, there exist several types of decompositions of trees, perhaps the best known being the centroid decomposition (CD) [30].

The set of components of a CD can be defined recursively as follows. Assume subtree T_s is in the decomposition. Then, a special vertex x_s from T_s is identified. This vertex is called the *centroid* of T_s (see Figure 2.1).

Definition 2.3 (Centroid). A centroid of subtree T_s is a vertex $x_s \in T_s$ with neighbors denoted y_1, y_2 ... y_t such that the size of every component of T_s obtained by removing x_s is no more than half the size of the subtree,

$$|T_s(y_i)| \leq \frac{1}{2}|T_s|.$$

The entire set of components $T_s(y_i)$ is added to the centroid decomposition and the procedure is applied recursively on each $T_s(y_i)$ until the components obtained reach size one. Note that a centroid need not be unique, as for example in a tree with two vertices and one edge, $(\{u, v\}, \{uv\})$. If this occurs while the CD is built, one of the centroids is chosen arbitrarily.

The CD can be represented as a rooted tree whose nodes correspond to subtrees of the original tree, the components of CD. The root of the CD denoted s_{CD} maps to entire input tree T. The children of CD node s that corresponds to component T_s are nodes that map to each of the components $T_s(y_i)$ defined above. In this representation, the height of the CD translates directly to the height of the binary tree that represents the CD.

From the definition of the centroid, it immediately follows that the height of the decomposition is $O(\log n)$ and the requirement set forth in the previous chapter is satisfied. Unfortunately, in the decomposition there is no control on, or representation of the path between a centroid – for example x_s in Figure 2.1 – and the centroids at the level below and above – for example x_q in the same figure. In our k-median problem we need control over such a path because of the interaction between vertices in a component and the outside world. In addition, we need to make sure that whatever we compute at some lower level in the decomposition remains valid at higher levels too.

Take for example the computation of function $F(r, u, j)$ from function $G(r, \alpha, j')$ as described in Section 1.4.1 by Equation (1.3). Recall that $G(r, \alpha, j')$ returns the cost of T_r if there is at least one facility within distance α from a special reference vertex of T_r. We notice that the reference vertex used in $G(r, \alpha, j')$ depends on the position of the external vertex u that appears as parameter in function $F(r, u, j)$ to be computed. More precisely, the reference vertex is the vertex of T_r that is closest to u. But $u \in T \setminus T_r$ and because of the structure of the CD there could be many different reference vertices that have to be considered. In fact, one can show that there are $O(\log n)$ possible reference vertices at any given component of a centroid decomposition which means that $O(\log n)$ different functions G need to be calculated at every CD node. But this becomes too cumbersome. Ideally we would like to have, if not a single version of function G, at least a constant number of them.

For this reason we use instead of the centroid decomposition, the spine decomposition (SD) described in two of our previous papers [13, 9]. The SD is applied on rooted binary trees. It is built around tree paths called *spines* that connect the root with leaves of the tree. Once a spine is identified, the decomposition is applied recursively on the components obtained by removing the path from the tree. In this way, the interaction of the entire tree with a component is localized; it takes place only through the spine. The main concern remains now to insure that the depth of the decomposition is logarithmic. We can achieve this by selecting the spines carefully. The following section contains a detailed exposition of the structure of the SD.

2.1 The structure of the SD

Without loss of generality (*w.l.o.g.*), consider a rooted binary tree T with root r_T. If the tree is not rooted, we can root it at an arbitrary vertex. If it is not binary, we can transform

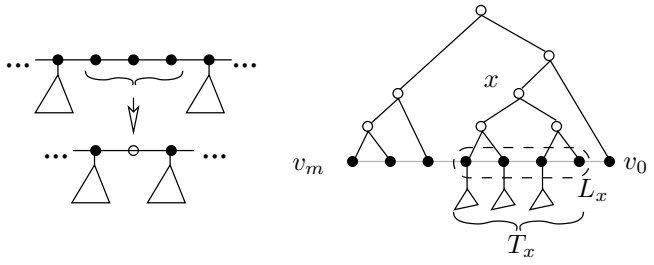
it to a binary one by adding a linear number of new vertices and edges with zero weight as in the paper of Tamir [99].

A simple way to select paths and to control the depth of the recursion is the following. Recall that $p(v)$ is the parent of v. Let $N_l(v)$ be the number of leaves from the set of all descendants of v in T. A path $\pi(r_T, l)$ from r_T to a leaf l of T is first identified such that for any two consecutive vertices v_i and v_{i+1} on the path, the following conditions are satisfied.
Cond. 1: $v_0 = r_T$ and $p(v_{i+1}) = v_i$, and
Cond. 2: if u_i and v_{i+1} are the two children of v_i, then $N_l(v_{i+1}) \geq N_l(u_i)$.
In this way, the path follows vertices from the root to a leaf such that the next vertex chosen is always the child of the current vertex with the most number of leaf descendants. The procedure is then applied recursively on each $T(u_i)$ obtained from T by removing vertex v_i. Path $\pi(v_0, v_m)$ is called a *spine*, subtrees $T(u_i)$ are called *spine components* or components of spine $\pi(v_0, v_m)$, and vertices on the path are called *spine vertices*. Similar to the centroid decomposition, spine components have at most half the number of leaves of the original tree. Clearly, the total number of spines ever constructed for a tree equals the number of leaves of the tree[1] and therefore the recursive depth of this method is $O(\log l)$, where l is the number of leaves of T.

Unfortunately, a spine has length $O(n)$ and therefore $O(n)$ spine components including spine vertices of degree two in T must be handled for every given spine. For our purpose, this situation is not suitable. Consider as example a dynamic programming algorithm that pre-processes the information bottom up. In this setting, it becomes quite complicated to process and gather information that is distributed among $O(n)$ entities in order to pass it one level of recursion up. Ideally, we would like to combine functions stored for only two subtrees as in the case of the classic dynamic programming algorithm of Tamir illustrated in Section 1.4.1.

There are two ways to reduce the number of components found at the same recursive level, (i) by collapsing successive spine vertices of degree two into a single component called *super-node* (Figure 2.2 (a)), and/or (ii) by constructing a binary search tree on top of every spine (Figure 2.2 (b)). The benefit of super-nodes is obvious. If they are included, the number of components on any spine becomes $O(l)$, where l is the number of leaves of T. However, using super-nodes alone is not enough. The following section describes the search

[1] We do not count the root vertex as leaf if it has degree one.



(a) (b)

Figure 2.2: Ingredients of a spine tree decomposition: (a) a super-node; (b) a binary search tree

trees of the spine decomposition in detail and explains how they are used.

2.1.1 SD binary search trees

Let $\Pi = \{v_0, \dots v_m\}$ be a given spine and '$\preceq$' a total order relation on Π determined by the sequence of vertices on the spine,

$$v_i \preceq v_j \Leftrightarrow i \leq j.$$

Let u_i be the child of v_i not on the spine, and $T(u_i)$ the spine component adjacent to v_i. Based on total order relation $\preceq$, we construct a balanced binary search tree S_Π with root s_Π and whose set of leaves is Π. For the first spine computed in the whole input tree, the root of the search tree is denoted s_{SD} and is called *root of the spine decomposition* (Figure 2.3). For every $x \in S_\Pi$, let $L_x \subseteq \Pi$ be the set of leaves descending from x in S_Π. Similarly, let T_x be the subtree of T induced by L_x and all spine components adjacent to some $v_i \in L_x$ (see Figure 2.2 (b)),

$$L_x = \{v_i \in \Pi : x \text{ is ancestor of } v_i\},$$
$$T_x = T\big(L_x \cup \bigcup_{v_i \in L_x} T(u_i)\big).$$

In particular, $T_{v_i} = T\big(v_i \cup V(T(u_i))\big)$. We can now define the spine decomposition of tree T.

Definition 2.4 (Spine decomposition (SD)). The spine decomposition $SD(T)$ of tree T is the decomposition (see Definition 2.1) generated by subtrees T_x of T defined for all nodes x of all balanced binary search trees.

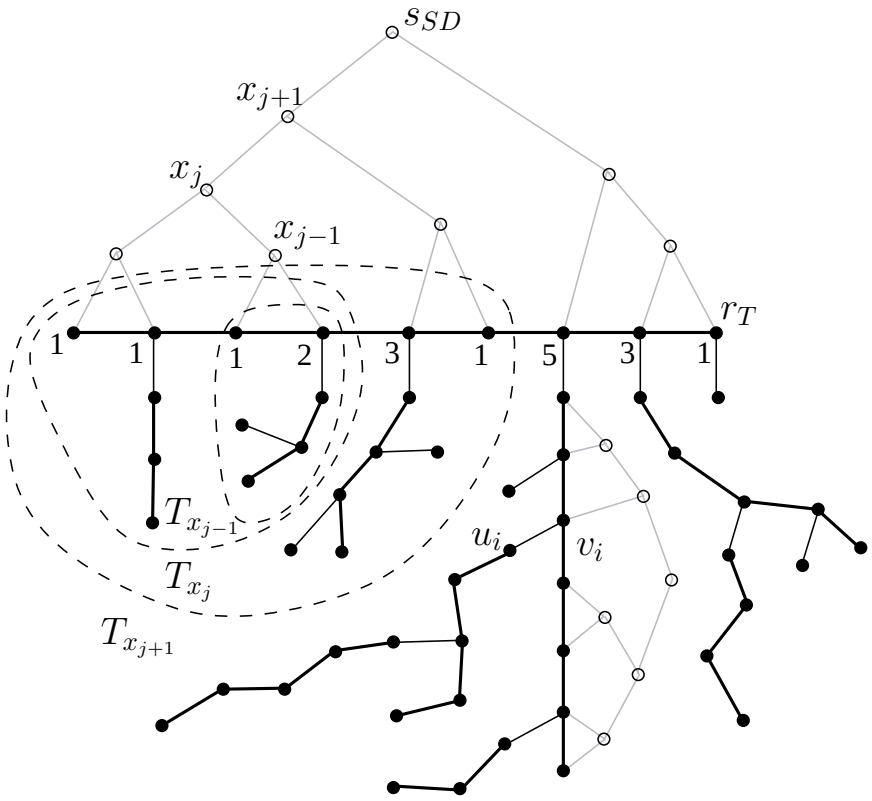


Figure 2.3: A typical spine decomposition; spines are shown in thick lines, search trees as thin lines and components are outlined by dashed lines; the numbers beside spine vertices at the top-most spine give the number of leaves of T for the corresponding SD component

It is more convenient to work with the balanced binary search trees because they illustrate the inclusion relationship that might exist between the components of a decomposition (Property P2 from Definition 2.1). For example, if x_0, x_1, ... x_i, ... are consecutive search tree nodes on a path from leaf to root in a search tree, then

$$T_{x_0} \subset T_{x_1} \subset \ldots \subset T_{x_i} \ldots$$

The inclusion relationship is exploited by our dynamic programming algorithms for the computation of cost functions. Each cost function associated with component T_x of the decomposition is recursively calculated from the value of the cost functions associated with the two subtrees included by T_x. Therefore, the notion of path in a search tree is important, and we denote it by $\sigma(x, y)$ where x and y are any two internal nodes or leaves of a search tree. For our purposes, one of the nodes x and y will always be an ancestor of the other, and thus the paths we work with are always oriented from root to leaf or vice versa. Given now a spine vertex v_i from spine Π with a component $T(u_i)$ adjacent to it, we notice that SD component T_{v_i} includes $T(u_i)$. Subtree $T(u_i)$ has its own spine Π' different from Π and its own search tree. To capture the inclusion relationship between T_{v_i} and $T(u_i)$, we can extend the notion of path in a search tree so that it spans several search trees. We can consider that search tree root $s_{\Pi'}$ of $T(u_i)$ is connected to leaf v_i in the search tree over Π. In this way, we can work with paths $\sigma(x, y)$ even when search tree nodes x and y belong to different search trees. Program 2.1 illustrates one usage scenario for our decomposition.

Now, the recursive computation of cost functions at node x involves only functions computed at the two children of x. What remains to be done is to make sure that the recursion depth using search trees is logarithmic in n. More precisely, we need to show that the length of any path $\sigma(s_{SD}, v)$ from the root of the SD s_{SD}, to any tree vertex v sitting on some spine, is $O(\log n)$.

This requirement is not difficult to satisfy. We want to assign keys to the internal nodes of S_Π such that the root s_Π of the search tree is closer in S_Π to a spine vertex $v_i \in \Pi$ whose component $T(u_i)$ has a large number of leaves in T, than to any other spine vertex $v_j \in \Pi$ whose component has fewer tree leaves of T. Intuitively, the structure of the search tree tries to compensate for the greater effort consumed during searching inside spine components with a large number of leaves. Here is one way of doing this.

We assign each vertex $v_i \in \Pi$ a weight $\lambda(v_i)$ equal to the number of leaves of tree $T(u_i)$, and thus also to the number of spines of T_{u_i}. If v_i is of degree two ($T(u_i) = \emptyset$), then

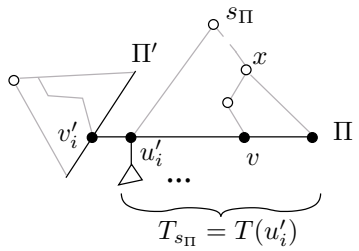


Figure 2.4: Illustration of variables from Program 2.1

$\lambda(v_i) = 1$. Weight $\lambda(v_i)$ is used in balancing the binary search tree. Observe that the total weight over all vertices in tree T cannot exceed n, the number of vertices in the tree. We also need to identify two special spine vertices for each subtree T_x,

$$x_L = \max\{v_i \in L_x\}, \quad \text{(reference towards leaf)}$$
$$x_R = \min\{v_i \in L_x\}, \quad \text{(reference towards root)},$$

where max and min are taken relative to the total order relation '$\preceq$' (see Figure 2.3). Then, given a search tree node x with children y_1 and y_2 where L_{y_1} precedes L_{y_2} on the spine, we insure that

$$\Big| \sum_{v \in L_{y_1}} \lambda(v) - \sum_{v \in L_{y_2}} \lambda(v) \Big| \text{ is minimized.} \tag{2.1}$$

To impose a structure typical to search trees, we assign node x a key with value y_{1L}. In this way, all leaves stored below child y_1 precede the key of node x. For us, key assignment is not important since we traverse the SD mostly bottom-up, and when we do start from the root of the SD we do not generally search for a particular vertex but visit all vertices in the input tree. A typical usage scenario is illustrated by Program 2.1 with the help of Figure 2.4.

We conclude this section with a brief presentation of other tree decompositions that have been proposed in the literature. Some of them are equally suitable for our algorithms. Our predilection for the SD is partly motivated by the simplicity of the algorithms that construct and use the structure, and it is partly a matter of personal preference. The proofs on recursive depth and storage space are given in Section 2.2, and an algorithm for constructing the SD is presented and analyzed in Section 2.3

- Let current SD search tree node $x \leftarrow v$, where v is some vertex of T, and current search tree $S \leftarrow S_\Pi$ where $v \in S_\Pi$.
- repeat until x reaches s_{SD}:
  - Process information at x.
  - If x has reached the root of the current search tree $x = s_\Pi$ then,
    - Denote by Π' the parent spine of Π, *i.e.* T_{s_Π} is a component of spine Π'.
    - Let $v'_i \in \Pi'$ be a spine vertex of parent spine Π' such that given edge $u'_i v'_i \notin \Pi'$, we have $T(u'_i) = T_{s_\Pi}$.
    - Assign to current search tree $S \leftarrow S_{\Pi'}$, and to current search tree node $x \leftarrow v'_i$.

    otherwise
    - Assign x to the parent of x in current search tree S

Program 2.1: A typical SD traversal

2.1.2 Other tree decompositions used in the literature

Several tree decompositions were proposed to overcome the shortcomings of the centroid decomposition. A tree decomposition very similar to our SD was proposed independently by Boland [18] for solving the circular ray shooting problem. The input to the problem is a simple polygon and a circular arc with one endpoint inside, and the goal is to find the first point of intersection of the arc with the boundary of the polygon. The ray shooting query is efficiently solved after processing the polygon into a special decomposition and constructing a set of weighted binary trees on it. Apart from the above mentioned computational geometry problem, Boland notes that the data structure is general and can be applied to many problems outside the computational geometric domain.

Another decomposition based on identifying paths in the input tree was produced by Cole *et al.* [29]. There, paths start from the root of the tree and end at a leaf in the same way as our spines do, except that (i) a tree vertex is added to the path if it has the most number of successors (and not *leaf* successors) among its siblings, and (ii) the components remaining after the path is removed from the tree are *not* further processed with weighted binary search trees. The data structure was named centroid path decomposition.

Holm *et al.* [61, 3, 2] take a different approach in surmounting the problems of the

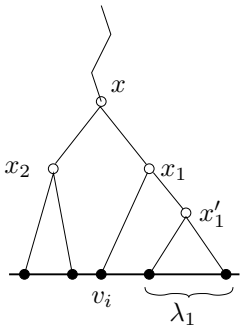


Figure 2.5: Proof of height bound in the spine decomposition

centroid decomposition. Their data structure called top-trees encodes the input tree using a hierarchy of coupling operations that connect two subtrees into one by joining them at two vertices or less. Components at the lowest level in this hierarchy are the edges on the input tree.

2.2 Properties of the SD

In this section, we are concerned to bound the length of the longest path through one or more search trees which starts at the root of the SD, s_{SD}, and ends at a tree vertex. This length is the depth, sometimes also called height, of the tree decomposition. We also establish that the storage space complexity of the SD structure alone is linear in n. Of course, when we use the SD in our algorithms, we associate data with each node of the SD and this affects the storage space complexity for those algorithms.

Recall that the path through one or several binary search trees of the spine decomposition is denoted $\sigma(x, y)$, where x and y are the beginning respectively the end of the path and can be search tree nodes or tree vertices of T. For example, the set of search tree nodes and tree vertices visited by the bottom-up traversal illustrated by Program 2.1 is denoted $\sigma(v, s_{SD})$.

Based on two facts, (a) the way a root to leaf path is chosen to form a spine (see Conditions 1 and 2 on page 24) and (b) the way binary search trees are balanced (see Relation (2.1)), we show that the number of nodes and vertices on any SD path $\sigma(s_{SD}, v)$ for some vertex $v \in T$ is $O(\log n)$. If super-nodes are implemented, the bound becomes $O(\log l)$ where l is the number of leaves of T.

Consider spine $\Pi = \pi(v_0, v_m)$ with associated search tree S_Π. Recall that $\lambda(v_i)$ is the weight of vertex v_i used in balancing search tree S_Π. We extend this weight on the internal nodes of S_Π by letting $\lambda(x) = \sum_{v \in L_x} \lambda(v)$. For simplicity, we also denote by $\lambda_{tot} = \lambda(s_\Pi)$, the total weight of spine Π. We can locally bound the search tree distance between the root of the search tree and node x, *i.e.* the number of nodes in $\sigma(s_\Pi, x)$. Let $depth(x)$ denote this distance. For the proof, we consider another strategy to balance the search tree, a strategy that generates search trees with larger height but that is easier to analyse. This strategy is the following.

Consider Figure 2.5 for an illustration. Assume node x has children x_1 and x_2 and *w.l.o.g.* $\lambda(x_1) \geq \lambda(x_2)$. Assume also that x_1 and x_2 define an optimal (balanced) partition of the weight λ under x. Let $v_i \in L_{x_1}$ be the spine vertex adjacent to L_{x_2}. Instead of recursively partitioning the spine vertices below x_1, we connect v_i directly to x_1, as in the figure and create an additional node x'_1 that becomes the other child of x_1. The structure of the search tree below nodes x_2 and x'_1 is then determined recursively. We are interested in the depth of any search tree node whose children are determined recursively.

Lemma 2.1. *The depth of node $y \in \{x'_1, x_2\}$ satisfies the following inequality,*

$$depth(y) \leq c \log \frac{\lambda_{tot}}{\lambda(y)}, \text{ where } c \geq 2 \text{ is constant and } \log \text{ is to the base } 2. \tag{2.2}$$

Proof. We assumed that leaves from L_{x_2} and L_{x_1} are partitioned in a balanced way, *i.e.*

$$\lambda(x_1) - \lambda(x_2) \text{ is minimized.}$$

$$\begin{aligned} &\Rightarrow \lambda(x'_1) \leq \frac{\lambda(x)}{2} \\ &\Rightarrow \frac{\lambda(x)}{\lambda(x'_1)} \geq 2. \end{aligned} \tag{2.3}$$

Similarly,

$$\Rightarrow \frac{\lambda(x)}{\lambda(x_2)} \geq 2.$$

We now prove the lemma by induction. We focus only on the depth of node x'_1 since the depth of x_2 is smaller and we are aiming for an upper bound on the depth. We can write,

$$depth(s_\Pi) = c \log \frac{\lambda_{tot}}{\lambda_{tot}} = 0,$$

and Relation (2.2) is satisfied. Now assume the relation is satisfied by node x and we need to prove it for x_1'. We have the following,

$$\begin{aligned}
depth(x_1') &= depth(x) + 2 \leq \\
&\leq c \log \frac{\lambda_{tot}}{\lambda(x)} + 2 \leq \\
&\leq \log \left(\frac{\lambda_{tot}}{\lambda(x)} \right)^c + \log \left(\frac{\lambda(x)}{\lambda(x_1')} \right)^2, \quad \text{by (2.3)} \\
&= \log \left(\left(\frac{\lambda_{tot}}{\lambda(x)} \right)^{c-2} \left(\frac{\lambda_{tot}}{\lambda(x_1')} \right)^2 \right) \leq \\
&\leq \log \left(\left(\frac{\lambda_{tot}}{\lambda(x_1')} \right)^{c-2} \left(\frac{\lambda_{tot}}{\lambda(x_1')} \right)^2 \right) = \\
&= c \log \frac{\lambda_{tot}}{\lambda(x_1')}.
\end{aligned}$$

□

Theorem 2.1. *The height of the spine decomposition,* i.e. *the length of the search tree path between the root of the decomposition and any vertex $v \in T$, $\sigma(s_{SD}, v)$, is $O(\log n)$. The constant from notation O is at most* 4.

Proof. Consider a path $\sigma(s_{SD}, v)$ for some $v \in T$ and let S_1, S_2, ... S_t be the different search trees visited along the path starting from s_{SD}. From Condition 2 on page 24 it follows immediately that the number of search trees t is at most $\log l$ where l represents the number of leaves of T. If we consider now Figure 2.5 again, we notice that any path $\sigma(s_{SD}, v)$ contains at most $t \leq \log l$ groups of consecutive nodes of type x, x_1, v_i. All the other nodes from the path are of type x_2 or x_1'. We can estimate the total length of path $\sigma(s_{SD}, v)$ by separately counting the number of nodes in the sequence x, x_1, v_i and x_2 or x_1'.

– For the sequence of x_2 or x_1': Denote by $s_1 = s_{SD}$, s_2, ... s_t the root nodes of search trees S_1, S_2, ... S_t, and by v_{i_1}, v_{i_2}, ... $v_{i_t} = v$ the leaves of the same search trees with the property that v_{i_j} is adjacent to the spine component used for the spine of search tree S_{j+1}. The number of nodes of type x_2 or x_1' in search tree S_j is equal to the depth of leaf v_{i_j} in search tree S_j plus one (the root of S_j must also be counted). Then, the total

number of nodes of type x_2 or x_1' in path $\sigma(s_{SD}, v)$ is

$$\begin{aligned} H &= \sum_{j=1}^{t} \big(depth(v_{i_j}) + 1\big) \leq \\ &\leq c\Big(\log \frac{\lambda(s_{SD})}{\lambda(s_2)} + \log \frac{\lambda(s_2)}{\lambda(s_3)} + \ldots \log \lambda(s_t)\Big) + t, \quad \text{(from Lemma 2.1)} \\ &= c \log \lambda(s_{SD}) + t \leq \\ &\leq c \log n + \log l. \end{aligned}$$

In the last expression above, we used the inequality $\lambda(s_{SD}) \leq n$. The inequality follows immediately if one accounts first for the weight assigned on all vertices on the first spine that are adjacent to spine components. This total weight cannot be larger than the number of vertices with degree one in T. The remaining spine vertices receive a weight of one, but the total number of such vertices is not larger than the total number of tree vertices with degree two in T, and therefore, one can conclude that

$$H \in O(\log n).$$

– For the sequence x, x_1, v_i: There are at most t additional groups of nodes x_1 and v_i totalling no more that $2\,t$ nodes.

Since $t \leq \log l \leq \log n$, by adding the number of nodes on path $\sigma(s_{SD}, v)$ in the two cases established above, it follows that the length of the path is $O(\log n)$. □

It should be pointed out here that the bounds presented above do not consider the case where super-nodes are used. If two or more vertices of degree two in T are collapsed into super-nodes, then the bound on $\lambda(s_{SD})$ in the proof of Theorem 2.1 becomes

$$\lambda(s_{SD}) \leq 2 \log l,$$

and the following result can be directly inferred.

Corollary 2.1. *The height of the spine decomposition of a tree T when super-nodes are constructed is $O(\log l)$ where l represents the number of leaves of T.*

To estimate the storage complexity of the spine decomposition, we need to bound the total number of search tree vertices. Since all search trees are full binary trees, *i.e.* any node in such tree has either degree one (a leaf) or degree three except for the root which

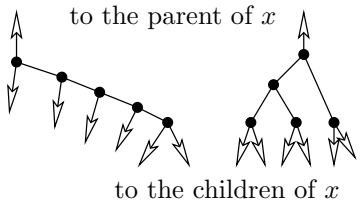


Figure 2.6: Making a binary tree from an arbitrary rooted one

has degree two, the total size of a search tree is twice the number of its leaves. The set of leaves of any search tree is the set of spine vertices and any spine vertex is leaf in only one search tree. From these observations, we can directly state the following theorem.

Theorem 2.2. *The storage space complexity of the SD is $O(n)$ whether or not super-nodes are used.*

2.3 Computation of the SD

In this section we describe and analyze a simple algorithm to construct the spine decomposition of a tree. The steps involved follow the ideas described in the previous sections.

First, if the input tree is not rooted or binary, we chose an arbitrary vertex as root and insert additional vertices and edges to make the tree binary. It is obvious how to transform an arbitrary rooted tree into a binary one. Tamir [99] describes rigorously such a process. Intuitively, one needs to replace every tree vertex x with more than two children by a binary tree of a certain configuration (Figure 2.6) that connects the parent of x with all its children. Notice that this step takes time linear in the size of the original tree.

The next step involves the construction of spines. For this, we require to know the number of leaf descendants for every vertex in the tree. This can be easily obtained in linear time by traversing the tree in post-order. Once the number of leaves $N_l(v)$ is recorded at all vertices v, spines can be selected greedily as described in Section 2.1.

The final phase concerns building the balanced binary search trees on top of each of the spines computed earlier. In order to obtain an efficient final algorithm, we must execute this phase more carefully. First, we need to traverse all spines and compute weights $\lambda(v)$ for all vertices in tree T. The weights are used in balancing the search trees and can be easily computed in total linear time once $N_l(v)$ and the spine configurations are known. Consider now spine $\Pi = \pi(v_0, v_m)$. We can use the technique of Hershberger and Suri [57]

to construct the search tree for this spine in time linear in the number of spine vertices. Then it follows that the overall algorithm for all spines is also linear because spines are disjoint. The algorithm of Hershberger and Suri is described in the following paragraphs.

We start building search tree S_Π from root to leaf. The process is recursive. Assuming that node x is already in the search tree, we want to construct its two children, x_1 and x_2. We use two pointers to visit Π, one starting from x_R and moving towards x_L, the other starting from x_L and moving in the opposite direction. The goal is to find the right partition of the spine vertices in L_x into sets L_{x_1} and L_{x_2} that eventually will represent vertices x_1 and x_2. Recall that L_x is the set of spine vertices that have x as common ancestor in search tree S_Π.

Consider that node x has pointers to spine vertices x_L and x_R, as well as the value of weight $\lambda(x)$. In a pre-processing step we have also computed for all $v \in \Pi$ the prefix sums

$$\lambda(v_0, v) = \sum_{v_0 \preceq u \preceq v} \lambda(u),$$
$$\lambda(v, v_m) = \sum_{v \preceq u \preceq v_m} \lambda(u),$$

such that it is easy to compute for $v \in L_x$

$$\lambda(x_R, v) = \sum_{x_R \preceq u \preceq v} \lambda(u) = \lambda(v_0, v) - \lambda(v_0, x_R) + \lambda(x_R), \text{ and}$$
$$\lambda(v, x_L) = \sum_{v \preceq u \preceq x_L} \lambda(u) = \lambda(v, v_m) - \lambda(x_L, v_m) + \lambda(x_L).$$

To minimize $|\lambda(x_1) - \lambda(x_2)|$, we need to find spine vertex v_i for which $\lambda(x_R, v_i) \geq \frac{\lambda(x)}{2}$ and $\lambda(v_i, x_L) \geq \frac{\lambda(x)}{2}$, basically vertex v_i from Figure 2.5. Once v_i is identified, it is easy to decide whether $v_i \in L_{x_1}$ or $v_i \in L_{x_2}$ and thus we can completely determine nodes x_1 and x_2 as children of x.

Spine vertex v_i is found via unbounded binary search from both ends of L_x simultaneously. For this purpose, spine vertices v_i need to be stored into an array such that we can have random access to its entries. The search starts at one of the endpoints of L_x and checks prefix sums at distances 0, 1, 2, 2^2, ... 2^i, ... away from the endpoint until the prefix sum surpasses the value $\frac{\lambda(x)}{2}$. If this occurs the first time for entry at distance 2^j from one of the endpoints, v_i is then found by regular binary search in the interval defined by distances 2^{j-1} and 2^j. The process is illustrated by Program 2.2. Since we perform the

---

***make_tree*(x, L_x)**

- Let $q = |L_x|$ and $V[0 \ldots q-1]$ be the array to store the vertices in L_x.
- Initialize iteration counter $I \leftarrow 0$.
- Increment I as long as $\lambda\big(V[0], V[2^I]\big)$ AND $\lambda\big(V[q-1-2^I], V[q-1]\big)$ are smaller than $\frac{\lambda(x)}{2}$.
- Find partition vertex $v_i \in L_x$ through binary search in the appropriate interval defined by 2^{I-1} and 2^I.
- Determine the optimal L_{x_1} and L_{x_2}; construct nodes x_1 and x_2
- *make_tree*(x_1, L_{x_1}).
- *make_tree*(x_2, L_{x_2}).
- Assign x_1 and x_2 as children of x.

Program 2.2: Recursive procedure to construct a balanced binary search tree over a given spine

search simultaneously from both ends and we stop immediately after v_i is identified, the recurrence relation for $T(|L_x|)$ which represents the running time of the procedure above is

$$T(q) = T(i) + T(q-i) + \min\big\{\log i, \log(q-i)\big\}. \tag{2.4}$$

Computing the search trees for every spine is the last step in the algorithm to construct the spine decomposition of a tree. The entire algorithm is illustrated in Program 2.3. Its complexity is established by the following theorem.

Theorem 2.3. *The algorithm described by Program 2.3 constructs the spine decomposition $SD(T)$ of any input tree T in time linear in the size of T.*

Proof. It is easy to observe that the overall running time and storage space used at all phases of Program 2.3 is linear in n, except perhaps for computing the search trees. To show that this phase is also linear, we need to bound $T(q)$ from Relation (2.4). We can use standard techniques to prove $T(q) \in O(q)$, see for example Cormen *et al.* [31].

We want to show that

$$T(x) \leq cx - d\log x, \tag{2.5}$$

where $x \geq x_0$ for a sufficiently large constant x_0, c and d are constants, and the logarithm is to the base 2. Assume (2.5) is true for all $x < q$, and consider *w.l.o.g.* that $i \leq q - i$. By

- Root T at a vertex, if not rooted; make T binary, if not binary.
- Traverse T in post-order and compute the number of descendant leaves $N_l(v)$ for all $v \in T$.
- Construct the first spine; recurse on each of the spine components.
- Assign the weights $\lambda(v)$ for all $v \in T$ by traversing every spine.
- For all spines Π do:
  - traverse Π and compute prefix sums.
  - $make\_tree(s_\Pi, \Pi)$.

Program 2.3: Construction algorithm for the spine decomposition $SD(T)$

substitution into (2.4), we obtain the following inequality,

$$\begin{aligned} T(q) &\leq ci + c(q-i) - d\big(\log i + \log(q-i)\big) + \log i = \\ &= cq - d\big(\log i + \log(q-i)\big) + \log i. \end{aligned}$$

To prove that (2.5) is satisfied when $x = q$, it is enough to show now that

$$cq - d\big(\log i + \log(q-i)\big) + \log i \leq cq - d \log q,$$

which, after arranging terms and simplifications, is equivalent to

$$\begin{aligned} d\big(\log i + \log(q-i) - \log q\big) &\geq \log i \\ \Leftrightarrow \left(\frac{i(q-i)}{q}\right)^d &\geq i. \end{aligned} \tag{2.6}$$

From our assumptions, we know that $q - i \geq \frac{q}{2}$, so by substituting $q - i$ into (2.6) we now need to show the following inequality,

$$\left(\frac{i}{2}\right)^d \geq i. \tag{2.7}$$

But (2.7) is satisfied for all $i \geq 3$ if $d \geq 3$, and the induction hypothesis is also satisfied. Now we can chose constant x_0 and c sufficiently large so that the boundary condition $x = x_0$ in (2.5) is also satisfied, and the theorem is proved. □

2.4 Conclusion

This chapter presents a data structure called the spine decomposition (SD) that is used in processing arbitrary trees into structures that mimic the properties of balanced binary trees. In this way, many algorithms that are efficient on balanced trees can be generalized for arbitrary trees. Other tree decompositions exist in the literature. Among them the centroid decomposition is the most popular, however we discuss at the beginning of this chapter that the centroid decomposition leads to more complicated algorithms for many of the problems studied in this thesis.

Another advantage of the SD over the centroid decomposition is that the SD admits a simple linear time construction algorithm, whereas the linear time algorithms for the centroid decomposition are involved. For the linear time algorithms that are based on the centroid decomposition, it might be better to use the SD instead. Finally, the height of the SD is proportional to $\log l$, where l is the number of leaves in the input tree, and for certain type of trees, this number can be significantly less than the total number of vertices in the tree. For some optimization problems, using the SD instead of the centroid decomposition can automatically give algorithms with complexity depending on the number of leaves in the tree instead of the number of vertices [9]. There are other problems though where, in order to fully exploit the versatility of the SD, one has to compromise on simplicity. For example, the 2-median problem in trees can be easily solved in $O(n \log n)$ time using the SD. However, it is possible to achieve a running time of $O(n \log l)$, but several passes through the decomposition have to be made and a lot more information needs to be pre-processed. It is debatable whether the added degree of intricacy justifies the gain in efficiency. Nevertheless, the role of the SD for many of the algorithms presented in this thesis, and especially for the solution to the k-median problem in trees discussed in the next chapter, is decisive.

Chapter 3

The k-median problem in trees: algorithm UKM

In this chapter, we describe an undiscretized dynamic programming algorithm for the k-median problem in trees with a running time sub-quadratic in n. We name this algorithm UKM from *Undiscretized K-Median*. The general approach of the algorithm, as well as a brief literature review and the motivation for our research were already discussed in Section 1.3. Since our approach is based on dynamic programming, we start by defining the cost functions used. All cost functions are associated with nodes of the spine decomposition (SD) and are computed recursively from nodes at the lower level.

3.1 The dynamic programming cost functions

Let x be an internal node of the search tree of the spine decomposition and let $\Pi = \pi(v_0, v_m) = (v_0, v_1, \dots v_m)$ be the corresponding spine. Let also i_R and i_L be the indices for spine nodes x_R respectively x_L. An illustration for these nodes is given by Figure 2.3. Denote by $T(x_R)$ the tree component containing x_R obtained after removing edge $e = x_R v_{i_R - 1}$ from T. Since edge e connects x_R to its parent in T, $T(x_R)$ is the subtree induced by all successors of x_R in T.

Our cost functions play roles similar to those of the classic dynamic programming algorithm of Tamir [99] reviewed in Section 1.4.1. They return partial costs of the global solution that only considers vertices from subtrees of the input tree. The subtrees are components

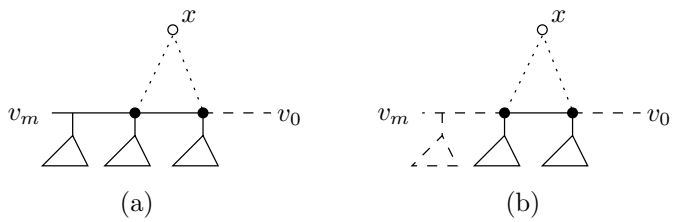


Figure 3.1: Subtrees of the input tree for which cost functions are defined; (a) big trees (b) small trees

of the SD.

As in the algorithm of Tamir, we define two classes of cost functions. One class consists of discrete functions for which one argument is a tree vertex representing the median that covers a reference vertex of the subtree. The median is internal to the subtree, and therefore this set of functions is similar to $G()$. The other class is formed by continuous functions for which one parameter is a real value representing the distance from a median outside the subtree that covers the reference vertex of the subtree. These functions are similar to functions $F()$.

We further distinguish the functions by the type of subtree whose cost they return. We have a choice of two different subtrees to associate with the function at a given node x of the SD: (i) subtree T_x (informally, T_x is referred as the small subtree), and (ii) subtree $T(x_R)$ (informally referred as the big subtree). Recall that T_x is a component of the SD and consists of all the tree vertices whose path in the decomposition to root s_{SD} contains x, and $T(x_R)$ is the subtree of T rooted at x_R. Figure 3.1 displays the two subtrees associated with a given SD node x.

A third feature of our cost functions concerns the restrictions imposed on the location of split edges inside the subtrees. Split edges are introduced in Section 1.4.1. They represent a set of edges whose removal defines k connected components such that the total cost of the 1-median of these components is equal to the optimal k-median cost of the entire tree. More precisely, two types of cost functions are defined, (a) for which the choice of split edges is constrained to avoid the spine edges of T_x or $T(x_R)$, and (b) for which split edges are unconstrained. We chose to restrict the placement of split edges because this gives us more control on the recursive computation of the cost functions and simplifies the equations. Intuitively, for the computation of cost functions of the type $F()$ mentioned earlier, which account for an outside facility serving a reference vertex in the subtree, the presence of a

split edge on the spine separates a significant part of the subtree from the external facility. This introduces an asymmetry in the recursive computation that is difficult to handle. To avoid this, we choose to consider spine edges as candidates for splitting, separately.

Based on these observations, we decide to use letter codes to name each cost function. The first letter (O) or (I) specifies whether the median covering the reference vertex is outside or inside the subtree. The second letter (S) or (B) shows whether the cost function is defined for T_x (small subtree) or $T(x_R)$ (big subtree). Finally, the third letter (U) or (C) marks whether the choice of split edges is unconstrained or constrained to avoid the spine. It is also important to be able to unambiguously identify the reference vertex for each of the subtrees T_x or $T(x_R)$. We already discussed at the beginning of Chapter 2 about reference vertices and argued that the centroid decomposition is not suitable because $O(\log n)$ different functions need to be maintained, one for each reference vertex that might occur. With the spine decomposition, there are at most two reference vertices for a given SD node x, x_R and x_L. For T_x for example, two different functions of the same class might be needed, one where the reference vertex is x_R, the other using x_L as reference. To specify which of these two functions are referred to, we add subscript (R) or (L) to the name of all cost functions, even when such a distinction is not needed because it can be inferred by other means. The cost functions can be defined as follows.

1. IBU = Inside Big Unconstrained.

   $IBU_R\big(x, j, z\big)$ returns the optimal cost of $T(x_R)$ if $j+1$ facilities – or j split edges – are selected in $T(x_R)$ possibly on the spine, and the facility covering x_R is vertex z chosen from T_x. Note the asymmetry between where z is selected from (T_x) and the tree for which cost is returned ($T(x_R)$). The cost of the optimal k-median solution for T can be retrieved from

   $$\min_{z \in T} IBU_R\big(s_{SD}, k-1, z\big).$$

   Hence, our goal is to evaluate $IBU_R\big()$ at the SD root node s_{SD}.

2. OSC = Outside Small Constrained.

   $OSC_R\big(x, j, \alpha\big)$ returns the optimal cost of T_x if j split edges are chosen from T_x, none of them on the spine, and the closest external median is at distance α from x_R and covers x_R. Function $OSC_L\big(x, j, \alpha\big)$ is the same except that the external median is at distance α from x_L, and covers x_L.

3. OBU = Outside Big Unconstrained.

   Function $OBU_R\big(x, j, \alpha\big)$ returns the cost of subtree $T(x_R)$ when there is no spine edge restriction for split edges. If we wanted to make the definition of $OBU_R\big()$ parallel with that of $OSC_R\big()$, then we would force all split edges to lie in T_x. However, if split edge e is on the spine, there is no point in continuing to force the remaining split edges to lie in T_x. The contribution of vertices separated from the external median by e is returned by the optimal cost of the p-median (for some $p \leq j$) on the component obtained by removing e. Thus, we do not care where in the separated component the $p-1$ split edges are. Formally, let $C_{opt}(T, j)$ denote the optimal j-median of some tree T. Let i_R and i_L be the indices of the spine vertices that correspond to x_R and x_L. We consider all edges $v_{i-1}v_i$ on the spine for $i_R + 1 \leq i \leq i_L + 1$ and we denote by $T(v_i)$ the component obtained by removing edge $v_{i-1}v_i$. We define now the unrestricted cost function $OBU_R\big(x, j, \alpha\big)$ as taking the best choice between splitting a spine edge from $\pi(x_R, x_L)$, or not splitting it. The latter choice simply uses the value of the restricted cost function $OSC_R\big(x, j, \alpha\big)$ (see Expression 3.2),

$$OBU_R\big(x, j, \alpha\big) = \min \Bigg\{ OBC_R\big(x, j, \alpha\big), \min_{i_R+1 \leq i \leq i_L+1} \Big\{ \min_{0 \leq q \leq j-1} \big\{ OSC_R\big(T(x_R) \setminus T(v_i), j-1-q, \alpha\big) + C_{opt}(T(v_i), q+1) \big\} \Big\} \Bigg\}. \tag{3.1}$$

   Above we abuse the notation and denote by $OSC_R\big(T(x_R) \setminus T(v_i), p, \alpha\big)$ the value of function $OSC_R\big()$ in tree $T(x_R) \setminus T(v_i)$. Observe that tree $T(x_R) \setminus T(v_i)$ does not generally correspond to a particular SD node. In addition, we use in the expression a function we did not yet define, $OBC_R\big()$. From the name convention however, it should be obvious that $OBC_R\big()$ returns the cost in $T(x_R)$ if j split edges are in T_x but not on the spine, and an external median is at distance α from x_R and covers x_R,

$$OBC_R\big(x, j, \alpha\big) = OSC_R\big(x, j, \alpha\big) + \sum_{v \in T(x_R) \setminus T_x} w(v)\big(d(v, x_R) + \alpha\big). \tag{3.2}$$

Properties of cost functions

Before we describe the computation of the cost functions, notice that the continuous functions are piece-wise linear and concave. Indeed, consider a fixed set of j split edges in tree

- Construct the SD of tree T.
- Traverse the SD botom up. For each SD node x do
  - if x is leaf, compute trivial value for $IBU_R()$, $OBU_R()$, $OSC_R()$, and $OSC_L()$ at x, otherwise use the functions stored at the two children of x to obtain the functions at x.
  - store the cost functions computed at node x.
- For all $z \in T$, retrieve $IBU_R(s_{SD}, k-1, z)$ and retain the minimum. Return the minimum as solution.

Program 3.1: Main steps of the dynamic programming algorithm for solving the k-median problem in trees

T_x. The value of function $OBU_R(x, j, \alpha)$ is determined by the 1-median costs of the j components plus the cost of the $(j+1)^{\text{th}}$ component served by the external median. The later term is an expression linear in α, while the former terms are constant for a fixed set of split edges. By definition, $OBU_R(x, j, \alpha)$ is the minimum over the linear cost functions for all possible sets of j split edges, and thus it is piece-wise linear and concave in α. Exactly the same argument can be made for functions $OSC_L()$ and $OSC_R()$.

The main dynamic programming algorithm

Cost function $IBU_R()$ is the only function used directly in the retrieval of the optimal k-median solution in tree T. All other functions defined above just support the computation of $IBU_R()$. The main algorithm for the k-median problem in trees is simple. After the construction of the SD, the nodes of the decomposition are populated bottom up with the values of cost functions $OBU_R()$, $OSC_L()$, $OSC_R()$, and $IBU_R()$. The optimal solution is obtained finally as the minimum value of function $IBU_R()$. The process is outlined by Program 3.1.

3.2 Computation of the cost functions

The actual dynamic programming algorithm that solves the k-median problem in trees is straightforward, as shown the the previous section. What is important for us is to establish the recurrence relations necessary in the computation of all cost functions defined earlier.

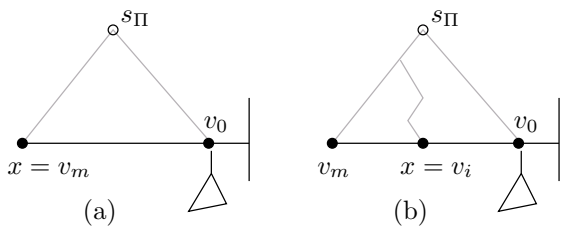


Figure 3.2: (a) A spine vertex of degree one (a leaf of T), and (b) a spine vertex of degree two

Each function is associated with a particular SD node, but not all nodes in the decomposition have two children. In the next paragraphs, we give the recursive formulae for each function in three different circumstances: when the current SD node is a spine vertex of degree at most two in T, when it is a spine vertex of degree three in T, and when it is an SD node that is parent of two other nodes in some binary search tree. In all three sections to follow, we denote by x the SD node for which the respective cost function is defined.

3.2.1 Calculating functions $OSC_R()$ and $OSC_L()$

Restricted cost functions $OSC_L()$ and $OSC_R()$ return, as specified at the beginning of Section 3.1, the optimal cost in subtree T_x when j split edges are chosen in T_x but not on the spine, and the reference vertex x_L or x_R is served by the outside median. These functions have the simplest defining recursive formulae.

Node x is a spine vertex of degree at most two

Consider Figure 3.2 as reference. In both cases, x corresponds to a trivial component of the spine decomposition, leaf vertex v_m in case (a), respectively tree vertex v_i in case (b). Of course, on a component with exactly one vertex, it is quite easy to define cost functions $OSC_L()$ and $OSC_R()$ since no extra facilities can be located inside. The single vertex must be served by the outside median, and thus,

$$\begin{aligned} &OSC_L(x,0,\alpha) = OSC_R(x,0,\alpha) = w(x)\,\alpha, \\ &OSC_L(x,j,\alpha), OSC_R(x,j,\alpha) \text{ – undefined for } j \geq 1. \end{aligned} \tag{3.3}$$

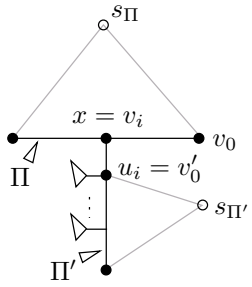


Figure 3.3: A spine vertex of degree three

Node x is a spine vertex of degree three

The situation is illustrated in Figure 3.3 and is no longer trivial. Node x corresponds to spine vertex v_i which in turn is adjacent to subtree $T(u_i) = T_{s_{\Pi'}}$. As before, $T(u_i)$ denotes the component of tree T that contains vertex u_i and that is acquired by removing edge u_iv_i. Π is the spine containing x, and Π' is the spine obtained recursively from $T(u_i)$.

Split edges, if any, can now be chosen within $T(u_i)$ or edge v_iu_i itself can be split. Note that split edges inside $T(u_i)$ can be located on spine Π', and therefore function $OBU_R()$ at node $s_{\Pi'}$ should be used to account for the contribution of vertices in $T(u_i)$. Our recursive formula, should insure that both the contribution of vertex v_i, as well as the possibility to split edge v_iu_i are not left out. We can write,

$$OSC_L(x, j, \alpha) = OSC_R(x, j, \alpha) = \min \Big\{ OBU_R(s_{\Pi'},\ j,\ \alpha + l(v_iu_i)) + w(v_i)\,\alpha, \\ C_{opt}(T_{s_{\Pi'}},\ j) + w(v_i)\,\alpha \Big\}. \quad (3.4)$$

The first part of the equation considers that edge v_iu_i is not split, and as consequence it simply adds the contribution of subtree $T(u_i)$, pre-computed and stored at SD node $s_{\Pi'}$, with the contribution of vertex v_i. The second part views edge v_iu_i split, and thus the remaining $j - 1$ edges form an optimal j-median in $T(u_i)$. Of course, for the second part $j \geq 1$. If $j = 0$, only the first part is present in the formula.

Node x is an internal search tree node

Let t and y be the two children of node x in search tree S_Π such that the spine vertices below t are towards the root, as shown by Figure 3.4. The computation of the cost function

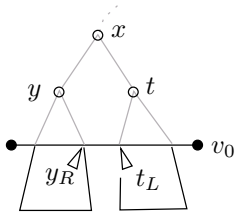


Figure 3.4: An internal search tree node

at x is quite simple in this case. A total of j split edges must be placed in T_x, which means that an appropriate partition of the j split edges between subtrees T_t and T_y has to be computed. Formally,

$$\begin{aligned} OSC_R(x,j,\alpha) &= \min_{0\le q\le j}\Big\{OSC_R(t,q,\alpha) + OSC_R(y,\ j-q,\ \alpha + d(t_R,y_R))\Big\},\\ OSC_L(x,j,\alpha) &= \min_{0\le q\le j}\Big\{OSC_L(y,q,\alpha) + OSC_L(t,\ j-q,\ \alpha + d(y_L,t_L))\Big\}. \end{aligned} \tag{3.5}$$

3.2.2 Calculating function $OBU_R()$

Unrestricted cost function $OBU_R()$ returns the cost of big subtree $T(x_R)$ if the outside median serves reference vertex x_R. The split edges may be chosen from the spine. From the defining equation (3.1), we notice that the spine edges that are candidates for splitting are those with the parent from set L_x. We say that an edge $e = uv$ has parent v if $p(u) = v$, where p is the parent function introduced in Section 1.1. Set L_x is the set of spine edges (leaves of the search tree) that are below internal node x, therefore the set of spine edges that are candidate for splitting are those below node x in S_Π together with the spine edge towards the leaf.

Node x is a spine vertex of degree at most two

We consider the same situation as depicted in Figure 3.2. When there is no split edge to be located, all vertices in $T(x_R)$ are served by the external median and thus,

$$OBU_R(x,0,\alpha) = \sum_{v\in T(x_R)} w(v)\big(\alpha + d(x_R,v)\big).$$

Unlike the restricted cost function discussed in the previous section, the set of edges that can be split is not empty when node x is a spine vertex of degree two. According to the definition of the cost function (3.1), exactly one spine edge may be split. This edge has

vertex $v_i = x$ as parent. Since the split edges separated from the external median by the spine edge can lie anywhere in the separated component, cost function $OBU_R()$ is defined for any value of $j \geq 1$ as long as $j \leq k-1$ and $j \leq |T(v_{i+1})| - 1$. Thus, we have

$$OBU_R(x, j, \alpha) = \begin{cases} w(v_i)\,\alpha + C_{opt}\big(T(v_{i+1}),\ j\big), & \text{when } x = v_i \text{ and } 1 \leq j \leq |T(v_{i+1})| - 1, \\ \text{undefined} & \text{when } x = v_m \text{ and } j \geq 1. \end{cases} \tag{3.6}$$

Node x is a spine vertex of degree three

Assume we have already computed function $OSC_R(x, j, \alpha)$ using (3.4). Refer to Figure 3.3, where node x coincides with spine vertex v_i. To account for the contribution of vertices in $T(v_{i+1})$, there are two possibilities. (1) Edge $v_i v_{i+1}$ is split, in which case the optimal j-median of $T(v_{i+1})$ is required for $1 \leq j \leq |T(v_{i+1})| - 1$ as explained earlier, or (2) edge $v_i v_{i+1}$ is not split, in which case no other median resides in $T(v_{i+1})$ which is served entirely by the external median. We have,

$$OBU_R(x, j, \alpha) = \min \Bigg\{ \min_{1 \leq q \leq j} \Big\{ OSC_R(x,\ j-q,\ \alpha) + C_{opt}\big(T(v_{i+1}),\ q\big) \Big\}, \\ OSC_R(x, j, \alpha) + \sum_{v \in T(v_{i+1})} w(v)\big(\alpha + d(v, x_R)\big) \Bigg\}. \tag{3.7}$$

Node x is an internal search tree node

Consider Figure 3.4. Because the set of candidate split edges for the cost function at node x is simply the union of candidate split edges for the function at the children nodes t and y, there is no additional edge that must be explicitly checked for splitting and the computation is simpler. The j split edges must be partitioned between T_y and T_t. If it happens that one of these edges is on the spine for the child node towards the root $\pi(t_R, y_R)$, then the value of the function at x is equal to $OBU_R(t, j, \alpha)$ which is already computed. The only other situation that has to be checked is when no spine edge is split in $\pi(t_R, y_R)$. Then, function $OBU_R\big(y, j, \alpha + d(t_R, y_R)\big)$ does not include the contribution of T_t which may also contain split edges, but none of these split edges are on the spine. Thus the value of $OSC_R(t, q, \alpha)$

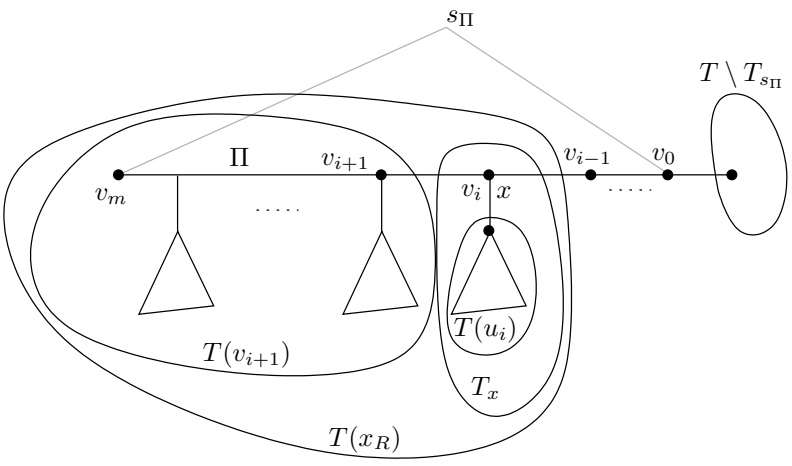


Figure 3.5: Part of a spine decomposition and the various subtrees and vertices used

can be used to add this contribution. We have,

$$OBU_R(x, j, \alpha) = \min\Big\{OBU_R(t, j, \alpha), \min_{0\le q\le j}\Big\{OSC_R(t, q, \alpha) + OBU_R(y,\ j-q,\ \alpha + d(x_R, y_R))\Big\}\Big\}. \quad (3.8)$$

3.2.3 Calculating function $IBU_R()$

Function $IBU_R()$ is the discrete cost function in our undiscretized dynamic programming algorithm. It receives as parameter a vertex z from the SD component it is defined for, and returns the optimal $(j+1)$-median cost in the larger subtree, given that z is the facility covering the reference vertex x_R of T_x.

Node x is a spine vertex

Consider figures 3.2 and 3.3 where node x is shown as a spine vertex with degree one, two, and three respectively. For reference, Figure 3.5 depicts some of the variables used in the exposition that follows. Note that $x = x_R$. Fix now a tree vertex $z \in T_x$. The set of tree vertices from $T(x_R)$ that contribute to the value of function $IBU_R()$ can be partitioned in two:

* Vertices from T_x; recall that $T_x = \{v_m\}$ if x has degree one in T, $T_x = \{v_i\}$ if x has degree two in T, and $T_x = \{v_i\} \cup T(u_i)$ if x has degree three in T.

∗ Vertices from $T(x_R) \setminus T_x$; recall that $T(x_R) \setminus T_x = \emptyset$ if x has degree one in T and $T(x_R) \setminus T_x = T(v_{i+1})$ if x has degree two and three in T.

For lack of a better notation, denote the contribution of $T(x_R) \setminus T_x$ by

$$F_{OBU}(T(v_{i+1}),\ j,\ d(z, v_{i+1})).$$

We use subscript OBU to hint that this contribution is similar in nature to that of cost function $OBU_R()$. In fact, we show in Section 3.2.4 how to compute it from cost functions $OBU_R()$, $OSC_L()$, and $OSC_R()$ stored at the nodes of the decomposition. It is important to note that F_{OBU} is not a function but a value since vertex z is fixed and $d(z, v_{i+1})$ is available. It gives the optimal cost in subtree $T(v_{i+1})$ if j split edges are placed *anywhere* in $T(v_{i+1})$ and root v_{i+1} is covered by z, the outside median[1]. The contribution of vertices from T_x is obtained recursively from function $IBU_R()$ computed for spine component $T(u_i)$ if $T(u_i)$ exists.

When $x = v_m$ is a tree leaf, then $z = v_m$ and we simply have

$$IBU_R(x, j, v_m) = \begin{cases} 0, & \text{if } j = 0 \\ \text{undefined}, & \text{if } j \geq 1. \end{cases} \tag{3.9}$$

When $x = v_i$ has degree two in T, then $z = v_i$ and $T(u_i) = \emptyset$, therefore we directly have

$$IBU_R(x, j, v_i) = F_{OBU}(T(v_i),\ j,\ 0).$$

When $x = v_i$ is a spine vertex of degree three in T, we have more complicated recurrence relations. As shown in Figure 3.3, Π' is the spine obtained in $T(u_i)$ whose binary search tree has the root denoted $s_{\Pi'}$. For $z \in T(u_i)$,

$$\begin{aligned} IBU_R(x, j, z) = \min \Bigg\{ & \min_{0 \leq q \leq j} \Big\{ IBU_R(s_{\Pi'}, q, z) + \\ & + F_{OBU}(T(v_{i+1}),\ j - q,\ d(z, v_{i+1})) + w(v_i)\, d(v_i, z) \Big\}, \\ & \min_{0 \leq q \leq j-1} \Big\{ IBU_R(s_{\Pi'}, q, z) + C_{opt}(T(v_{i+1}),\ j - q) + w(v_i)\, d(v_i, z) \Big\} \Bigg\}, \end{aligned} \tag{3.10}$$

[1]The assumption that z covers v_{i+1} does not embrace all possible cases for the computation of $IBU_R()$. For now, we ignore that edge $v_i v_{i+1}$ might be split too, because we want to focus only on introducing F_{OBU}.

and for $z = v_i$,

$$IBU_R\big(x, j, v_i\big) = \min\Bigg\{ \min_{0\le q\le j} \Big\{ OSC_R\big(x, q, 0\big) + F_{OBU}\big(T(v_{i+1}),\ j-q,\ d(z, v_{i+1})\big)\Big\}, \\ \min_{0\le q\le j-1} \Big\{ OSC_R\big(x, q, 0\big) + C_{opt}\big(T(v_{i+1}),\ j-q\big)\Big\}\Bigg\}. \quad (3.11)$$

The first term in both equations assumes that edge v_iv_{i+1} is not split and thus z does serve some vertices from $T(x_R) \setminus T_x$. As we do not know the optimal distribution of split edges between T_x and $T(x_R) \setminus T_x$, we need to verify all possible combinations. For the second term, edge v_iv_{i+1} is considered split, therefore the contribution of $T(x_R) \setminus T_x$ is returned by the optimum $(j-q)$-median. For the case $z = v_i$, the contribution of T_x is no longer returned by function $IBU_R()$ because $z \notin T(u_i)$. Instead, we can use $OSC_R()$ which is already computed for node x.

Node x is an internal search tree node

In this case, the computation of $IBU_R()$ is more simple. Figure 3.4 is again representative. Vertex z can be either in T_t or in T_y. If it is in T_t, because node t is towards the root, we have that $IBU_R\big(x, j, z\big) = IBU_R\big(t, j, z\big)$. If $z \in T_y$, we need to evaluate the contribution of $T(x_R) \setminus T(y_R)$ which is not accounted for in the value of $IBU_R\big(y, j, z\big)$. Clearly, according to the definition, z must cover vertex x_R, and therefore no split edges can be chosen on the spine from y_R to $x_R = t_R$. Thus, we can evaluate the contribution of the unaccounted vertices using the restricted cost function $OSC_L()$ stored at node t. Of course, we still need to verify all possible assignments of the j split edges between $T(y_R)$ and T_t. Formally,

$$IBU_R\big(x, j, z\big) = \begin{cases} IBU_R\big(t, j, z\big), & \text{if } z \in T_t \\ \min\limits_{0\le q\le j}\Big\{ IBU_R\big(y, q, z\big) + OSC_L\big(t,\ j-q,\ d(z, t_L)\big)\Big\}, & \text{if } z \in T_y. \end{cases} \quad (3.12)$$

3.2.4 Implementation of the recursive equations for the cost functions

In the previous paragraphs, we describe formulae that allow the computation of all cost functions associated with the nodes of the spine decomposition in a bottom up procedure. However, two values are employed in several places and we haven't shown yet how to compute them. One of them is $C_{opt}(T', j)$ which returns the optimal j-median on subtree T' of T. The difficulty of computing C_{opt} efficiently stems from the need to evaluate it on a linear

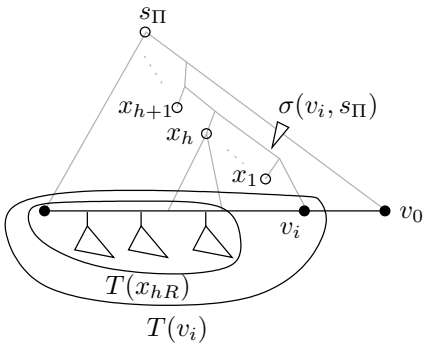


Figure 3.6: Evaluating term F_{OBU}

number of subtrees of T. If we want a sub-quadratic k-median algorithm for T, we need to evaluate C_{opt} in amortized sub-linear time per subtree. This is not a trivial task and is discussed separately in Section 3.6. We also study three special instances of the k-median problem in trees where the evaluation of C_{opt} is either not needed, or it is simpler than in the general case. These special instances are discussed in Section 3.5

In this section, we concentrate on the computation of the second term, $F_{OBU}\big(T(v_i), j, a\big)$, where a is a given distance. F_{OBU} is just a value and not a function, that is why we avoid to use the notation α as parameter. Value F_{OBU} is used in the computation of function $IBU_R\big(x = v_{i-1}, j, z\big)$ (see Section 3.2.3) and a is the distance from facility $z \in T_x$ to v_i. Consider Figure 3.6 where $\Pi = \pi(v_0, v_m)$ is the current spine and s_Π is the root of the binary search tree. Value $F_{OBU}\big(T(v_i), j, a\big)$ represents the optimal cost in subtree $T(v_i)$ if j split edges are to be located anywhere in the subtree but root v_i of the subtree is served by an external facility situated at exactly distance a from v_i.

Let x_1, x_2, ... x_b be the SD nodes adjacent to the path in the search tree from v_i to root s_Π and sitting on the leaf side of the path. In a search tree, a node x is on the leaf side of the path from some spine vertex v_i to root s_Π if the spine vertices shadowed by x are successors of v_i on the spine, otherwise x is on the root side of the path. Denote v_i by x_0. Observe that the set of vertices of $T(v_i)$ is the union of the vertices of T_{x_c} for $0 \leq c \leq b$. We will use the cost functions stored at nodes x_c to calculate the value of F_{OBU}. What we need to obtain is the optimal distribution of the j split edges in each of the components T_{x_c} and among the spine edges between consecutive components T_{x_c} and $T_{x_{c+1}}$. Observe that in the case when no split edges are chosen in the optimal solution from spine $\pi(v_i, v_m)$, then F_{OBU} is simply the summation of functions $OSC_R\big()$ stored at each node x_c. More precisely, if j_c

represents the optimal number of split edges from T_{x_c} with $\sum_{c=0}^{b} j_c = j$, then

$$F_{OBU}\big(T(v_i), j, a\big) = \sum_{c=0}^{b} OSC_R\big(x_c,\ j_c,\ a + d(v_i, x_{cR})\big).$$

However, it is possible that some spine edges are split in the optimal solution. Let h be the smallest index for which spine $\pi(x_{hR}, x_{(h+1)R})$ contains a split edge. Then, we can use cost functions $OSC_R()$ from nodes x_0, x_1, ... x_{h-1}, and $OBU_R()$ from x_h to retrieve the value for F_{OBU}.

The algorithm proposed here follows the ideas outlined above. The value of F_{OBU} can be expressed as the summation of two terms, one that considers the contribution of vertices from $\bigcup_{c=0}^{h-1} T_{x_c}$, the other from $T(x_{hR})$. Recall that $T(x_{hR})$ is the subtree obtained by removing edge $x_{hR}x_{(h-1)L}$ from T. Denote the first contribution by $F_{OSC}(h-1,\ j',\ a)$. We use subscript OSC to hint that the values come from the evaluation of cost functions $OSC_R()$ stored at nodes x_0, x_1, ... x_{h-1}. Here j' represents the total number of split edges in $\bigcup_{c=0}^{h-1} T_{x_c}$.

F_{OSC} can be obtained recursively, by dynamic programming, as follows. When $h = 1$, we have directly

$$F_{OSC}(0, j', a) = OSC_R\big(x_0, j', a\big), \quad \forall j' \le j.$$

For $1 < h \le c$, we can write,

$$F_{OSC}(h-1, j', a) = \min_{0 \le q \le j'} \Big\{ OSC_R\big(x_{h-1},\ q,\ a + d(x_{(h-1)R}, x_0)\big) + \\ + F_{OSC}(h-2,\ j'-q,\ a) \Big\},\ \forall j' \le j. \quad (3.13)$$

Finally, we obtain F_{OBU} as the summation of the two terms mentioned earlier, for all possible choices where the first spine edge may be split and for all possible distributions of the j split edges between the two parts of subtree $T(v_i)$. Formally,

$$F_{OBU}\big(T(v_i),\ j,\ a\big) = \min \Big\{ OBU_R\big(x_0, j, a\big), \\ \min_{1 \le h \le b} \Big\{ \min_{0 \le q \le j} \Big\{ F_{OSC}(h-1,\ q,\ a) + OBU_R\big(x_h,\ j-q,\ d(x_{hR}, x_0)\big) \Big\} \Big\} \Big\}. \quad (3.14)$$

Lemma 3.1. *Term $F_{OBU}\big(T(v_i),\ j,\ a\big)$ can be evaluated, for a given number of split edges j and a given distance to the external facility a, in $O(\log^2 n)$ time.*

Proof. We noticed that there are a total of $O(\log n)$ SD nodes involved in the computation of F_{OBU} for which functions $OSC_R()$ and *cst* are queried. The computation uses another term denoted F_{OSC} which is associated with the index of a given node and a given number of split edges, and is computed by dynamic programming. We count first the total time spent in computing F_{OSC}. For a fixed index and a given number of split edges, F_{OSC} is obtained by evaluating function $OSC_R()$ at most $k-1$ times (see (3.13)). The evaluation of $OSC_R()$ takes $O(\log n)$ time, which gives $O(k \log n)$ time spent for one value F_{OSC}. There are $O(k \log n)$ different values F_{OSC}, thus the total time required for all F_{OSC} values is $O(k^2 \log^2 n)$ which is $O(\log^2 n)$ for fixed k. F_{OBU} is finally obtained from (3.14) after an additional call to function $OBU_R()$, but the time for this additional call does not dominate the time for calculating F_{OSC}. Therefore, F_{OBU} can be evaluated in $O(\log^2 n)$ time. □

3.3 The complexity of cost functions

In this section we provide an upper bound on the complexity of the continuous cost functions employed in our algorithms. By the complexity of a continuous piece-wise linear function, we understand the number of linear pieces that make up the function. As argued in Section 1.4.1, this upper bound is essential for proving the running time of our algorithm for the k-median problem in trees. Any data structure that stores such a function takes space proportional to its complexity and the time consumed in computing the function in its entirety is also proportional to the complexity of the function. Note that discrete cost function $IBU_R()$ is represented by the set of its values for each of its parameters z. By definition, z is taken from the vertex set of the SD component for which the function is defined. As a consequence, a trivial bound on the storage needed for function $IBU_R()$ is the size of the SD component.

The continuous cost functions discussed here are $OBU_R()$, $OSC_R()$, and $OSC_L()$. Our proof for a bound on the size of these functions is based on two ideas, (a) that they are piece-wise linear and concave, and (b) that the recursive equations used in their computation can be used in deriving a recurrence relation on the size of the functions. It is known that the complexity of the sum or minimum of two piecewise linear concave functions is at most the sum of the complexities of the two functions. The recurrence relation is obtained from the recursive formulae by adding the sizes of any functions involved in an addition or in a minimum operation.

Consider Figure 3.4, with SD node x having children y and t. Obviously $|T_x| = |T_y|+|T_t|$. Denote by $f_j(|T_x|)$ an upper bound on the complexity of all three cost functions defined at node x, $OSC_R\big(x, j, \alpha\big)$ $OSC_L\big(x, j, \alpha\big)$, and $OBU_R\big(x, j, \alpha\big)$. In other words,

$$f_j(|T_x|) \geq |OSC_R\big(x, j, \alpha\big)| \quad f_j(|T_x|) \geq |OSC_L\big(x, j, \alpha\big)| \quad f_j(|T_x|) \geq |OBU_R\big(x, j, \alpha\big)|.$$

From (3.5) one can directly express the following recurrence relation,

$$\begin{aligned} f_j(|T_x|) &\leq \sum_{q=0}^{j}\big(f_q(|T_t|) + f_{j-q}(|T_y|)\big) = \\ &= \sum_{q=0}^{j}\big(f_q(|T_t|) + f_q(|T_y|)\big). \end{aligned}$$

Similarly, from (3.8), one obtains,

$$\begin{aligned} f_j(|T_x|) &\leq \sum_{q=0}^{j-1}\big(f_q(|T_t|) + f_{j-q}(|T_y|)\big) + f_j(|T_t|) \leq \\ &\leq \sum_{q=0}^{j}\big(f_q(|T_t|) + f_q(|T_y|)\big). \end{aligned}$$

So, in both cases, the following recurrence relation holds,

$$f_j(|T_x|) \leq \sum_{i=0}^{j}\big(f_i(|T_t|) + f_i(|T_y|)\big). \tag{3.15}$$

Consider now the special case when x is a spine node. For nodes with one child, the situation is very similar. From (3.4), observe that the complexity of functions $OSC_R\big()$ and $OSC_L\big()$ at parent x is only one more than that of the child node $s_{\Pi'}$. For function $OBU_R\big()$, equation (3.7) can be interpreted as generating the same recurrence relation (3.15) where $|T_y| = 0$ and $|T_t| = |T_x|$. Moreover, we can safely assume that $f_j(1) = 1$ for any positive integer j.

Lemma 3.2. *For any node x of $SD(T)$,*

$$f_j(|T_x|) \leq \begin{cases} 1, & \text{if } j = 0 \\ |T_x| - 1, & \text{if } j = 1 \\ c|T_x|\log|T_x|\Big(1 + c\log|T_x|\Big)^{j-2}, & \text{if } j \geq 2, \end{cases} \tag{3.16}$$

where c and j are constant.

Proof. When $j = 0$, evidently $f_0 = 1$ because both cost functions are simply linear functions. For $j = 1$, there are exactly $|T_x| - 1$ edges that can be considered as split candidates so there are at most $|T_x| - 1$ linear pieces in both cost functions. Note that we compute $OBU_R\big(x, 1, \alpha\big)$ differently, but we count the split edges only when they belong to the tree induced by vertices from x.

For $j = 2$ we have,

$$f_2(|T_x|) \leq f_2(|T_t|) + f_2(|T_y|) + |T_t| - 1 + |T_y| - 1 + 2 = f_2(|T_t|) + f_2(|T_y|) + |T_x|.$$

$f_2(|T_t|)$ and $f_2(|T_y|)$ are recursively decomposed until we reach components of size one. Since the depth of the decomposition is bounded by $c' \log |T_x|$, where c' is some constant, the following inequality is satisfied,

$$f_2(|T_x|) \leq \underbrace{f_2(1) + \ldots + f_2(1)}_{|T_x|\times} + c'|T_x| \log |T_x| \leq c|T_x| \log |T_x|,$$

where $c = c' + 1$, and thus relation (3.16) is satisfied for $j = 2$.

The proof proceeds by induction on j. Assume (3.16) is true for all values smaller or equal to j. Then,

$$\begin{aligned}
f_{j+1}(|T_x|) &\leq f_{j+1}(|T_t|) + f_{j+1}(|T_y|) + \sum_{i=0}^{j}\Big(f_i(|T_t|) + f_i(|T_y|)\Big) \leq \\
&\leq f_{j+1}(|T_t|) + f_{j+1}(|T_y|) + |T_x|\Big(1 + c\log|T_x| + \sum_{i=3}^{j} c\log|T_x|\big(1 + c\log|T_x|\big)^{i-2}\Big) = \\
&= f_{j+1}(|T_t|) + f_{j+1}(|T_y|) + |T_x|\big(1 + c\log|T_x|\big)\Big(1 + c\log|T_x| + \\
&+ \sum_{i=4}^{j} c\log|T_x|\big(1 + c\log|T_x|\big)^{i-3}\Big) = \ldots \\
&\ldots = f_{j+1}(|T_t|) + f_{j+1}(|T_y|) + |T_x|\big(1 + c\log|T_x|\big)^{j-1}.
\end{aligned}$$

As before, $f_{j+1}(|T_t|)$ and $f_{j+1}(|T_y|)$ are recursively decomposed until we reach components of size one. Since the depth of the decomposition from x is at most $c' \log |T_x|$ and $c = c' + 1$, we obtain

$$\begin{aligned}
f_{j+1}(|T_x|) &\leq \underbrace{f_{j+1}(1) + \ldots + f_{j+1}(1)}_{|T_x|\times} + c'|T_x| \log |T_x|(1 + c\log|T_x|)^{j-1} = \\
&= |T_x| + (c-1)|T_x| \log |T_x|(1 + c\log|T_x|)^{j-1} \leq \\
&\leq c|T_x| \log |T_x|(1 + c\log|T_x|)^{j-1},
\end{aligned}$$

which concludes our proof. □

Using the previous lemma, the following result follows immediately. Note that k represents the number of facilities and thus the number of split edges is $k-1$.

Corollary 3.1. *For any given value of k considered constant, the bound $f_{k-1}(n)$ on the size of all continuous cost functions defined in Section 3.1 is $O(n \log^{k-2} n)$.*

Notice that if parameter k is considered part of the input, the bound from Corollary 3.1 must include c^{k-2} as factor. From the proof of Lemma 3.2 we know that $c = c' + 1$, where c' is the constant factor from the bound on the depth of the spine decomposition which was established in Section 2.2 and is at most 4. Therefore c is not larger than 5.

3.4 The complexity of the dynamic programming algorithm

Once we have ascertained a bound on the size of the continuous cost functions used in our dynamic programming algorithm, we can attempt to evaluate its running time performance and storage space. The algorithm is outlined by Program 3.1 and as we already know, most of its execution time is spent in the computation of the cost functions.

Storage space: Lemma 3.2 bounds the complexity of the cost functions stored at any internal node of the SD, if we consider k to be constant. By adding the sizes of the cost functions stored at all nodes from the same level in the decomposition, we obtain functions of the same order irrespective of which level the nodes were selected from, *i.e.* the functions have total size $O(n \log^{k-2} n)$. Since there are $O(\log n)$ different levels, it follows that the total size of all data structures used in storing the cost functions at every node of the decomposition is $O(n \log^{k-1} n)$. The storage space of the spine decomposition structure itself was shown to be $O(n)$ in Section 2.2. It follows that the space complexity of the dynamic programming algorithm as outlined by Program 3.1 is $O(n \log^{k-1} n)$.

However, not all cost functions are needed simultaneously to carry out the algorithm. We can modify the bottom-up procedure such that to discard the functions stored at nodes that are below some other node for which all cost functions are computed already. We are allowed to do this because we are only interested to calculate the cost functions at root s_{SD} of the spine decomposition.

For example, consider that every spine $\pi(v_0, v_m)$ is processed from its leaf v_m toward root v_0. This is done by traversing the search tree of the spine in post order and always selecting the child node toward the leaf first. Whenever the cost functions for the currently visited search tree node are determined, those of the two children are not needed anymore and the storage space can be freed. So, at any moment in the life cycle of the algorithm, the only search tree nodes that store cost functions are those adjacent to the path in $SD(T)$ that starts at the current SD node and ends at the root of the decomposition. Since the subtrees of T that correspond to these nodes are disjoint, it follows from Lemma 3.2 that the total storage space used by the algorithm is $O(n \log^{k-2} n)$. The steps of the algorithm are sketched in Program 3.2.

The reason why we chose to process the child towards the leaf first, becomes obvious if we look at the procedure to evaluate F_{OBU} which is described in Section 3.2.4. There, we need the value of functions $OSC_R()$ and $OBU_R()$ at nodes x_i adjacent to the path from the current SD node x_0 to the root, such that x_i are on the leaf side.

Running time: We now analyse the running time of the merging steps for computing cost functions recursively. Suppose we use a simple array to store the linear pieces of the continuous cost functions. The entries of such an array are ordered according to the succession of intervals for distance α of each linear piece. Similarly, $IBU_R(x, j, z)$ is stored in an array indexed by vertex z. With this data structure, given any fixed α, one can extract the value of any continuous cost function by binary search in $O(\log n)$ time. Given a fixed vertex z, the value $IBU_R(x, j, z)$ of a particular node x and number of split edges j can be retrieved in $O(1)$ time.

To estimate the total computation time, observe that we can sequentially traverse the cost functions stored at the children and construct the parent function in constant time for each linear piece of the parent. From here, it follows immediately that the time spent in the computation of all continuous cost functions has the same asymptotic behaviour as the total space of all cost functions at all SD nodes. If we denote by $T_{cont}(n)$ the fraction of the running time used in the computation of the continuous cost functions, then,

$$T_{cont}(n) \in O(n \log^{k-1} n).$$

We want to evaluate now the time consumed for the computation of discrete cost function $IBU_R()$. Consider first equations (3.10) and (3.11). Except for the evaluation of term F_{OBU},

---

k-median(T) {

- Compute $SD(T)$.
- Execute *compute_cost*(s_{SD}).
- For each vertex z in T do
  - Evaluate $IBU_R\big(s_{SD}, k-1, z\big)$.
- Return as solution the minimum value computed above.

}

compute_cost(x) {

- If x is a spine node with degree at most 2 in T, *i.e.* it is not adjacent to a spine component, then
  - Compute $OSC_L\big()$, $OSC_R\big()$, $OBU_R\big()$ and $IBU_R\big()$ directly using (3.3), (3.6), and (3.9).
- If $x = v_i$ is a spine node with degree 3 in T, let $s_{\Pi'}$ be the root of search tree for component $T(u_i)$. Then,
  - Execute *compute_cost*$(s_{\Pi'})$.
  - Compute $OSC_L\big()$, $OSC_R\big()$, $OBU_R\big()$ and $IBU_R\big()$ using (3.4), (3.7), (3.10), and (3.11).
  - Free memory used for functions at $s_{\Pi'}$.
- If x is internal to some search tree, let y and t be the children of x with y toward the leaf. Then,
  - Execute *compute_cost*(y).
  - Execute *compute_cost*(t).
  - Compute $OSC_L\big()$, $OSC_R\big()$, $OBU_R\big()$ and $IBU_R\big()$ using (3.5), (3.8), and (3.12).
  - Free memory used for functions at y and t.

  }

Program 3.2: Main steps of the improved k-median algorithm

all other operations can be performed in constant time for any given z. Different values of F_{OBU} are needed $O(k)$ times and thus cost $IBU_R()$ is acquired for a given parameter z in $O(\log^2 n)$ time for fixed k. Any parameter z can appear in at most $O(\log n)$ different functions $IBU_R()$, therefore the total time for the computation of discrete cost functions is $O(n \log^3 n)$.

Note that in our analysis above, we have assumed that the value C_{opt} used in (3.7) and (3.10) is available in constant time. However, we have not discussed yet how to compute this value. Since there are at most $k \cdot n$ different values for C_{opt}, if we denote by $T_{C_{opt}}(n)$ the time needed for the computation of one value of C_{opt}, then the total computation time for C_{opt} is simply $O(nT_{C_{opt}}(n))$. We can state all our observations in the following theorem.

Theorem 3.1. *The algorithm described by Program 3.2 for solving the k-median problem in trees has space complexity $O(n \log^{k-2} n + S_{C_{opt}}(n))$ and running time complexity $O(n \log^{k-1} n + n \log^3 n + nT_{C_{opt}}(n))$, where $k \geq 3$ is a constant, $T_{C_{opt}}(n)$ is the time complexity for solving the j-median problem on a particular subtree of T, and $S_{C_{opt}}(n)$ is the extra space required by the method that computes C_{opt}.*

3.5 Solving special instances of the k-median problem

The algorithm described earlier is not complete until we show how to compute the values C_{opt} that are needed in the recursive formulae for cost functions. In the present and following section we concentrate on this problem. To simplify the explanations, for a given $v_i \in T$ we refer to the evaluation of $C_{opt}(T(v_i), j)$ as the j-median subproblem. First, we consider the k-median problem applied on two special classes of trees, the directed and balanced trees. As it will become evident, these trees have a special structure for which the solution to the j-median subproblem is trivial. The algorithm that results is more simple. In Section 3.5.3 we examine the case of arbitrary trees but for $k = 3$. For this case, we use the framework of algorithm UKM and we provide an algorithm for the j-median subproblem where $j \in \{1, 2\}$.

3.5.1 Directed trees

In directed rooted trees edges are directed towards the root and any facility serving a particular vertex v must be an ancestor of v. The problem in this formulation has practical applications in optimizing the placement of web proxies to minimize average latency, see for

example Li *et al.* [74].

In [74], the authors presented a $O(k^2n^3)$-time algorithm for this problem. This was later improved to $O(k^2n^2)$ by Vigneron *et al.* [107]. In [107] the authors also noted that the Kariv-Hakimi algorithm can be modified to find k-median in directed trees in time $O(k^2n^2)$. In particular, it is also possible to adapt the algorithm of Tamir [99] for directed trees and use the same analysis to prove a running time of $O(kn^2)$. Chrobak *et al.* [28] gave algorithms sub-quadratic in n for $k = 2$ with running time $O(n \log n)$, and $k = 3$ with running time $O(n \log^2 n)$.

In the special case when the tree is a line, Li *et al.* [73] proposed an algorithm with running time $O(kn^2)$. This was subsequently improved by Woeginger [109] to $O(kn)$. As in the case of trees, the linear problem can also be solved in time $O(kn)$ by adapting the undirected k-median algorithm of Hassin and Tamir [56].

We can easily adapt the framework of algorithm UKM to model the restrictions of the directed edges of trees, which for this problem are always oriented from children to the root. For example, notice that discrete function $IBU_R\big(x, j, z\big)$ is defined only for $z = x_R$. Indeed, since a facility can only serve vertices in T that are descendants of the facility, it is not possible to have a subtree $T(x_R)$ rooted at x_R such that another vertex $z \in T_x$ serves x_R and $z \neq x_R$. Vertex x_R must be a facility, therefore we can use the value of $F_{OBU}(x, j, 0)$ instead of both $IBU_R\big(x, j, z\big)$ and $C_{opt}\big(T(x_R), j + 1\big)$. Another cost function that is undefined is $OSC_L\big(x, j, \alpha\big)$ because again, no external facility towards the leaf can serve any vertex from T_x.

Algorithm UKM in directed trees remains the same except for small differences that simplify the computation of cost functions. When node x is on the spine, we can use (3.3) and (3.4) substituting

$$C_{opt}(T_{s_{\Pi'}}, j) \overset{\text{with}}{\longleftarrow} OBU_R\big(s_{\Pi'}, j - 1, 0\big).$$

We can use here function $OBU_R\big(s_{\Pi'}, j - 1, 0\big)$ and not $F_{OBU}(T_{s_{\Pi'}}, j - 1, 0)$ because at root $s_{\Pi'}$ function $OBU_R\big()$ takes in consideration the optimal distribution of split edges in the entire subtree. For an internal search tree node, equation (3.5) remains unmodified.

To compute $OBU_R\big()$, when x is an internal search tree node, we can use (3.8) as is. When $x = v_i$ is a spine node, we can use (3.6) and (3.7) replacing

$$C_{opt}\big(T(v_{i+1}), q\big) \overset{\text{with}}{\longleftarrow} F_{OBU}(v_{i+1}, q - 1, 0).$$

Note that in the algorithm UKM described in Program 3.2, all cost functions at node v_{i+1}

are known when v_i is processed.

To estimate the running time of the algorithm for directed trees, observe that C_{opt} is obtained from function F_{OBU} evaluated for $\alpha = 0$. F_{OBU} is evaluated exactly as for the general case. Then, from the discussion in Section 3.4 it follows that

$$T_{C_{opt}}(n) \in O(\log^2 n).$$

By substituting this into Theorem 3.1 with the observation that the only value of function $IBU_R()$ that is computed coincides with the value for C_{opt} and, as consequence, term $O(n \log^3 n)$ is not present, we can state the following.

Theorem 3.2. *The algorithm for computing the k-median problem on directed trees where edges are oriented from the root toward the leaves and $k \geq 3$ takes $O(n \log^{k-1} n)$ time and $O(n \log^{k-2} n)$ space.*

3.5.2 Balanced trees

When tree T is balanced, there is no need for the spine tree decomposition as argued in Section 1.4.1. We can simply use the undiscretized cost functions from algorithm UKM associated now with subtrees of T rooted at a vertex of the tree. Recall from Section 1.1 that given $x \in T$, $T(x)$ denotes the subtree rooted at x, *i.e.* the subtree containing x obtained by removing edge $x\,p(x)$ from T.

Since there is no spine involved, restricted cost functions $OSC_L()$ and $OSC_R()$ are not defined. We work only with $OBU_R()$ and $IBU_R()$:

- Function $OBU_R(x, j, \alpha)$ is defined for every $x \in T$ and returns the cost in $T(x)$ if an external facility serves x and j split edges are located optimally in $T(x)$.
- Function $IBU_R(x, j, z)$ is defined for every $x \in T$ and $z \in T(x)$, and returns the cost in $T(x)$ if z is a facility covering x and j split edges are placed optimally in $T(x)$.

Consider Figure 3.7 where x has children y and t. Function $OBU_R()$ is obtained from the cost functions stored at the children vertices by considering the two new edges, xy and xt, being either both, or exactly one, or none split. If we denote by $A(j, \alpha)$ the contribution of vertices from $\{x\} \cup T(y)$ if j split edges are chosen from $xy \cup T(y)$, then we consider only new edge xy as possible split candidate and we can write,

$$A(j, \alpha) = w(x) \cdot \alpha + \min\left\{C_{opt}(T(y),\ j), OBU_R(y,\ j,\ \alpha + d(x, y))\right\}.$$

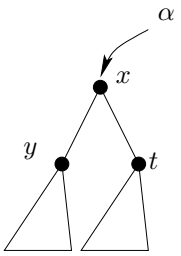


Figure 3.7: Computing cost functions in a balanced binary tree

Then, we can similarly incorporate the contribution of vertices from $T(t)$ by checking new edge xt for splitting, in other words,

$$OBU_R\big(x,j,\alpha\big) = \min\Big\{ \min_{1\le q\le j}\big\{C_{opt}(T(t),\ q) + A(j-q,\ \alpha)\big\}, \\ \min_{0\le q\le j}\big\{OBU_R\big(t,\ q,\ \alpha + d(x,t)\big) + A(j-q,\ \alpha)\big\}\Big\}.$$

To obtain the list $IBU_R\big()$, we can proceed in a similar way. We split the value of $IBU_R\big()$ in two, the first term $A'(j,z)$ accounting for the contribution of $\{x\}\cup T(y)$, and the second accounting for the rest. In these expressions, we assume that $z \in \{x\}\cup T(y)$. The case $z \in T(t)$ can be handled symmetrically. Now, we do not have to consider the new edges as split candidates, because function $OBU_R\big()$ incorporates this choice. We can simply write,

$$A'(j,z) = \begin{cases} IBU_R\big(y,j,z\big) + w(x)d(x,z), & \text{if } z \in T(y) \\ A(j,0), & \text{if } z = x. \end{cases}$$

To add the contribution of the remaining vertices, we need to check whether to split edge xt or not,

$$IBU_R\big(x,j,z\big) = \min_{0\le q\le j}\Big\{A'(j-q,z) + \min\big\{OBU_R\big(t,\ q,\ d(z,t)\big), C_{opt}(T(t),\ q)\big\}\Big\}.$$

For the computation of $C_{opt}\big(T(x),j\big)$, we can afford to spend time linear in $|T(x)|$ and use

$$C_{opt}\big(T(x),j\big) = \min_{v\in T(x)} IBU_R\big(x,\ j-1,\ v\big).$$

Indeed, any vertex $v \in T(x)$ can appear at most $O(\log n)$ times in a subtree of the type $T(x)$, and therefore, C_{opt} is evaluated in $O(\log n)$ amortized time. The recurrence relation

for the complexity of cost function $OBU_R()$ obtained from these formulae is again (3.15), so Theorem 3.1 can be used to state the following.

Theorem 3.3. *The algorithm for computing the k-median problem in trees with $O(\log n)$ height takes $O(n \log^{k-1} n)$ time and $O(n \log^{k-2} n)$ space for $k \geq 3$ constant.*

Note here that the constant hidden in the big oh notation is again exponential in k, but its value depends on constant c' that bounds the height of the input tree,

$$height(T) \leq c' \log n.$$

3.5.3 Arbitrary trees for case $k = 3$

This case is the most involved. The spine tree decomposition is used and all cost functions are computed as described in Section 3.2.

For a given spine vertex $v_i \in T$, we show in this section how to compute $C_{opt}(T(v_i), 1)$ and $C_{opt}(T(v_i), 2)$. Consider Figure 3.6 as reference, where $v_i = x_0$ and SD nodes x_1, x_2, ... x_b are adjacent to the path $\sigma(v_i, s_\Pi)$ and x_h is on the leaf side of the path. Spine Π contains vertex v_i.

Computing $C_{opt}(T(v_i), 1)$

As shown by Kariv and Hakimi [67], the optimal solution of the 1-median problem in a tree coincides with the weighted centroid of the tree. The weighted centroid is a vertex $v_c \in T$ such that the total weight of any of the three components of T obtained after the removal of v_c is smaller that $\frac{1}{2}$ of the total weight of T.

We can use the ideas of Breton [20] that allow us to compute in logarithmic time the 1-median of any subtree of T. Basically, we use the spine decomposition in which every node x contains the value $w(T_x) = \sum_{v \in T_x} w(v)$ which can be pre-processed. Then, a traversal of the decomposition from v_i to the root of the current search tree s_Π reveals weight $w(T(v_i))$, after which one can traverse the decomposition from s_Π top down towards the weighted centroid v_c. The top-down traversal is easy to implement because at every SD node x for which $T_x \subseteq T(v_i)$, we can determine in constant time the child y for which T_y contains the weighted centroid.

Computing $C_{opt}(T(v_i), 2)$

From Figure 3.6 we see immediately that subtree $T(v_i)$ is partitioned into subtrees T_{x_h} for $0 \leq h \leq b$. We will determine the optimal 2-median of $T(v_i)$ by finding the facility that covers root v_i. Assume this facility is vertex z^* from T_{x_h} for some $0 \leq h \leq b$. There are two possibilities where the only split edge of the 2-median can sit inside $T(v_i)$. The split edge can be (i) in one of T_{x_a} with $a < h$, or (ii) in $T(x_{hR})$. Recall that $T(x_{hR})$ represents the subtree of T rooted at x_{hR}.

Case (ii) is relatively easy to solve. The cost of 2-median solution is given by

$$\begin{aligned} C_{opt}\big(T(v_i), 2\big) =& IBU_R\big(x_h, 1, z^*\big) + \sum_{a=0}^{h-1} \sum_{v \in T_{x_a}} w(v) \cdot d(v, z^*) = \\ =& IBU_R\big(x_h, 1, z^*\big) + \underbrace{\sum_{a=0}^{h-1} \sum_{v \in T_{x_a}} w(v) \cdot d(v, x_{hR})}_{B_{h-1}} + \\ &+ \underbrace{\Big(\sum_{a=0}^{h-1} \sum_{v \in T_{x_a}} w(v)\Big)}_{W_{h-1}} \cdot d(x_{hR}, z^*). \end{aligned} \tag{3.17}$$

The first term of (3.17) represents the cost of subtree $T(x_{hR})$ when exactly one split edge is chosen in the subtree. The second term denoted B_{h-1} is the contribution of the remaining vertices but only up to the root x_{hR} of $T(x_{hR})$. The final term equals the additional cost spent by routing the service from x_{hR} to the actual facility inside subtree $T(x_{hR})$. Notice that the value for B_{h-1} above does not depend on z^* but only on the choice of node x_h. If we consider node x_h fixed, we can determine quickly which vertex $z^* \in T_{x_h}$ optimizes (3.17) using the techniques of Auletta *et al.* [8], and Hassin and Tamir [56]. Intuitively, the larger W_{h-1} is, the closer to root x_{hR} the optimal facility z^* must sit. The idea of Auletta is to compute, in a pre-processing phase, a partition of the values for weight W_{h-1} into intervals such that every interval is associated with exactly one minimizing vertex $z^* \in T_{x_h}$. This partition can be built without knowledge of the position of vertex v_i, and depends only on node x_h and on function $IBU_R()$ associated with it. Then given v_i, we can compute query weight W_{h-1} and use it in binary search over the partition stored at node x_h to retrieve z^* and the 2-median cost in logarithmic time.

In fact this process computes a function defined on the set of real numbers representing

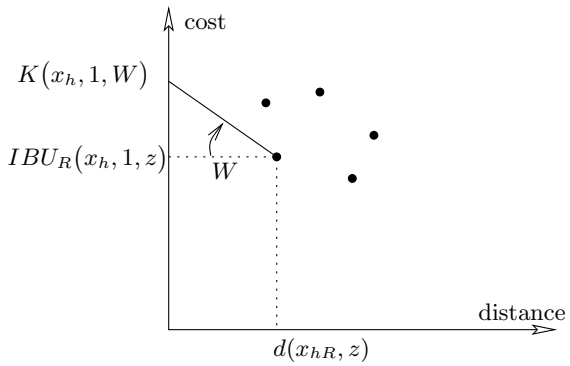


Figure 3.8: Function $IBU_R()$ represented as a set of points in two dimensional space, and an interpretation of the cover function

the values of weight W_{h-1}, and that returns a real number representing the optimal contribution of vertices from $T(x_h)$ under the constraints of case (ii). We denote this function by $K(x_h, 1, W)$ and call it the *cover* function. The second parameter represents the number of split edges located in $T(x_{hR})$. As expected, there also exists cover function $K(x_h, 0, W)$ which will be discussed shortly. Clearly, we obtain $C_{opt}(T(v_i), 2)$ under case (ii) as

$$C'_{opt}(T(v_i), 2) = \min_{0<h\leq b} \Big\{ K\big(x_h,\ 1,\ \sum_{a=0}^{h-1} \sum_{v\in T_{x_a}} w(v)\big) + B_{h-1} \Big\}.$$

In the above relation, we write C'_{opt} because the equation satisfies only one case of the two possible.

The cover function can be computed in linear time on top of any cost function $IBU_R()$ as follows. Consider the two dimensional Euclidean space where we have a set of $|T_{x_h}|$ points with y-coordinate $IBU_R(x_h, 1, z)$ and x-coordinate $d(z, x_{hR})$ as in Figure 3.8. From (3.17) it follows immediately that the cover function returns the intercept of a line with slope $-W$ that passes through the optimal point z^*. Since we are interested in the vertex minimizing the cover function, the set of candidate vertices for the optimum facility is given by the lower convex hull of the points from the two dimensional distance-cost space. The lower convex hull can be pre-computed for a fixed x_h in time linear in $|T_{x_h}|$ if the vertices $z \in T_{x_h}$ are available sorted by the distance to x_{hR}. Then, a Graham scan algorithm [88] computes the lower convex hull in linear time and stores it as a sequence of slope values. This sequence of slopes gives, in fact, the partition of the weight which defines the cover function. Note that it is easy to pre-process the SD in $O(n \log n)$ time so that every node x contains a list

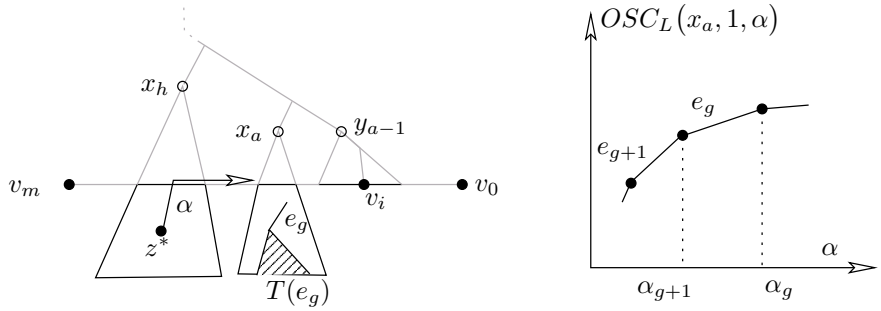


Figure 3.9: Solving the 2-median subproblem when the split edge is towards the root

of the tree vertices in T_x ordered by their distance to x_R.

We can now sketch a procedure that finds the optimal vertex $z^* \in T_{x_h}$ given that we know the total weight W_{h-1} of vertices in $\bigcup_{q<h} T_{x_q}$. Our definition of the cover function is independent on the number of split edges from subtree $T(x_{hR})$. Here we use j' for this number. Of course, for the 3-median problem, $j' \in \{0, 1\}$.

Step 1) Pre-process the lower convex hull $\mathcal{L}$ of the points defined by coordinates

$$\big(d(z, x_{hR}),\ IBU_R\big(x_h, j', z\big)\big).$$

Store it as a sorted array of slopes w_{ab} (called the slope-sequence of $\mathcal{L}$ [27]), where z_a and z_b are two vertices that determine consecutive points on the convex hull (Figure 3.10 (a)).

Step 2) Given query weight W_{h-1} for all vertices in $\bigcup_{q<h} T_{x_q}$, find the tangent point z^* to the line with slope $-W_{h-1}$ using binary search over the sorted w_{ab} array. Return

$$IBU_R\big(x_h, j', z^*\big) + W_{h-1} d(z^*, x_{hR}).$$

The result returned by Step 2 above is cover function $K\big(x_h, j', W_{h-1}\big)$. As already mentioned, a cover function can be evaluated in $O(\log n)$ time for a given W_{h-1}. The data structure for storing the cover function is the data structure used to store the lower convex hull $\mathcal{L}$. To compute $C_{opt}(T(v_i), j' + 1)$, we only need to evaluate the cover function for each T_{x_h} at $W_{h-1} = \sum_{q=0}^{h-1} w(T_{x_q})$. Parameter W_{h-1} can be easily precomputed so that it is available in constant time. Here $w(T_{x_q})$ represents the total weight of vertices in T_{x_q}.

Case (i) is not so simple. For this case, we consider that vertex z^* belongs to subtree T_{x_h} fixed but the split edge is now located in subtree T_{x_a} for $a < h$. Consider node x_a fixed as in Figure 3.9. For x_h and x_a fixed, we want now to quickly find the optimal median $z^* \in T_{x_h}$

and the optimal split edge $e_g \in T_{x_a}$. In the previous case, we used a cover function for node x_h to determine z^*, but now, because of edge e_g which is also unknown, we cannot compute the query weight to be used in the cover function at x_h. Indeed, edge e_g, if split, determines component denoted here $T(e_g)$ (see Figure 3.9) that is served by a separate 1-median and not by z^*.

However, if we were to estimate the cost of the 2-median optimal solution for $T(v_i)$, the contribution of subtree $T(x_{hR})$ is returned by cost function $IBU_R\big(x_h, 0, z^*\big)$, and the contribution of T_{x_a} by $OSC_L\big(x_a,\ 1,\ d(z^*, x_{aL})\big)$. Every linear piece of the later function is in fact determined by one particular edge of T_{x_a} and this edge is known. Thus, for every linear piece of function $OSC_L\big()$, the query weight W can be easily determined.

Intuitively, what happens when we solve the 2-median subproblem for subtrees $T(v_c)$ for $c < i$? Clearly, the weight of all vertices that are to the right of T_{x_a} and T_{x_h} becomes larger and larger, and therefore the optimal facility z^* moves towards x_{hR} to compensate for the extra weight. Then the distance α between z^* and the root x_{aL} of T_{x_a} decreases, and at some point, edge e_{g+1} becomes the optimal split edge in T_{x_a}. Now, our strategy has become to determine the amount of the extra weight W that causes the shift of the optimal edge from e_g to e_{g+1}.

If we do this operation for every pair of consecutive split edges from $OSC_L\big(x_a, 1, \alpha\big)$, we obtain another partition of weight W. This new partition can be used to obtain, by binary search, the optimal split edge in T_{x_a} first. Then, we can determine the complete query weight W' to be used with cover function $K\big(x_h, 0, W\big)$ which gives us the optimal facility z^* and the cost of the 2-median subproblem, for nodes x_h and x_a fixed. To calculate the true optimal solution, we need to repeat this procedure for all pairs of nodes x_h and x_a for which $b \geq h > a \geq 0$.

Computing the new weight partition: We now show how to compute the partition of weights for the optimal split edge. This weight partition can be determined at the time of computation of all cost functions (see Program 3.2). The weight partition needs to be calculated for every relevant pair of SD nodes x_h and x_a.

Consider for the fixed pair x_h - x_a that vertex v_i is mobile. Observe that v_i must be one of the leaves of node y_{a-1} in order for x_h and x_a to be relevant, where y_{a-1} is the sibling of x_a (see Figure 3.9). Denote by W_i the weight involved in determining the new partition,

i.e.

$$W_i = \sum_{c=0}^{a-1} w\big(T(x_c)\big).$$

We need a value $w_{g,g+1}$ such that if $W_i < w_{g,g+1}$, then e_{g+1} is a better candidate split edge, otherwise e_g is better. Instead of computing $w_{g,g+1}$ exactly, we can use the fact that $v_i \in L_{y_{a-1}}$ and thus we have knowledge of the actual value of W_i as v_i moves in the allowed range. Let Ω be the set of values for W_i in increasing order as v_i moves towards the root. Using prefix sums, it is easy to pre-compute Ω for all spines in linear time globally.

To determine $w_{g,g+1}$, our idea is to compute the 2-median cost of $T(v_i)$ for a given v_i in the allowed range (*i.e.* for a given $W_i \in \Omega$) in two versions, one in which edge e_g is split, the other in which e_{g+1} is split. Then, by comparing the two values, we can determine in which direction v_i should be moved (*i.e.* whether W_i should be increased or decreased). Clearly, if edge e_g gives a smaller cost than e_{g+1}, we have to increase W_i. As consequence, we can use binary search on Ω to find $w_{g,g+1}$.

Denote by $C_{opt}(e_g)$ the 2-median cost of $T(v_i)$ if e_g is split, and by $C_{opt}(e_{g+1})$ the solution of the same problem when e_{g+1} is split. We have,

$$\begin{aligned} C_{opt}(e_g) = K\big(x_h,\ 0,\ W_i + w(T_{x_a}) - w(T(e_g)) + \sum_{c=a+1}^{h-1} w(T_{x_c})\big) + \\ + \underbrace{\sum_{c=a-1}^{h-1} \sum_{v \in T_{x_c}} w(v) \cdot d(v, x_{hR})}_{\text{cancels out}} + \underbrace{\sum_{v \in T_{x_a} \setminus T(e_g)} w(v) \cdot d(v, x_{hR}) + C_{opt}\big(T(e_g), 1\big)}_{f_g(d(x_{aL}, x_{hR}))} + \\ + \underbrace{\sum_{c=0}^{a-1} \sum_{v \in T_{x_c}} w(v) \cdot d(v, x_{hR})}_{\text{cancels out}}. \end{aligned}$$

A similar expression exists for $C_{opt}(e_{g+1})$. Above, we denote by $f_g(\alpha)$ and $f_{g+1}(\alpha)$ the linear functions that determine the linear pieces of $OSC_L\big(x_a, 1, \alpha\big)$ for edges e_g respectively e_{g+1}. The first term in the right hand side of the equality computes the contribution of vertices in subtree $T(x_{hR})$. Using the cover function insures that the choice for the median in T_{x_h} is optimal given the total weight of vertices outside that it serves. The second term is redundant because it is present in the expressions for all split edges in T_{x_a}. It gives the cost of vertices in the subtrees between T_{x_h} and T_{x_a} up to the root x_{hR}. The cost for the extra distance between x_{hR} and the facility inside T_{x_h} is present in the value of the cover

function. The third term equals the contribution of vertices in T_{x_a} that are served from the median in T_{x_h} but only up to root x_{hR}. This value is in fact given by the appropriate linear function that makes up $OSC_L\big(x_a, 1, \alpha\big)$ for distance $\alpha = d(x_{aL}, x_{hR})$. Finally, the last term again cancels out. It represents the cost of vertices from the subtrees to the right of T_{x_a} when the service is routed x_{hR}.

All terms of the expressions above are known, and therefore they can be used in the binary search algorithm for finding $w_{g,g+1}$. The next border value $w_{g+1,g+2}$ is found in the same way. However, we must verify that $w_{g+1,g+2} < w_{g,g+1}$, otherwise we cannot use the partition later on. Basically, if the query weight W_i is such that $w_{g+1,g+2} \leq W \leq w_{g,g+1}$, then e_{g+1} is the best split edge in T_{x_a} for the given v_i. However, it is possible that, by computing the weight partitions, the values obtained are in fact $w_{g+1,g+2} > w_{g,g+1}$. In this case though, it can be argued that e_{g+1} will never be a candidate for the best split edge, and thus it can be eliminated from the picture. A new partition $w_{g,g+2}$ must be then computed. However, the total cost for the computation of this new boundary value does not exceed the size of function $OSC_L\big()$ and thus cannot be the dominating step in the algorithm. The entire algorithm for computing the optimal 3-median of input tree T is sketched by Program 3.3.

Analysis of the 3-median algorithm

In this section we analyse the complexity of the algorithm UKM for the case $k = 3$. The main result is established in Theorem 3.1, here we only complete the analysis by providing bounds on the total running time for computing all values C_{opt}.

The query for 1-median on a subtree of T can be done easily in $O(\log n)$ time (see the beginning of Section 3.5.3). For the 2-median query, case (ii) is more simple. The cover functions can be calculated whenever $IBU_R\big()$ functions are calculated in total time $O(n \log n)$, and each query can be solved in time $O(\log n)$. For a fixed v_i, there are $O(\log n)$ evaluations of the cover function, which means $O(\log^2 n)$ time per tree vertex, adding up to a total of $O(n \log^2 n)$ time for case (ii).

We now look at the time consumed for case (i). To compute one value $w_{g,g+1}$ of the partition intervals for a given pair of nodes x_h-x_a, we perform binary search over set Ω of weight values. This takes $O(\log n)$ time. At each element of Ω, we evaluate $C_{opt}(e_g)$ and $C_{opt}(e_{g+1})$. This time is dominated by the time to evaluate the cover function which is $O(\log n)$. All other values that are needed can be pre-computed in amortized constant time.

- Compute the SD and cost functions as described by Program 3.2.
- for each $x \in SD(T)$, do in addition:
  - Maintain the vertices in T_x sorted by distance to x_R.
  - Maintain ordered weights Ω.
  - Compute cover functions $K\big(x, 1, W\big)$ and $K\big(x, 0, W\big)$ with Graham scan.
  - For all SD nodes x_h adjacent to the path from x to the root of the curent search tree, compute the split edge weight partition relative to x_h.
- If $x = v_i$ on some spine, compute $C_{opt}(T(v_i), 1)$ and $C_{opt}(T(v_i), 2)$:
  - Identify the weighted centroid;
  - (case ii) For all SD nodes x_h adjacent to the path from v_i to the root of the curent search tree, query cover $K\big(x_h, 1, W\big)$ with the appropriate value W. Retain the minimum.
  - (case i) For all SD pair nodes x_h, x_a, identify the optimal split edge using split edge weight partition. Use $K\big(x_h, 0, W\big)$ with the W obtained from previous step to calculate the cost.
  - Return the smallest cost over all cases.

Program 3.3: Algorithm for computing the optimal 3-median of an arbitrary tree

Therefore, the time needed to compute one element of the partition is $O(\log^2 n)$.

We now evaluate the total number of consecutive edges e_g and e_{g+1} for which a $w_{g,g+1}$ is needed. From Corollary 3.1, the total size of all functions $OSC_L\big(x, 1, \alpha\big)$ is $O(n \log n)$, and for each pair of adjacent linear pieces of these functions we use $O(\log^2 n)$ time. The total time to compute the weight partitions for the split edge becomes $O(n \log^3 n)$. For a fixed v_i, to compute the 2-median cost of $T(v_i)$ under case (i), we look at $O(\log^2 n)$ pairs of nodes adjacent to the path from v_i to the root of the search tree. For each pair, we query the weight partition for split edge, then we query the cover function to calculate the cost. This amounts to $O(\log^3 n)$ for each v_i, *i.e.* a total of $O(n \log^3 n)$. We can gather all these conclusions into the following theorem.

Theorem 3.4. *The* 3*-median problem on an arbitrary undirected tree can be solved in* $O(n \log^3 n)$ *time and* $O(n \log n)$ *space.*

3.6 Solving the general instance of the k-median problem

The framework for solving the general k-median problem in sub-quadratic time is algorithm UKM. We only show in this section how to compute C_{opt}. Let $x_0 = v_i$ be a SD node on some spine as in the previous section. We want to evaluate $C_{opt}(T(v_i), j+1)$ where $T(v_i)$ is the component obtained by removing edge (v_{i-1}, v_i). Consider the search tree path from x_0 to the root of the search tree (Figure 3.6) and nodes $x_1, x_2 \ldots x_b$ adjacent to the path and defining $T(v_i)$.

The general idea for calculating C_{opt} for an arbitrary number of facilities is somewhat similar to the procedure used for the 3-median problem. We still determine the facility that covers vertex v_i by computing cover functions and, as long as the set of split edges is accounted by function $IBU_R()$, this is enough. But when split edges are outside the subtree for which we use $IBU_R()$, we proceed in a different manner. Recall that the cover functions are interpreted as the lower convex hull of all points that correspond to the values of function $IBU_R()$ in distance-cost space. If we want to see the effect of placing several split edges in some subtree outside, we simply add cost function $OSC_L()$ of the subtree outside to the y coordinate of all points in the distance-cost space and recompute the convex hull. In this way we obtain the representation of another cover function (which we call *generalized cover function*) that considers the influence of the split edges and that can be used directly to retrieve the optimal facility covering v_i. We describe this idea formally in the paragraphs

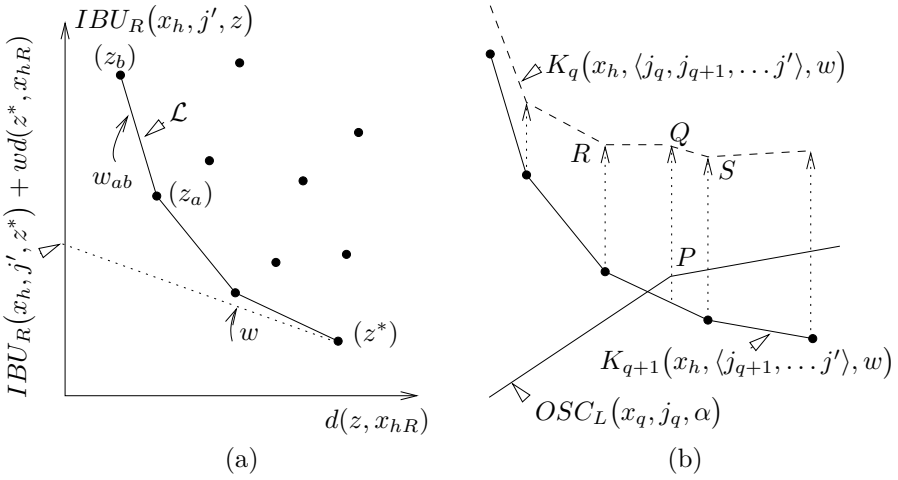


Figure 3.10: (a) Representation of function $IBU_R()$ as a point in distance-cost space. (b) Updating the lower convex hull after adding function $OSC_L()$

to come.

3.6.1 Computing $C_{opt}(T(v_i), j+1)$ for any constant j

Assume that the median covering v_i in the optimal $(j+1)$-median of $T(v_i)$ is vertex $z^* \in T_{x_h}$, and that we also know the distribution of the optimal j split edges in trees T_{x_q} for $0 \leq q \leq b$. Let j_q be the number of split edges in tree T_{x_q} (note that we are faithful to our convention and consider the spine edge between T_{x_q} and $T_{x_{q+1}}$ as associated with T_{x_q}). Denote by j' the number of split edges from T_{x_h} and above, *i.e.* $j' = j_h + j_{h+1} + \ldots + j_b$. The value $C_{opt}(T(v_i), j+1)$ is determined by the contribution of vertices from $T_{x_h} \cup \ldots \cup T_{x_b}$ returned by $IBU_R(x_h, j', z^*)$, and that of vertices from $T_{x_0} \cup \ldots \cup T_{x_{h-1}}$ that contain $j - j'$ split edges. We now describe how we can find vertex z^* in T_{x_h} quickly. Note that finding z^* must take time sub-linear in n, but we are allowed preprocessing in total time sub-quadratic in n.

If no split edge is chosen in any of the subtrees T_{x_q} for $q < h$ ($j' = j$), then we have a simple situation identical to case (ii) from Section 3.5.3, and cover function $K(x_h, j', W_{h-1})$ can be used as before to compute $C_{opt}(T(v_i), j+1)$.

We can show now how to accommodate an arbitrary number of split edges in trees T_{x_q} for $q < h$. If j_{h-1} edges are split in $T_{x_{h-1}}$, the contribution of vertices from $T_{x_{h-1}}$ to the $(j+1)$-median required is given by function $OSC_L(x_{h-1}, j_{h-1}, d(z^*, x_{h-1,R}))$ (see Figure 3.11). Hence, we need to add function $OSC_L(x_{h-1}, j_{h-1}, d(z, x_{h-1,R}))$ to each of the points

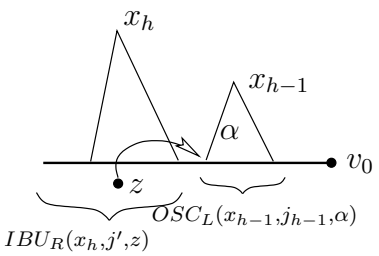


Figure 3.11: Computing the generalized cover function when j_{h-1} split edges are chosen in $T_{x_{h-1}}$

from Figure 3.10 (a) and update the lower convex hull for the given j_{h-1}. The new convex hull can then be used in a procedure identical to the one outlined before for computing the $(j+1)$-median. This updated convex hull encodes, in fact, a different cover function called *generalized cover function* that accommodates j_{h-1} split edges in $T_{x_{h-1}}$. We denote this generalized cover function by $K_{h-1}\big(x_h, \langle j_{h-1}, j'\rangle, w\big)$ and associate it with SD node x_{h-1}. In general, $K_q\big(x_h, \langle j_q, \ldots j_{h-1}, j'\rangle, w\big)$ for $q < h$ denotes the optimal median cost in subtrees $\bigcup_{p=q}^{b} T_{x_p}$ if the median covering v_i is in T_{x_h} and j_q, ... j_{h-1}, j' are the number of split edges selected in T_{x_q}, ... $T_{x_{h-1}}$ and $T(x_{hR})$ respectively. If we denote by $J(h)$ the set of all tuples of positive integers $\langle j_0, \ldots j_{h-1}, j'\rangle$ with $j_0 + \ldots + j_{h-1} + j' = j$, then

$$C_{opt}(T(x_0), j+1) = \min_{0\le h\le b}\Big\{\min_{\langle j_0,\ldots j'\rangle\in J(h)} K_0\big(x_h, \langle j_0, \ldots j'\rangle, 0\big)\Big\}. \tag{3.18}$$

Generalized cover functions

We can compute generalized cover function $K_q\big(x_h, \langle j_q, j_{q+1}, \ldots j'\rangle, w\big)$ from the convex hull corresponding to $K_{q+1}\big(x_h, \langle j_{q+1}, \ldots j'\rangle, w\big)$ as portrayed in Figure 3.10 (b). Recall from Section 3.1 that function $OSC_L\big()$ is piecewise linear and concave. Observe that if P from Figure 3.10 (b) is a critical point for $OSC_L\big()$, then Q is a reflex point, *i.e.* a point that violates the convexity requirement. However, the convexity can be easily restored by visiting the points on $\mathcal{L}$ adjacent to Q, S and R from Figure 3.10 (b) for example, and eliminating them if needed until no other reflex points remain. The procedure is sketched by Program 3.4.

The updating of the convex hull must satisfy certain requirements in order to keep our algorithm sub-quadratic in n. The time complexity of the procedure above is linear in the size of cost function $OSC_L\big()$ and it is proportional to the number of reflex points eliminated

---

– For every critical point P of $OSC_L\big(x_q, j_q, \alpha\big)$ do
  * Find the corresponding point Q on $K_{q+1}\big(x_h, \langle j_{q+1}, \ldots j'\rangle, w\big)$ by binary search.
  * Traverse the neighbors of Q on $K_{q+1}\big(x_h, \langle j_{q+1}, \ldots j'\rangle, w\big)$ sequentially, add function $OSC_L\big()$, and restore convexity by eliminating reflex points. Stop when no reflex points are left.

Program 3.4: Computing the generalized cover functions

from $K_{q+1}\big(x_h, \langle j_{q+1}, \ldots j'\rangle, w\big)$. Using these facts, we show in the following section that the total time required for the computation of $C_{opt}(T(v_i), j+1)$ for all v_i is still sub-quadratic in n.

3.6.2 Analysis of the UKM algorithm for the k-median problem

In the previous section we describe our method to compute the $(j+1)$-median subproblem $C_{opt}(T(v_i), j+1)$ based on generalized cover functions. Before explaining how to store and manage the generalized cover functions, we analyse the running time and storage space of the algorithm. The final details of the algorithm are presented in Section 3.6.3.

In Theorem 3.1, we prove the complexity of the general UKM algorithm as a function of the complexity of the algorithm for solving the $(j+1)$-median subproblem. We show that the space complexity is $O(n \log^{k-2} n + S_{C_{opt}}(n))$ and running time complexity $O(n \log^{k-1} n + n \log^3 n + nT_{C_{opt}}(n))$, where $k \geq 3$ is a constant, $T_{C_{opt}}(n)$ is the time complexity for solving the j-median problem on a particular subtree of T, and $S_{C_{opt}}(n)$ is the extra space required by the method that computes C_{opt}.

In this section we are concerned with proving the bounds on $T_{C_{opt}}(n)$ and $S_{C_{opt}}(n)$. To evaluate $T_{C_{opt}}(n)$, two facts must be established. First, because of (3.18) we need to bound the number of generalized cover functions that have to be calculated in order to obtain the result, *i.e.* the number of tuples $\langle j_0, \ldots j'\rangle$ used as arguments for function $K_0\big()$. There are at most $k-1$ split edges that must be distributed among $O(\log n)$ subtrees for constant k. We can consider each split edge as a distinguishable object that is assigned a digit from 0 to $O(\log n)$ identifying the subtree containing the edge. Therefore the total number of tuples generated is at most the total count of $k-1$ digit numbers that can be formed, *i.e.* it is $O(\log^{k-1} n)$ for constant k. If $T_K(n)$ is the time consumed by the evaluation of a generalized

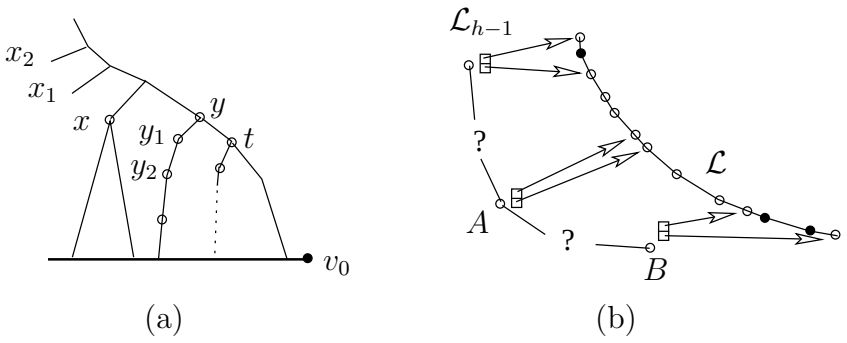


Figure 3.12: (a) Computing the generalized cover functions from function $IBU_R(x)$. (b) The data structure that stores generalized cover functions in distance-cost space; points eliminated from $\mathcal{L}$ appear black

cover function for a given parameter w, it follows immediately that computing the optimum cost takes $O(T_K(n)\log^{k-1} n)$ once all cover functions are known.

Second, we need a bound on the time needed to compute the cover functions. Consider a particular SD node x whose sibling SD node is y such that y is towards the root from x (Figure 3.12 (a)). Function $IBU_R()$ stored at node x is used in the computation of the generalized cover functions only if vertex $v_i \in T_y$ and y is towards the root. Recall that v_i is the vertex for which we need to evaluate $C_{opt}(T(v_i), j+1)$. We compute the generalized cover functions using the algorithm from Program 3.4. Denote by $T_{bin}(n)$ respectively $T_{ref}(n)$ the fraction from the total computation time taken by the binary search procedure, respectively the reflex point elimination procedure.

To evaluate $T_{ref}(n)$, consider y_1, y_2, ... y_l to be the left child (towards the leaf) of y, y_1, ... y_{l-1} respectively (Figure 3.12 (a)). If z_i comes from $T_{y_{a-1}} \setminus T_{y_a}$ for some $2 \le a \le l$, any convex hull vertex $z \in T_x$ can be eliminated in the process of computation of all generalized cover functions at most k times, once for each function $IBU_R(x, j', z)$ with $0 \le j' \le k-1$. Since l is $O(\log n)$, any vertex $z \in T_x$ can be visited at most $O(\log n)$ times. Any vertex z belongs to at most $O(\log n)$ trees T_x where x is a node of the decomposition, and therefore node z can be eliminated as reflex point for all generalized cover functions at most $O(\log^2 n)$ times. Thus, the total number of times an elimination takes place is $O(n \log^2 n)$. However, it is possible that we visit a point without eliminating anything. In this case we stop the convex hull updating procedure, and the time of this operation is dominated by the time of the binary search operation which is estimated shortly.

If we ignore the cost of checking but not eliminating anything, we can state that

$$T_{ref}(n) \in O(nT_K(n)\log^2 n).$$

We now estimate $T_{bin}(n)$. Consider now SD node x from Figure 3.12 (a) and nodes x_1, x_2, ... $x_{l'}$ above x that hang from the SD path $\sigma(x, s_{SD})$ towards the leaf. We want to evaluate the number of binary search procedures for the algorithm in Program 3.4 that involve the critical points of $OSC_L\big(x, j, \alpha\big)$ for some $j \leq k-1$. Clearly, there are $O(\log n)$ possibilities to chose one of the trees T_{x_1}, ... $T_{x_{l'}}$ as containing the median covering v_i wherever that vertex is. For one such possibility, there are $O(\log n)$ trees that must receive at most $k-1-j$ split edges (j split edges are already in T_x), which makes up for $O(\log^{k-1-j} n)$ cases, each involving $O(|T_x|\log^{j-1} n)$ critical points (from Lemma 3.2). Summing over all $O(\log n)$ possibilities, this amounts to $O(|T_x|\log^{k-1} n)$ binary search routines. Adding the binary search operations over all SD nodes x, we obtain $O(n\log^k n)$ search routines taking a total time

$$T_{bin}(n) \in O(nT_K(n)\log^{k+1} n).$$

From Theorem 3.1 and using the expressions above, we can state that the k-median problem on an arbitrary undirected tree can be solved in $O(nT_K(n)\log^{k+1} n)$ time, where $T_K(n)$ is the time required to evaluate the generalized cover function. For a simple data structure, $T_K(n) \in O(\log^2 n)$. However, we can use fractional cascading and improve this bound to $O(\log n)$. Details are provided in the next section.

3.6.3 Overview of the entire algorithm and implementation issues

From the analysis in the previous section we assumed that to compute cover function $K_q\big(x_h, \langle j_q, \ldots j'\rangle, w\big)$ we afford to spend time linear in the complexity of the cost function stored at x_q, $OSC_L\big(x_q, j_q, \alpha\big)$. This suggests that we can store generalized cover function $K_0\big(x_h, \langle j_0, \ldots j'\rangle, w\big)$ in a data structure distributed over nodes x_0, x_1, ... x_h such that at node x_q $(0 \leq q < h)$ the data structure is linear in the size of the cost function.

Consider Figure 3.12 (b) where we illustrate the data structure. It is simply a linked list of sorted slope-sequence values for the convex hull of the corresponding generalized cover function. Each list element corresponds to a critical point from continuous cost function $OSC_L\big()$. It contains two pointers to some element of the discrete cost function $IBU_R\big(x_h, j', \cdot\big)$ that define the interval of reflex points eliminated because of $OSC_L\big()$. In

between two consecutive list elements, for example A and B in the figure, we do not know the exact shape of the convex hull, we only know the slopes at the extremities of A and B. There are at most $O(\log n)$ lists such as $\mathcal{L}_{h-1}$ in the figure, one list for each node x_q with $0 \leq q \leq h-1$. Thus list $\mathcal{L}_q$ stores cover function $K_q\big(x_h, \langle j_q, \ldots j' \rangle, w\big)$.

Perhaps the best way to think about these lists is to view them as layers of an onion. Each layer maintains a list of fragments of the convex hull it represents (convex hull = generalized cover function). Two adjacent fragments can contain gaps of points from the original convex hull that were eliminated once the concave function was added. This makes the onion layers look more like slices of Swiss cheese, punctured by holes. For each convex hull fragment from a given layer, we know the actual linear functions at the extremities of the fragment, but we do not know the exact shape of the fragment. We only know it must be convex. If we need to traverse the fragment, we must visit layers at the lower level so that we can correctly skip over any gaps that might exist there. Notice also that any gaps from lower layers must be strictly contained in fragments at higher levels because of the way these fragments are built. For instance, whenever we compute the boundary of a gap at some level, we evaluate the generalized cover function by traversing all lower layers until a point that exists in the function is found, so the boundary cannot fall inside the gap from a lower layer. Therefore it is not difficult, with this data structure, to sequentially visit the critical points of any generalized cover function $K_q\big()$ in time proportional to $|K_q\big()| \cdot T_K(n)$ (this is needed in Program 3.4).

More on the storage of generalized cover functions

Consider Figure 3.13 where we illustrate several layers of the lists $\mathcal{L}_q$. To simplify the notation, we do not distinguish between a list, the convex hull it represents, and the generalized cover function. It should be clear from the context which object we refer to. Before we describe more precisely how the computation and the evaluation of the generalized cover functions take place, we make a few observations.

Observation 3.1. Given a polygonal line in distance-cost space, by adding a linear function to each point of the polygonal line, we obtain another polygonal line which is a rotated and translated image of the original one. In particular, we do not modify convex hull $\mathcal{L}$ or $\mathcal{L}_q$ (except for translation and rotation) if we add a linear function.

Observation 3.2. An element of a list $\mathcal{L}_q$ (for example, A in Figure 3.13) corresponds to a

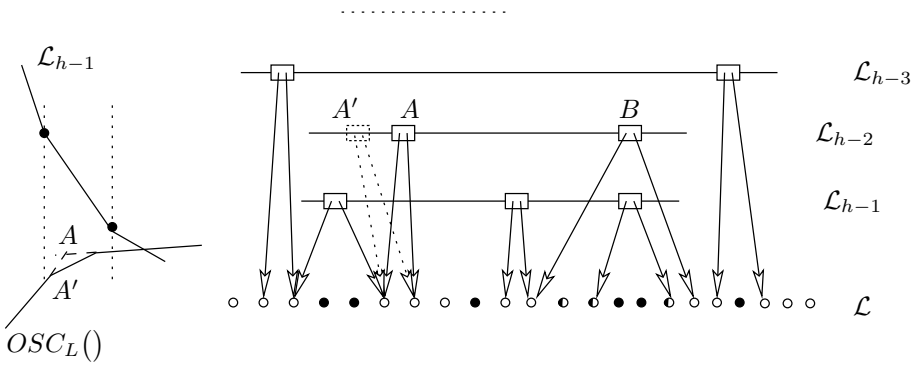


Figure 3.13: Three lists of generalized cover functions $K_{h-1}()$, $K_{h-2}()$, and $K_{h-3}()$

critical point of function $OSC_L()$ and it stores two indices to points of the convex hull $\mathcal{L}$. These indices identify convex hull points that exist in $\mathcal{L}_q$. In between these two points, it is possible that there are one or more points from $\mathcal{L}$, however these points are not present in $\mathcal{L}_q$. They were eliminated because they violated the convexity property for $\mathcal{L}_q$. We call the eliminated points *black*. The remaining points are *white*.

Observation 3.3. It is possible that two critical points of $OSC_L()$ that determine $\mathcal{L}_q$ fall on the same edge of the convex hull of $\mathcal{L}_{q+1}$ (for example, point A' in Figure 3.13). If this happens, one of the critical points can be eliminated.

Observation 3.4. Two consecutive elements of a list $\mathcal{L}_q$ (for example, A and B from Figure 3.13), correspond to the convex hull of the white points between the pointers of A and B. They also represent the convex hull fragment of $\mathcal{L}_{q+1}$, the list below $\mathcal{L}_q$, rotated and translated appropriately by the linear piece of function $OSC_L()$ that determines the critical points A and B.

Observation 3.5. Any element of a list $\mathcal{L}_q$ (for example, A in Figure 3.13) stores the true slope and y-coordinate of the two white points that are on the convex hull of $\mathcal{L}_q$. "True" here means that the values for slope and y-coordinate are those of the line $\mathcal{L}_q$. Since we cannot afford to touch any white points P' between those pointed by two consecutive elements of $\mathcal{L}_q$, the white points P' have the slope and y-coordinate of the original convex hull $\mathcal{L}$.

Observation 3.6. Any sequence of black points from different lists is either disjoint or the sequence for the list at lower level $\mathcal{L}_p$ is *contained* by the sequence for the list at the higher level $\mathcal{L}_q$, $q < p$. This will become more clear when we discuss the procedure to build the lists. An example is provided in Figure 3.13 by the sequence of points from $\mathcal{L}$ colored half

white, half black.

The evaluation of the cover function is simple. Suppose we are given weight w and we want to find the point on the convex hull $\mathcal{L}_q$ tangent to a line with slope $-w$. We first do a binary search over the list $\mathcal{L}_q$ to find the interval that contains the tangent point. From Observation 3.5, we know we can perform the search only on list $\mathcal{L}_q$. Once the interval is identified, we know that one white point in that interval is the tangent point. We do not know which one, but we know that the interval we just found is a fragment of the convex hull of $\mathcal{L}_{q+1}$, stored by the list immediately below $\mathcal{L}_q$, but which was rotated. The amount of rotation is given by the linear function mentioned by Observation 3.4 which is known. We can thus rotate our query line to account for this rotation, and we can proceed with a new query on $\mathcal{L}_{q+1}$, in the same way we did on $\mathcal{L}_q$. Because of observations 3.6 and 3.5, we know that whatever interval of white points we find in $\mathcal{L}_{q+1}$, the interval is contained by the one we found for $\mathcal{L}_q$. We proceed in this way, level by level, until we perform the final search over $\mathcal{L}$ which gives us the answer. The tangent point also gives us the value of the generalized cover function we queried. Note that this process takes $O(\log^2 n)$ time, but since it involves a sequence of searches through lists whose elements can be associated easily (the association is given by the amount of rotation necessary to hop from a list to the next one), we can use fractional cascading [26, 27] to reduce the query time to $O(\log n)$ without blowing up the storage space.

Observation 3.7. In an evaluation of a generalized cover function stored at $\mathcal{L}_q$, we have identified also a sequence of $h - q$ intervals (consecutive list elements) from $\mathcal{L}_q$, $\mathcal{L}_{q+1}$, ... $\mathcal{L} = \mathcal{L}_h$.

We now describe the operations needed during the construction of list $\mathcal{L}_{q-1}$. For this, we need to traverse the white points of $\mathcal{L}_q$ as described by Program 3.4. First, we can identify the convex hull edge of $\mathcal{L}_q$ that contains the critical point of $OSC_L()$ at node x_{q-1} and that would determine an entry in $\mathcal{L}_{q-1}$. We can do this exactly as in the evaluation process described above, but we do not guide our query by slope $-w$, we guide it by coordinate α of the critical point. From Observation 3.7, we know that we have $O(\log n)$ intervals that we can use to jump over any black point we might encounter if we traverse points sequentially on $\mathcal{L}$. From Observation 3.6 we know that, if we encounter a boundary white point, *i.e.* if we are visiting a point on $\mathcal{L}$ whose index appears in an element of the $O(\log n)$ list intervals identified during the first step, we can safely skip the strip of black points. What is the

(i) Compute the SD of T
(ii) Traverse the search tree of top spine, bottom up and from the leaf of the spine towards the root. For current search tree node x do:
  * if $x = v_i$ is a spine vertex do: recursively process component T_{u_i}, then compute $C_{opt}(T(v_i), j)$ by querying the cover functions, and compute functions $OBU_R()$, $OSC_L()$, $OSC_R()$, and $IBU_R()$. Compute a new cover function by incorporating $OSC_L()$ as in Program 3.4.
  * if x has children t and y do: compute $OBU_R()$, $OSC_L()$, $OSC_R()$, and $IBU_R()$. Compute a new cover function by incorporating $OSC_L()$ as in Program 3.4. Discard all data stored at t and y.
(iii) Return the optimal solution by computing the minimum of $IBU_R(s_{SD}, k-1, z)$.

Program 3.5: The dynamic programming algorithm that solves the k-median problem in trees

procedure for skipping? We know it is possible that a boundary white point is a boundary point for more than just one list from the $O(\log n)$ lists we have traversed, but because of Observation 3.6, we can skip to the rightmost boundary point (leftmost point if we traverse $\mathcal{L}$ to the left). In this way, we are guaranteed that we never touch any black points. In consequence, the list of elements we build for $\mathcal{L}_{q-1}$ will also satisfy Observation 3.6. We thus have the following lemma.

Lemma 3.3. *We can traverse any convex hull $\mathcal{L}_q$ in time $O(t \cdot \log n)$, where t is the number of convex hull points of $\mathcal{L}_q$ visited.*

Lemma 3.4. *The total storage space for maintaining generalized cover functions is*

$$S_K(n) \in O(n \log^k n). \tag{3.19}$$

Proof. For a fixed x_q and j_q, the size of the cost function at x_q (the number of critical points of $OSC_L()$ and thus the size of list $\mathcal{L}_q$) is $O(|T_{x_q}| \log^{j_q - 1} n)$ according to Lemma 3.2. The remaining $k - j_q$ split edges can be distributed in $O(\log^{k-j} n)$ ways, each case corresponding to a different list $\mathcal{L}_q$. This totals to $O(|T_{x_q}| \log^{k-1} n)$ list elements, and since there are $O(\log n)$ subtrees that assume the role of T_{x_h}, the total size of all lists involving x_q does not

exceed $O(|T_{x_q}| \log^k n)$ elements. Adding this for all nodes x_q maintained simultaneously, we obtain (3.19) immediately. □

Final remarks about the k-median algorithm

To recapitulate, we list the entire dynamic programming algorithm that solves the k-median problem in trees, in Program 3.5. All the results established in the last two sections allow us to state the following theorem.

Theorem 3.5. *The k-median problem in trees can be solved, for any constant k in time $O(n \log^{k+2} n)$ and space $O(n \log^k n)$.*

3.7 Conclusion

In this chapter, we have presented several ideas that can be used with a dynamic programming algorithm to solve the k-median problem in trees in time sub-quadratic in n, the size of the tree. One of the requirements for this result is to consider parameter k as a constant and not as part of the input because the running time of our algorithms is exponential in k. Our method uses a decomposition of the input tree in recursive components that has depth logarithmic in n and whose properties are discussed in Chapter 2. Because of the decomposition, a constant exponential in k is hidden by the O notation for the complexity of our algorithm, but it is not larger than 3.6.

However, it is interesting to note a conjecture made by Chrobak *et al.* [28] which states that the bound on the cost functions handled by our algorithms is in fact linear in n. If true, this would have a significant impact on the complexity of our algorithms. To date, we were not able either to prove or to disprove this conjecture. This is definitely an interesting topic for further research.

We have also looked at two special classes of trees on which simplified versions of our UKM template work. One such class contains trees for which edges have a specific orientation, namely from the root towards the leaves, and service can only flow in the direction of the edges. Facilities have to be ancestors for the vertices they serve and no facility can route service to its parent in the tree. As a consequence, one of the cost functions from our template, function $IBU_R()$, can be discarded. Function $IBU_R()$ can be viewed as modeling service that flows towards the root. Naturally, one can ask about instances in which directed

trees have an arbitrary orientation. In this case however, it is possible that service comes from a facility that is a descendant of the vertex in question. Thus, we cannot discard $IBU_R()$ from the computation. Although service flow cannot traverse any edge in both directions, we believe that with UKM, one still has to use the generic algorithm to solve instances of directed trees with arbitrary orientation.

For the 3-median problem, we give an algorithm with a running time of $O(n \log^3 n)$. Note that the generic algorithm for $k = 3$ has a time complexity of $O(n \log^5 n)$. A reason for this difference is that in the 3-median algorithm we use additional information which we do not know to exploit in the generic algorithm. When we compute the weight partition for determining the optimal split edge for the 3-median problem, we use our knowledge of the weights that determine the query for the best split edge. Because of this, we compute in fact an approximation of the weight partition, but an approximation that gives exact results for the instance we need. In contrast, for the general k-median algorithm, we compute generalized cover functions exactly.

Concerning the k-median problem in arbitrary undirected trees, our result has significance from both a practical and a theoretical point of view. From the theoretical point of view, our work offers a new algorithm for solving an important optimization problem, the k-median in trees, for which few algorithmic results have been published in the last ten years. It also advances the research towards finding better k-median algorithms for more general classes of graphs that have not been fully considered yet, such as the graphs with bounded tree-width [89].

In practice, our algorithm is not as involved as it may seem. The data structures used are standard, except perhaps if one desires to implement fractional cascading, and the algorithm is a mere implementation of the recursive expressions on cost functions.

Chapter 4

The 2-median problem in trees with positive and negative weights

This chapter looks at one of the two generalizations of the k-median problem studied in this thesis, the mixed obnoxious facility location problem. As pointed out in the introduction, the mixed obnoxious facility location problem concerns the placement of facilities that are obnoxious for a subset of the clients but desirable for the remaining ones. This can be modelled by assigning negative weights to the clients that view the facilities as obnoxious and positive for the rest. It turns out that by allowing some vertices of the network to have negative weight, the problem becomes more difficult. In fact, there are two different interpretations for the objective function. One is denoted MWD (*Minimum Weighted Distance*) and assigns a client to the facility for which the weighted distance is smallest. For clients with negative weight the serving facility is the farthest one. The other is referred to as WMD (*Weight times the Minimum Distance*) and assigns the client to simply the closest facility.

In the following sections, we study both problems for the case of locating 2 medians in arbitrary trees. In the next paragraphs, we give an overview of the results that exist in the literature for this problem. We then consider version WMD, for which we propose an algorithm with a running time of $O(n \log n)$ [12]. This is an improvement over the first algorithm proposed by Burkard *et al.* [21] which has a quadratic running time, and over the algorithm of Breton [20] whose solution runs in $O(n \log^2 n)$ time. The last section presents an algorithm for the more difficult WMD version.

4.1 Background

Pure obnoxious facility location problems have been in the attention of researchers for a long time. These problems seek the placement of facilities as far as possible from the set of clients and therefore the goal is to *maximize* their objective function. A paper by Lozano and Mesa [77] contains a survey of recent results regarding obnoxious facility location problems. Not long ago, Burkard *et al.* [22] introduced the model in which clients have positive and negative weights, and the objective function is based on the weighted distance from clients to facilities. This seems to be a more natural model because in many real life applications, parties involved in a common contract have quite often conflicting interests. For example, the location of an industrial waste collection center is obnoxious for the residents in the area but desirable for the industries using it.

The one median positive/negative weights problem was solved in linear time for trees and cacti[1] by Burkard *et al.* [22]. In a different paper, Burkard *et al.* [21] considered the 2-median problem in trees proposing an $O(n^2)$ algorithm for MWD in arbitrary trees, $O(n \log n)$ for MWD in stars, and $O(n)$ for MWD in paths. The WMD problem is more difficult than MWD because the optimal solution can consist of a median located not on a vertex. The same paper [21] describes an $O(n^3)$ algorithm for the 2-median WMD problem in general trees and an $O(n^2)$ algorithm if we force the medians to be chosen from the vertices of the tree or if the tree is a path. A list of applications for the WMD and MWD k-median problem appears in [21].

In the second part of this chapter, we show that the 2-median WMD problem in an arbitrary tree can be solved in $O(nh \log^2 n)$, where h is the height of the tree. In balanced trees (trees with logarithmic height), our algorithm runs in $O(n \log^3 n)$ time. Our approach is to pre-compute partial one median costs for subtrees of the original tree and then use these precomputed values to extract the optimum 2-median cost. To avoid a running time cubic in the number of vertices of the tree, we use again the spine decomposition.

[1] A cactus is defined as a graph with cycles such that any two cycles have at most one vertex in common.

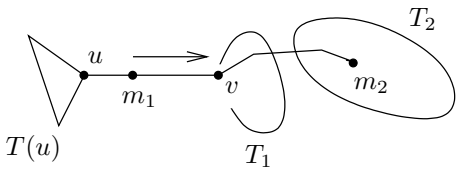


Figure 4.1: Case illustrating the vertex optimality of MWD 2-median problem

4.2 The MWD 2-median problem in trees

The MWD problem can formally be defined as to compute value

$$\min_{m_1,m_2\in T} \sum_{v\in T} \min\left\{w(v)\cdot d(v,m_1), w(v)\cdot d(v,m_2)\right\}.$$

Burkard *et al.* [21] show that problem MWD has always an optimal set of medians that are vertices of T even when we allow medians to be located on the edges of the network. We give here an intuitive argument.

Consider Figure 4.1 where m_1 and m_2 are the optimal location of the two medians in the tree. Let $T(u) \cup T_1$ be the subtree closer in distance to m_1, and assume that m_1 is on edge $[uv]$ (we use notation $[uv]$ to denote a network edge on which we can locate points). Here $T(u)$ is the component containing u obtained by deleting edge uv from T. Assume for now that there is a decrease in the total cost for the vertices served by m_1 if we move the median towards v. There are two possibilities, either (a) no vertex from T_2, the subtree of vertices originally closer to m_2, becomes closer to m_1 as a result of this move, or (b) at least one vertex from T_2 is closer to m_1 than to m_2 after the move. In the first case since there is no change in assignment for any vertex in T, we can continue moving m_1 until it hits v and overall, we get a better solution than the original one. This contradicts the optimality assumption for the starting configuration. For the second case, assume that v_2 is the vertex from T_2 which becomes closer to m_1. If v_2 has positive weight, then it is now served by m_1. Since m_1 moves towards v, the distance between m_1 and v_2 starts to decrease reducing even more the overall cost. If v_2 has negative weight, then v_2 is originally served by m_1 and its contribution to the 2-median cost increases because the distance to m_1 decreases. However, the overall cost decreases by assumption. Once v_2 is closer to m_1, it becomes served by m_2, but because m_2 is fixed, the contribution of vertex v_2 ceases to increase. But this contributes even more to the decrease of the overall cost, contradicting again the optimality of the original m_1 and m_2. A similar line of thought can be followed if the cost

for m_1 decreases as the median moves towards u.

The argument above suggests that problem MWD has nice properties. For example, if the allocation of client vertices to the medians is pre-determined, then the position of any median is the optimal 1-median solution for the subtree spanning the set of clients served by the median. In other words, we can thus use the split edge algorithm in the same way as for the k-median problem with positive weights (see Section 1.4.1).

The algorithm is simple. For every edge in the tree, we compute the 2-median cost if the edge is split. The optimal solution corresponds evidently to the edge with smallest cost. To compute the cost quickly given a split edge, the algorithm uses the spine decomposition of the input tree and extra information associated with the nodes of the decomposition and computed in a pre-processing phase. The ideas are very similar to those presented in Chapter 3. An obvious implementation leads to an algorithm with $O(n \log^2 n)$ time, however by postponing certain computations and executing them in a particular order, we can save a logarithmic factor.

4.2.1 Computing the 2-median cost given a split edge

Consider we have a spine decomposition of tree T and a split edge $v_i v_{i+1}$ on spine $\Pi = \pi(v_0, v_m)$ (Figure 4.2). As usual, v_0 is the root of the spine, *i.e.* the vertex adjacent to the parent spine. We denote by $T(v_i)$ and $T(v_{i+1})$ the components of T containing v_i respectively v_{i+1} obtained after deleting the split edge. In this chapter, we use quite frequently both components $T(v_i)$ and $T(v_{i+1})$, and since in the previous discussions we considered $T(v_i)$ as the subtree of T rooted at v_i (which was obtained by deleting edge $v_{i-1}v_i$ from T), we augment the notation introduced in Section 1.1 to avoid confusion.

Notation (See Figure 4.2). Given edge $e = v_i v_{i+1}$ for splitting,

- denote by $T(v_{i+1})$ the component containing v_{i+1} obtained as usual by removing edge e. This type of notation indicates that all vertices from $T(v_{i+1})$ have v_{i+1} as common ancestor.
- Denote by $T^c(v_i) = T \backslash T(v_{i+1})$ the complement of $T(v_{i+1})$ relative to T. This notation indicates that $T^c(v_i)$ is the component containing v_i but which also contains root r_T of the tree, and therefore the edge that was removed must have been $v_i v_{i+1}$.
- Let $V^+(v_i)$ and $V^-(v_i)$ be the set of vertices with positive respectively negative weight from $T(v_i)$. Alternatively, we use $T^+(v_i)$ and $T^-(v_i)$ to denote the same set of vertices.

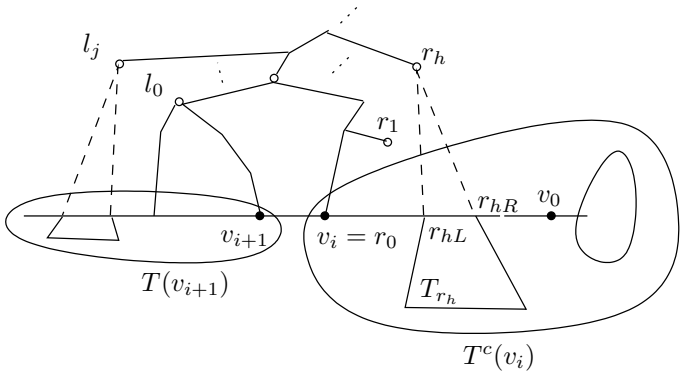


Figure 4.2: Computing the 2-median MWD cost for a given split edge

- Let $V^{c+}(v_i)$ (or $T^{c+}(v_i)$) and $V^{c-}(v_i)$ (or $T^{c-}(v_i)$) be the set of vertices with positive respectively negative weight from $T^c(v_i)$.
- Similarly, V_x^+ (or T_x^+) and V_x^- (or T_x^-) is the set of positive respectively negative vertices from T_x for some SD node x.

The cost of the two median is given by the optimal cost of one median serving $V^{c+}(v_i) \cup V^-(v_{i+1})$ and one median serving $V^{c-}(v_i) \cup V^+(v_{i+1})$.

We describe how to compute the median for $V^{c+}(v_i) \cup V^-(v_{i+1})$. The other case can be solved similarly. The best median for the case considered is located in $T^c(v_i)$. Let $\sigma(v_i, s_{SD})$ be the path in the spine decomposition from v_i to the root of the decomposition, and let r_h respectively l_j be the SD nodes adjacent to the path on the root side and leaf side of the path respectively. Let $r_0 = v_i$, as in Figure 4.2. Let l_0 be the highest node in the search tree with $l_{0R} = v_{i+1}$ (basically, the lowest node adjacent to the path from the leaf side). Denote by H_{SD} and J_{SD} the largest index of r_h and l_j respectively, *i.e.* $h \leq H_{SD}$ and $j \leq J_{SD}$. Denote also by H_Π (respectively J_Π) the largest index h (respectively j) for which r_h (respectively l_j) is a node in S_Π, the search tree node of spine Π (see Figure 4.3). Observe that the vertices of $T^c(v_i)$ are partitioned among components T_{r_h} for $0 \leq h \leq H_{SD}$ and T_{l_j} for $J_\Pi < j \leq J_{SD}$.

As in Chapter 3 for the problem with positive weights, we restrict the median to one of these components. Suppose the median lies in T_{l_j}. Based on the pre-computed information stored at l_j we then quickly find the best vertex in component T_{l_j} that minimizes the one median cost using cost function $IBU_R(l_j, 0, z)$ (Inside Big Unrestricted) and the cover function over $IBU_R(l_j, 0, z)$ for this purpose as in Section 3.1 (see Figure 4.4).

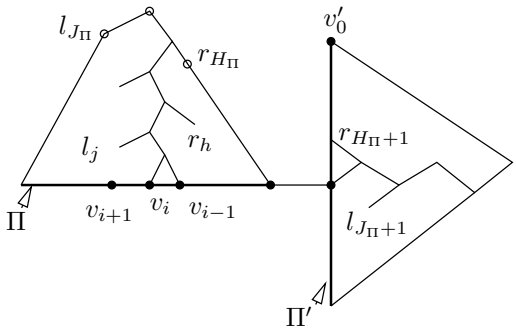


Figure 4.3: Spine nodes that determine subtree $T^c(v_i)$

If the median lies in T_{r_h}, we need a slightly different cost function than what has been used so far, one that measures the cost in the complement tree $T^c(v_i)$. This function denoted $IBU_L\big(r_h, 0, z\big)$ returns the cost of serving the positive weight vertices in subtree $T^c(r_{hL})$ by a facility located at $z \in T_{r_h}$. $T^c(r_{hL})$ is the component of T obtained after removing edge $r_{hL}\, r_{(h-1)R}$ and it contains root r_T of the tree. For node l_j, function $IBU_R\big(l_j, 0, z\big)$ is the same as in the definition from Section 3.1, except that it accounts *only* for positive vertices.

Consider node r_h with $h \leq H_\Pi$ belonging to the search tree of the current spine (Figure 4.2). Assuming that $z \in T_{r_h}$ is the median, the service cost of this median becomes

$$\sum_{v \in T^-(v_{i+1})} w(v) \cdot d(v, r_{hL}) + \sum_{j=0}^{h-1} \sum_{v \in T^+_{r_j}} w(v) \cdot d(v, r_{hL}) + \\ + \underbrace{\Big(\sum_{j=0}^{h-1} \sum_{v \in T^+_{r_j}} w(v) + \sum_{v \in T^-(v_{i+1})} w(v) \Big) d(r_{hL}, z) + IBU_L\big(r_h, 0, z\big)}_{\text{minimized by } K\left(r_h,\, 0,\, W^+_{h-1} + W^-(v_{i+1})\right)}, \quad (4.1)$$

where $W^+_{h-1} = \sum_{j=0}^{h-1} \sum_{v \in T^+_{r_j}} w(v)$ and $W^-(v_{i+1}) = w(T^-(v_{i+1}))$. The first term returns the cost of the negative vertices from the other side of the split edge but only up to the root r_{hL} of $T(r_{hL})$. The second term gives the cost of the positive vertices from the spine components between v_i and r_{hL}, again only to r_{hL}. In this way, these terms do not depend on the choice of the facility z. Finally, the third term returns the contribution of positive vertices inside $T^c(r_{hL})$ plus the difference of routing the total external weight from r_{hL} to z. This last term has an expression very similar to the first and last term of the addition in (3.17) from page 64. As explained in Section 3.5.3, the vertex that minimizes the expression can be obtained

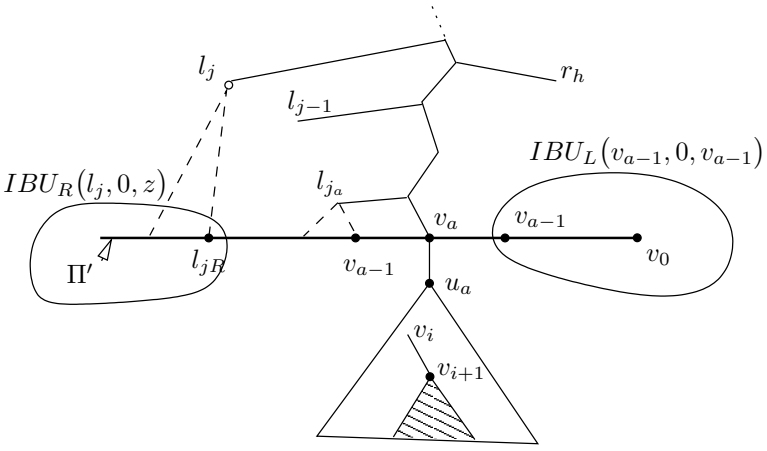


Figure 4.4: Computation of the 1-median cost when $z^* \in T_{l_j}$

by evaluating a *cover* function built on top of the discrete values of $IBU_L\big(r_h, 0, z\big)$. The cover function is denoted $K\big(r_h, 0, w\big)$.

The cover functions can be constructed exactly as in Section 3.5.3, the only difference being that $IBU_L\big()$ functions have a slightly different meaning, they account for the positive vertices towards the root from r_{hL}. This does not influence the computation procedure at all. For the case when $h > H_\Pi$, as in Figure 4.4, the best median in T_{r_h} is obtained in the same way. We consider now the computation involving the median in T_{l_j}.

Consider spine Π' such that the split edge $v_i v_{i+1}$ belongs to component $T(u_a)$, as in Figure 4.4. Let j_a be the smallest index for which $l_{j_a} \in S_{\Pi'}$ where $S_{\Pi'}$ denotes the search tree of spine Π'. The cost of 1-median for $T^c(v_i)$ when the median z is in T_{l_j} is,

$$\underbrace{\sum_{v \in T^-(v_{i+1})} w(v) \cdot d(v, l_{jR})}_{A} + \underbrace{\sum_{v \in T^+(u_a) \setminus T^+(v_{i+1})} w(v) \cdot d(v, l_{jR})}_{B} +$$
$$+ \underbrace{\sum_{v \in T^{c+}(v_{a-1})} w(v) \cdot d(v, l_{jR})}_{C} + \underbrace{\sum_{v \in T^+(v_{a+1}) \setminus T^+(l_{jR})} w(v) \cdot d(v, l_{jR})}_{D_{j-1}} +$$
$$+ \Bigg(\sum_{v \in T^-(v_{i+1})} w(v) + \sum_{v \in T^+(u_a) \setminus T^+(v_{i+1})} w(v) + \sum_{v \in T^{c+}(v_{a-1})} w(v) +$$
$$+ \sum_{v \in T^+(v_{a+1}) \setminus T^+(l_{jR})} w(v) \Bigg) d(l_{jR}, z) + IBU_R\big(l_j, 0, z\big) + w(v_a)\, d(v_a, z). \quad (4.2)$$

Term A represents the contribution of negative vertices from the other side of split edge v_iv_{i+1} and can be computed easily, for example with

$$A = \sum_{v \in T^-(v_{i+1})} w(v) \cdot d(v, v_{i+1}) + \Big(\sum_{v \in T^-(v_{i+1})} w(v) \Big) \cdot d(v_{i+1}, l_{jR}).$$

The first term can be pre-computed for any $v_{i+1} \in T$ in total linear time. The second term is available in constant time once the total weight of $T^-(v_{i+1})$ is pre-computed.

Term B from (4.2) can also be obtained in constant time using the same pre-computed information as in the case of A,

$$\begin{aligned} B = & \sum_{v \in T^+(u_a) \setminus T^+(v_{i+1})} w(v) \cdot d(v, u_a) + \Big(\sum_{v \in T^+(u_a) \setminus T^+(v_{i+1})} w(v) \Big) \cdot d(u_a, l_{jR}) = \\ = & \sum_{v \in T^+(u_a)} w(v) \cdot d(v, u_a) - \sum_{v \in T^+(v_{i+1})} w(v) \cdot d(v, v_{i+1}) - \Big(\sum_{v \in T^+(v_{i+1})} w(v) \Big) \cdot d(v_{i+1}, u_a). \end{aligned}$$

Term C can be immediately obtained from

$$C = IBU_L\big(v_{a-1}, 0, v_{a-1}\big) + \Big(\sum_{v \in T^{c+}(v_{a-1})} w(v) \Big) \cdot d(v_{a-1}, l_{jR}).$$

Term D_{j-1} is computed recursively as we move on the SD path from v_a to s_{SD}, in constant time per node l_j visited,

$$\begin{aligned} D_{j_a} = & \sum_{v \in T^+_{j_a}} w(v) d(v, l_{j_a R}), \text{ (can be pre-computed)} \\ W^+_{j_a} = & w(T^+_{l_{j_a}}), \\ D_j = & D_{j-1} + \sum_{v \in T^+_{l_j}} w(v) \cdot d(v, l_{jR}) + W^+_{j-1} \cdot d(l_{(j-1)L}, l_{jR}), \text{ and} \\ W^+_j = & W^+_{j-1} + w(T^+_{l_j}). \end{aligned}$$

Finally, the last two terms in (4.2) are minimized by the cover function, and the cover function can be computed in linear time using the discrete values of $IBU_R\big(l_j, 0, z\big)$.

The main steps of the 2-median MWD algorithm are given in Program 4.1.

We know from Section 3.5.3 that both $K\big()$ and $IBU_R\big()$ functions for all nodes in the decomposition can be constructed in total time $O(n \log n)$. We also know that the evaluation of the cover function in (4.1) takes $O(\log n)$ time, therefore the second step of the algorithm

- Compute first the SD of T and associated functions $IBU_R()$, $IBU_L()$ and $K()$ for each search tree node.
- For every edge in T, visit the SD nodes r_h and l_j and evaluate the cover function as in (4.1), on both sides of the split edge.
- Return the solution with smallest cost.

Program 4.1: Main steps of the 2-median MWD algorithm

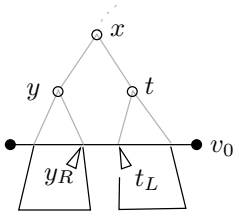


Figure 4.5: An internal search tree node

(evaluating all cover functions) takes $O(\log^2 n)$ time for each edge. In Section 4.2.3 we show that it is possible to determine an order of the edges of T such that the query weight for a particular cover function comes in increasing order thus eliminating the need for binary search over the weight partition. Instead, we can traverse the partition sequentially, once for all queries, saving a logarithmic factor in the running time. Now, we have to describe the computation of the new cost function $IBU_L()$.

4.2.2 Computing function $IBU_L()$

The computation of $IBU_L()$ is similar to that of $IBU_R()$, except that we have to account for the contributions of the positive vertices from a more complicated looking subtree, one that contains the root r_T. We can add these contributions if we perform two traversals, first of the input tree along the spines top-down, and second of the spine decomposition bottom up. Both operations are executed after the regular $IBU_R()$ functions are computed by the usual procedure.

Consider first the second traversal, the recursive part of computing $IBU_L()$. Here, in the same way as for the cost functions discussed in Chapter 3, we compute $IBU_L()$ for an SD node x using the $IBU_L()$ functions already known at its children, y and t. Using Figure

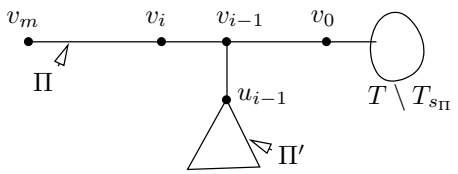


Figure 4.6: Computing function $IBU_L()$ for nodes on the spine

4.5 as reference and for $z \in T_t$, we have

$$IBU_L(x,0,z) = \underbrace{\sum_{v \in T_y^+} w(v) \cdot d(v, y_R)}_{C(y)} + IBU_L(t,0,z) + w(T_y^+)\, d(y_R, z).$$

It is very easy to verify that value $C(y)$ can also be computed bottom-up in total linear time for all y in the decomposition. Note that for $z \in T_y$, we have

$$IBU_L(x,0,z) = IBU_L(y,0,z).$$

We describe now the first traversal which is used to start-up the recursive procedure of the second traversal. Consider Figure 4.6 where Π is the current spine. We want to compute $IBU_L()$ for node v_i and we have to add for each $z \in T_{v_i}$ the contribution of vertices from $T \setminus T_{s_\Pi}$ and from T_{v_j} for all v_j between v_{i-1} and v_0, $i-1 \leq j \leq 0$. Assume that we recursively know the contribution of $T \setminus T_{s_\Pi}$ up to vertex v_0. For the top-level spine, this contribution is zero. The contribution of T_{v_j} between v_{i-1} and v_0 can be easily calculated by traversing the spine from v_0 to v_i. If we denote this contribution by $C(0:i-1)$, then we have

$$C(0:i-1) = C(0:i-2) + W(0:i-2) \cdot d(v_{i-2}, v_{i-1}) + IBU_R(s_{\Pi'}, 0, u_{i-1}) + \\ + w(T^+(u_{i-1})) \cdot d(u_{i-1}, v_{i-1}),$$

where $W(0:i-1)$ is just the total weight of subtrees T_{v_j} between v_{i-1} and v_0, $i-1 \leq j \leq 0$. In this equation, the second term equals the additional cost of vertices between $T_{v_{i-2}}$ and T_{v_0} for the extra distance between v_{i-2} and v_{i-1}. The third term computes the cost of vertices in $T(u_{i-1})$ up to u_{i-1} and the last term updates this contribution so that it accounts for the distance between u_{i-1} and v_{i-1}. With these values known, we obtain $IBU_L()$ at v_i from function $IBU_R()$ as follows,

$$IBU_L(v_i,0,z) = IBU_R(u_i,0,z) + w(T^+(u_i))\, d(u_i, v_i) + C(0:i-1) + \\ + W(0:i-1) \cdot d(v_{i-1}, z) + C(T \setminus T_{s_\Pi}) + W(T \setminus T_{s_\Pi}) \cdot d(v_0, z) + w(v_i)\, d(v_i, z).$$

Above, $C(T \setminus T_{s_\Pi})$ is the sum of weighted distances of positive vertices from the rest of the tree to v_0 and $W(T \setminus T_{s_\Pi})$ is the total weight of these vertices, therefore the last two terms amount to the cost of the rest of the tree when served by the facility z inside T_{v_i}. The two values $C(T \setminus T_{s_\Pi})$ and $W(T \setminus T_{s_\Pi})$ need to be updated when we recursively process spines at lower levels. For example for spine Π' in Figure 4.6, to calculate the new value of $C(T \setminus T_{s_{\Pi'}})$, we need also the cost of vertices between spine leaf v_m and v_i. This value is returned by $IBU_R\big(v_i, 0, v_i\big)$. Then, the contribution of the entire spine up to u_{i-1} is obtained as

$$C(T \setminus T_{s_{\Pi'}}) = IBU_R\big(v_i, 0, v_i\big) + \Bigg(\sum_{v \in T^+(vi)} w(v) \Bigg) \cdot d(v_i, u_{i-1}) + C(T \setminus T_{s_\Pi}) + \\ + W(T \setminus T_{s_\Pi}) \cdot d(v_0, u_{i-1}) + C(0 : i-2) + W(0 : i-2) d(v_{i-2}, u_{i-1}).$$

In this way, we can compute all functions needed by our algorithm in total time $O(n \log n)$ because the additional effort spent during the two extra traversals of T and $SD(T)$ is linear in n. Before we describe the entire algorithm, we need to show how to reduce the time for the computation of the 1-median solution on both sides of the split edge.

4.2.3 Improving the running time

As we already know, the dominating step in the obvious algorithm is to evaluate the cover function $K\big(r_h,\, 0,\, W^+_{h-1} + W^-(v_{i+1})\big)$ from (4.1) at $O(\log n)$ SD nodes for every spine edge. The evaluation of the cover function requires binary search over the partition of weights (the slope sequence of the convex hull of points in cost-distance space). But if we make sure that the values of the query weight come in sorted order at any given SD node, we can replace the binary search operation by sequential search. Any time a new value of the cover function is required, we resume the search from the position we stopped at the previous evaluation. Since there are $O(n \log n)$ total weight intervals counted over all nodes in the spine decomposition and $O(n \log n)$ total queries for the value of cover functions, the total time required by the evaluation is also $O(n \log n)$.

In the following paragraphs we describe an ordering of the split edges of T which generates a sequence of cover functions evaluated at non-decreasing weight values for any node in the decomposition. Refer to Figure 4.2 on page 87 where we consider edge $e = v_i v_{i+1}$ as

split edge. Node r_h uses weight

$$\begin{aligned} W_{r_h}(e) &= w\big(T^-(v_{i+1})\big) + w\big(T^{c+}(v_i) \setminus T^{c+}(r_{hL})\big) \\ &= W^-(v_{i+1}) + W^+_{h-1} \ \text{ (from (4.1))} \end{aligned}$$

where the first term is the total weight of negative vertices from the other side of split edge e and the second term is the weight of positive vertices from the same side of the split edge but between the split edge and the subtree considered by the cover function. Assume another edge $e' = v'_i v'_{i+1}$ generates a cover function call at the same node r_h. The weight it generates is

$$W_{r_h}(e') = w\big(T^-(v'_{i+1})\big) + w\big(T^{c+}(v'_i) \setminus T^{c+}(r_{hL})\big).$$

If we add $w\big(T^+(r_{hL})\big)$ to both expressions, we obtain two values, denoted $W(e)$ and $W(e')$ that do not depend on the choice of node r_h but only on the choice of the edge. Moreover, if $W(e) \leq W(e')$ then also $W_{r_h}(e) \leq W_{r_h}(e')$ for any node r_h for which both e and e' generate queries.

For every edge $e = v_i v_{i+1}$ we compute

$$W(e) = w\big(T^-(v_{i+1})\big) + w\big(T^{c+}(v_i)\big).$$

We sort the edges in increasing order by $W(e)$ and $W'(e)$ and solve the 1-median problem corresponding to each ordering by splitting the edges in the order given. For each split edge, we visit $O(\log n)$ SD nodes on the path in the decomposition from the edge to the root of the decomposition, and evaluate the cover functions sequentially. The algorithm is sketched in Program 4.2.

From all our arguments presented earlier, we can directly state the following result.

Theorem 4.1. *The 2-median problem on a tree with positive and negative weights under the objective MWD is solved by Program 4.2 in $O(n \log n)$ time and space.*

4.3 Solving problem WMD

In the WMD problem, one has to compute

$$\min_{m_1, m_2 \in N_T} \sum_{v \in T} \Big(w(v) \min \big\{ d(v, m_1), d(v, m_2) \big\} \Big).$$

- Compute the spine decomposition of T.
- Compute functions $IBU_R()$ for each node as in Chapter 3, and any additional information such as $C(y)$, *etc.*
- Traverse T top-down and $SD(T)$ bottom-up to compute $IBU_L()$ as in Section 4.2.2.
- Compute the cover functions $K()$ for $IBU_R()$ and $IBU_L()$.
- Sort the edges according to $W(e)$ and $W'(e)$.
- In each of the ordering, split the edges and compute the 1-median appropriate for the ordering, following the sequential procedure described in Section 4.2.3.
- For each edge, add the cost obtained above from both orderings and return the solution with minimum value.

Program 4.2: Algorithm for solving the 2-median MWD problem with positive/negative weights

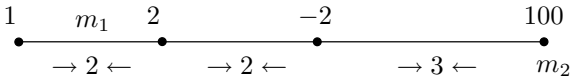


Figure 4.7: Optimal 2-median WMD solution on an edge of the path

Surprisingly, this problem is more difficult than version MWD. Consider our argument from page 85 on the vertex optimality of problem MWD. Using Figure 4.1 for illustration, we studied the case of median m_1 for which a direction improving the total cost of vertices it serves is towards m_2. Suppose that a vertex with negative weight from T_2 becomes closer to m_1. Originally, the vertex was served by m_2 and now it is served by m_1. Then, since the distance between m_1 and the vertex decreases, the overall cost might start to increase and we have a local optimum configuration with m_1 on an edge.

Figure 4.7 shows an example where the optimal solution has one median placed in the middle of the leftmost edge of the given path. The position of m_1 is such that the midpoint between the two medians falls exactly on the negative vertex. In fact, Burkard *et al.* [21] prove that the set of optimal solutions for the 2-median problem has two possible configurations:

1) Both medians are on the vertices of the network.
2) One median and the midpoint are on the vertices of the network, and the other median is on an edge.

It is also possible that the optimal solution consists of both medians placed on the same vertex of the network, as when all client vertices have negative weight. The value of the optimal solution for this degenerate case is equal to the solution of the 1-median problem. In the following, we assume that any two median algorithm computes also the optimal 1-median in a separate step, so we are simply ignoring this particular situation from our discussion.

The algorithm proposed by Burkard *et al.* [21] enumerates all possible candidate configurations for the optimal solution and updates the 2-median cost in amortized constant time. Our approach is similar to the techniques we used so far in Chapter 3. We partially enumerate the set of candidates for the optimal solution. Using pre-processed information, we are able to quickly find the best local optimum configuration from a larger set of configurations without going through each case sequentially. Details are provided in the following section.

4.3.1 General algorithm

Before we describe the algorithm, observe that midpoint p of any two points on the network, m_1 and m_2, is either along $\pi(m_1, r_T)$, or along $\pi(m_2, r_T)$, or both. This gives us three cases to consider if we want to cover the set of all candidates for the optimal solution:

(a) m_1 is a vertex, p is a vertex from the path from m_1 to the root, and m_2 is somewhere on an edge of T,

(b) m_1 is a vertex, p is on an edge from the path between m_1 and the root, and m_2 is somewhere on a vertex of T, and

(c) m_1 is on an edge, p is a vertex from the path from m_1 to the root, and m_2 is somewhere on a vertex of T.

Our algorithm considers each of the three cases above separately, as outlined by Program 4.3. To efficiently extract the best configuration from each set of partial configurations that we enumerate, we again make use of the spine decomposition (Chapter 2).

In each of the cases considered, the midpoint is located either on a vertex of the tree or on an edge, while the position of one of the medians is determined. As a consequence, any interaction between the two medians can be modeled by restricting the position of the second median somewhere in the second subtree at a certain distance from the midpoint. Generally, it is not difficult to compute the 1-median cost of the first median whose position and set of served vertices is pre-determined. The challenge is to compute the optimal cost of

---

- We compute the spine decomposition of T and preprocess all the information needed.
- For all possible cases do:

  Case 1: A median is on a vertex, the other on an edge. We fix vertex $m_1 \in V$ and for all vertices $p \in \pi(m_1, r_T)$ as midpoints, compute the smallest cost of the second tree if the second median is at distance $d(m_1, p)$ from p.

  Case 2: Both medians are on the vertices of T. Fix $m_1 \in V$. For all edges e on path $\pi(m_1, r_T)$, consider midpoint p as placed anywhere on e. Compute the best cost of the second tree if the second median is at distance $d(m_1, p)$ from p. This gives a range of values for the distance between m_2 and edge e.

  Case 3: A median is on a vertex, the other on an edge. We fix edge e in T for which $m_1 \in e$ and for all vertices p on the path from e to the root, compute the cost of the second tree served by m_2 such that $d(m_1, p) = d(p, m_2)$. Here, both m_1 and m_2 cover a range of distances from vertex p.

Program 4.3: Main algorithm for solving the WMD 2-median problem

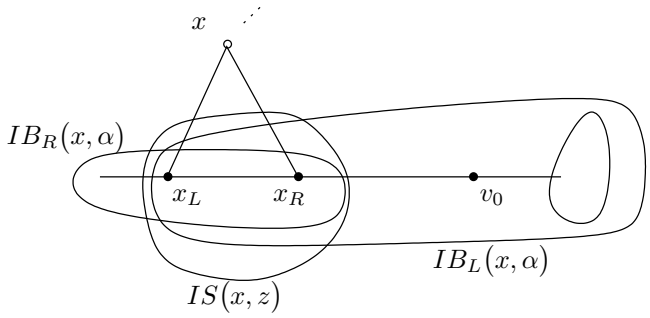


Figure 4.8: Illustration of cost functions $IB_R()$, $IB_L()$, and $IS()$

the second subtree. For this, we define and pre-compute a set of cost functions very similar to $IBU_R()$ and $IBU_L()$ from Section 3.1 and Section 4.2 respectively. As in Figure 4.8 for any given SD node x, we define:

- Function $IB_R(x,\alpha)$ (I = inside, B = big) is a continuous function and defines the best median cost in $T(x_R)$ given that the median is in T_x at distance exactly α from x_R. The median can be placed on an edge of T_x. Intuitively, this accounts for the contribution of all vertices in the tree toward the leaf from x_R.

- Function $IB_L(x,\alpha)$ is also continuous and returns the best median cost in $T^c(x_L)$ when the median is in T_x at distance α from x_L. Here, if e is the spine edge incident on x_L towards the leaf, then $T^c(x_L)$ is the component containing x_L obtained by deleting e, The function accounts for the cost of all the vertices towards the root from x_L.

- Function $IS(x,z)$ (I = inside, S = small) is a discrete function and returns the cost of vertices in T_x if they are served by a median located at vertex $z \in \{x_L, x_R\}$.

To compute the 1-median cost of the subtree containing the root of the tree (the difficult case) we use the same idea as in our previous algorithms. We restrict the median to each of the components making up the subtree and we compute the cost in each case selecting the one with the smallest value as solution.

4.3.2 Implementation of Case 1

Consider vertices m_1 and p fixed. We are going to evaluate the 2-median cost by walking along path $\sigma(p, s_{STD})$ and using the functions $IB_R()$, $IB_L()$, and $IS()$ stored at the nodes

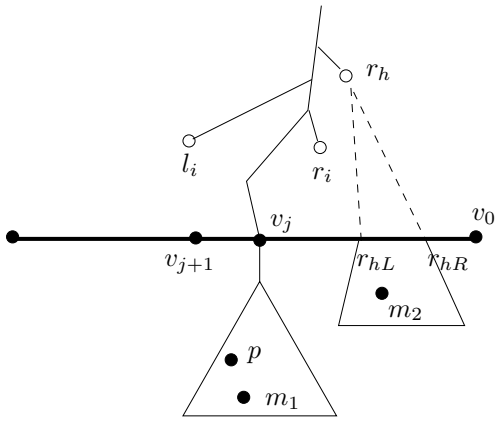


Figure 4.9: Computing the cost for the second median for Case 1

adjacent to the path.

Let $\Pi = \pi(v_0, v_m)$ be the current spine with v_0 towards the root and assume that p and m_1 come from T_{v_j} (see Figure 4.9). The cost of the vertices served by the first median (m_1) is easy to compute recursively as midpoint p moves towards the root because m_1 is fixed. Denote this cost by $C_1(m_1, p)$.

The value of $C_1(m_1, p)$ can be computed for different positions of midpoint p as it moves towards the root, in constant time for each new position of p. Consider Figure 4.10 where m_1 is on spine Π' and the current mid-vertex is on spine Π, $p = v_i$. Clearly, when p is moved over to v_{i-1}, the only change that occurs for $C_1(m_1, p)$ is that vertices from $T_{v_{i-1}}$ are now served by m_1. The contribution of all other vertices from the first subtree remains unaltered. Therefore,

$$C_1(m_1, v_{i-1}) = C_1(m_1, v_i) + IS\big(v_{i-1}, v_{i-1}\big) + w(T_{v_{i-1}}) \cdot \big(d(v_{i-1}, m_1)\big). \tag{4.3}$$

All terms from the relation above can be precomputed.

To compute the second median, let Π' be the spine that contains p and $s_{\Pi'}$ the root of the search tree over Π'. Let r_0, r_1, ... r_h, ... be the SD nodes adjacent to the SD path from p to s_{SD} towards the root, and let l_0, l_1, ... be the SD nodes adjacent to path $\sigma(s_{\Pi'}, s_{SD})$ towards the leaf. Nodes l_i and r_i above correspond to SD components whose union give the second subtree whose 1-median we want to compute (Figure 4.9).

Assume $m_2 \in T_{r_h}$. The cost of the second median is

$$C_2 = IB_L\big(r_h, d(p, m_1) - d(p, r_{hL})\big) + \sum_{v \in T^c(p) \setminus T^c(r_{hL})} \big(d(v, r_{hL}) + d(p, m_1) - d(p, r_{hL})\big) \cdot w(v),$$

- For vertices m_1 and p fixed do
  - Traverse the SD via search tree paths from p to the root s_{SD} and,
    - look at the left and right sibling node l_i and r_h.
    - Compute the 1-median of vertices served by m_2 if the median is restricted to the subtree induced by nodes shadowed by l_i or r_h.
    - Compute the 1-median cost $C_1(m_1, p)$ for the nodes closer to m_1, add with the value obtained above, and retain the solution with minimum cost.

Program 4.4: Implementation of Case 1

where, to simplify the equation, we write $T^c(p)$ for the subtree served by the second median. The relation is in fact very similar to (4.1) on page 88 except that here, we do not use cover functions to determine the location of the median, we actually know this location. It is at distance $d(p, m_1) - d(p, r_{hL})$ from r_{hL} inside T_{r_h} and we use function $IB_L\big(r_h, d(p, m_1) - d(p, r_{hL})\big)$ to determine it.

The second term of the expression equals the contribution of vertices not accounted for by function $IB_L()$ at node r_h, and can be computed in amortized constant time recursively, in almost the same way as function $IBU_L()$ from Section 4.2.2. A similar relation using function $IB_R\big(l_i, \alpha\big)$ is obtained for the case when $m_2 \in T_{l_i}$. Program 4.4 presents the steps for processing Case 1 as a whole.

4.3.3 Implementation of cases 2 and 3

For Case 3, the first median m_1 is fixed somewhere on an edge e and the midpoint p is placed on a vertex from the path from m_1 to root r_T. We do not know the position of the first median m_1 on edge e and therefore the position of the second median must be found at a distance from p that can vary over an interval of size equal to the length of e.

Let edge $e = uv$ with u being the endpoint closest to the root r_T (see Figure 4.10), and consider that median m_1 is located at distance $0 \leq \alpha \leq l(e)$ from u. The cost of the first median can be obtained exactly as in the previous section, only now this cost is a linear function of the distance from m_1 to u as m_1 moves on edge e. Let $f_1(\alpha)$ be the value of this cost as a linear function of α.

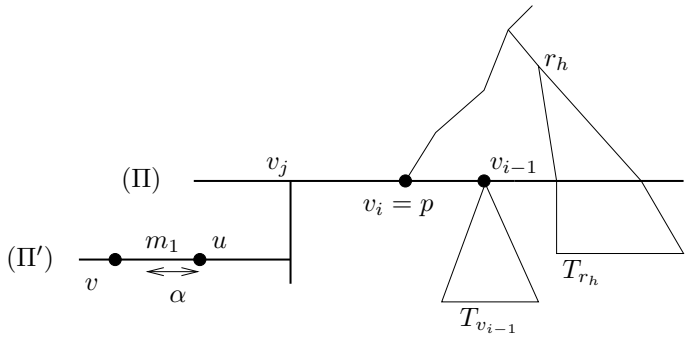


Figure 4.10: Computing the cost of the two medians for Case 3

For the second median, we proceed exactly as in Section 4.3.2. At every SD node r_h we need to add to function $IB_L\big(r_h, \alpha\big)$ the contribution of vertices in $T^c(p) \setminus T^c(r_{hL})$, exactly as for Case 1. Now the difference is that this contribution is a function linear in α because as m_1 moves on e, m_2 also shifts from a vertex to another in order to maintain the invariant $d(m_1, p) = d(m_2, p)$. Let $f_2(\alpha)$ be the value of this contribution as a linear function of α. The total cost of the 2-median problem is,

$$f_1(\alpha) + f_2(\alpha) + IB_L\big(r_h, d(u,p) - d(p, r_{hL}) + \alpha\big).$$

We denote $g(\alpha) = f_1(\alpha) + f_2(\alpha)$.

What we have to do is to quickly obtain a vertex $z^* \in T_{r_h}$ or that minimizes $IB_L\big(r_h, \alpha\big) + g(\alpha)$ for α in some interval A_α determined by the length of edge e. Figure 4.11 illustrates the situation. We can again interpret the value of $IB_L\big(r_h, \alpha\big) + g(\alpha)$ as the intercept of a line with slope equal to the negative of the slope of linear function $g(\alpha)$. The line must be tangent to the polygonal line that represents function $IB_L\big(r_h, \alpha\big)$ in the interval A_α. The same argument from Section 3.5.3 on page 63 can be used here to explain that the optimal vertex z^* is a lower convex hull vertex of the piece within A_α of the piecewise linear function $IB_L\big()$. The lower convex hull is determined by the critical points of $IB_L\big()$. In a similar way, we consider the case when $m_2 \in T_{l_i}$, this time using function $IB_R\big(l_i, \alpha\big)$.

The lower convex hull of functions $IB_L\big()$ or $IB_R\big()$ can be precomputed into a slope-sequence array as in Section 3.5.3 and thus the optimal point can be found in logarithmic time by binary search. One problem exists though, parameter α is confined to interval A_α. Therefore the correct approach is to compute the tangent point to the convex hull of only those critical points that fall in the interval. But the query intervals A_α are not known at preprocessing time.

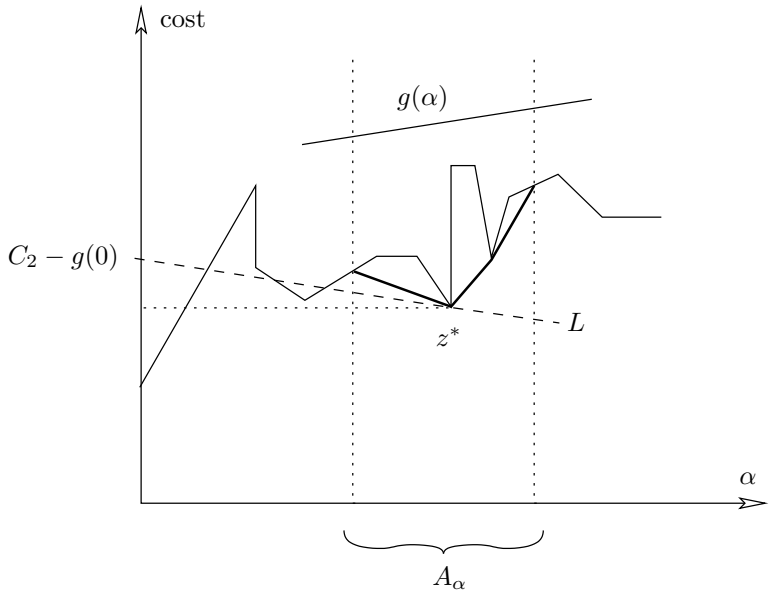


Figure 4.11: Finding the minimum of the sum between a piecewise linear function and a linear function over an interval A_α

The solution is to use the data structure of Overmars and van Leeuwen [84] for dynamic convex hull maintenance. The data structure consists of a binary search tree which we call *OL tree* to avoid confusion with the other tree structures mentioned in this thesis. The leaves of the OL tree store the critical points for the piecewise linear cost function, sorted by their x-coordinate in non-decreasing order. An internal node contains information which allows the reconstruction of the convex hull of all points at its leaves.

An illustration of an OL tree is given in Figure 4.12. The root node contains a list of the leaves that form the lower hull of the whole set of points, and an index in this list identifying the bridge between the convex hull of the left and right child node. The bridge is the line segment tangent to these two lower convex hulls and defines their union at the parent node. Any other internal node M contains a possibly empty list of points that are on the convex hull of the leaves of M but *are not* on the convex hull of the parent of M, and an index identifying the bridge between the hulls of the children of M. Starting from the root of the OL tree, one can visit any internal node and update the convex hull of its leaves in constant time per node, provided the data structure that stores the convex hull permits splitting and splicing in constant time. We can afford to use a doubly connected linked list to store the convex hull points because we will show later how to efficiently compute the tangent of the

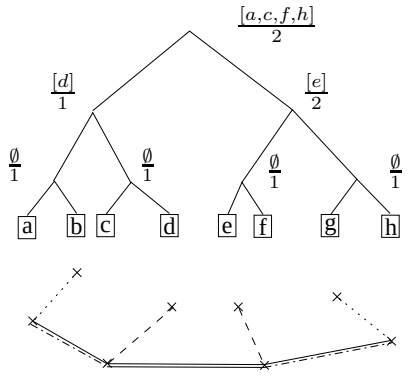


Figure 4.12: An OL tree. The nodes contain lists of convex hull points and indices in the complete convex hull list that identify the bridge. Bridges are shown by double lines, drawn differently if they belong to different levels in the OL tree

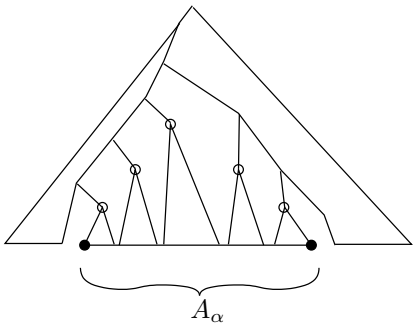


Figure 4.13: Representation of interval A_α in a segment tree

hull with a line by traversing the points on the hull sequentially. In conclusion, we construct an OL tree at each SD node r_h or l_i. We discuss now how to use the OL tree to compute the optimum cost of the 2-median problem under Case 3.

For a given SD node x with the corresponding interval A_α, we can use the OL tree at x to determine the critical points that fall in the interval A_α (the leaves of the OL tree are critical points of $IB_R()$ or $IB_L()$ sorted by their x coordinate) [88]. We identify $O(\log |T_x|)$ nodes in the OL tree that cover the interval A_α. Here, *to cover* means that the leaves of the $O(\log |T_x|)$ OL tree nodes are critical points of the respective cost function with the x coordinate in the interval A_α (see for example Figure 4.13 where the $O(\log |T_x|)$ nodes are drawn as hollow discs).

Given now the slope w of $g(\alpha)$, we start from the root of the OL tree and compute the lower convex hull at each of the $O(\log |T_x|)$ nodes of the OL tree as that cover interval

- For edge e and vertex p fixed, where $m_1 \in e$, do:
  - Compute the cost of the first median.
  - Traverse the $SD(T)$ from p to the root while updating function $g(\alpha)$; at every SD node r_h or l_i do
    - Start from the root of the OL tree and traverse the tree twice to identify interval A_α, once for each endpoint.
    - Identify the minimal number of OL nodes that cover A_α and assemble their convex hull.
    - Solve the query based on $g(\alpha)$ at each OL node identified above.

  Compute the cost of the 2-median based on the best result of the query above and retain the minimum over all cases.

Program 4.5: Implementation of cases 2 and 3

A_α as described in [84]. For each such lower convex hull, we compute the tangent point determined by weight w and the associated cost (the intercept of the tangent line) and retain the minimum value. This value represents the optimal solution to the 2-median WMD problem when the median is restricted in tree T_x. The algorithm is sketched by Program 4.5.

One observation should be perhaps made here. To simplify the presentation, we haven't mentioned anything about two points from the convex hull determined by an interval A_α. They are at the two extremities of interval A_α (see Figure 4.11). However, we do not have to consider these two points explicitely in our computation (the OL tree does not consider them!) because they do not correspond to a vertex in the subtree served by the second median. Our case only considers vertices as candidate medians for the second subtree.

The algorithm for Case 2 is identical with the one described above. The only difference is that the cost of the first median is constant because m_1 is chosen on a vertex. Now, $g(\alpha) = f_2(\alpha)$.

4.3.4 Preprocessing phase

The computation of functions $IB_R()$ and $IB_L()$ is not much different from the computation of the cost functions in Section 3.1 and Section 4.2. The only difference is that here, we merge (compute the minimum of) two piecewise linear functions that are not necessarily

concave. However, this is not difficult to implement in time linear in the size of the functions merged because the critical points of the two functions are available in sorted order by α.

4.3.5 Analysis of the WMD algorithm

Preprocessing:

We know from Chapter 2 that the spine decomposition can be constructed in $O(n)$ time, n being the size of tree T. The size of cost functions $IB_R()$ and $IB_L()$ at any SD node x is also linear in $|T_x|$. By counting the number of times an element (or vertex) can participate in different cost functions, it follows that the total computation time of all functions needed is $O(n \log n)$. The space complexity is also $O(n \log n)$. Finally, the OL tree at any node x can be built in time $O(|T_x| \log |T_x|)$ and therefore the total time to compute every OL tree is $O(n \log^2 n)$. The storage complexity of OL trees is linear, therefore the total space needed for all OL trees is $O(n \log n)$. We can conclude with the following lemma.

Lemma 4.1. *The time complexity for the pre-processing phase of the 2-median WMD algorithm is $O(n \log^2 n)$, and the space complexity is $O(n \log n)$.*

Implementation of Case 1:

For every pair of vertices m_1 and p, $O(\log n)$ SD nodes are visited and the value for a cost function at that node is required. The cost functions are piecewise linear with the critical points stored in sorted order by their abscissa. Therefore, to evaluate any cost function at given value α, one can identify the appropriate linear piece of the function by binary search. The time spent for the evaluation is dominated by the binary search step, and is $O(\log n)$. Therefore, the total time spent for a particular pair of vertices m_1 and p is $O(\log^2 n)$. Note that the cost of the first median is computed, as described in Section 4.3.2, in amortized constant time.

Lemma 4.2. *The time to solve Case 1 of our algorithm is $O(nh \log^2 n)$, where h represents the height of the tree.*

Implementation of Cases 2 and 3:

Similar to Case 1, the last two cases consider $O(nh)$ different choices for the first median and mid-vertex. For each choice, $O(\log n)$ SD nodes are visited. At each node, the quest for

- Enumerate pairs m_1 and p in the usual order; compute $g(\alpha)$ for every SD node x involved in the pair. Store the triple m_1, p, x under a key equal to the slope of $g(\alpha)$.
- Sort triples m_1, p, x according to their key.
- Generate and solve the queries in order for each triple m_1, p, x; use sequential search at the OL tree nodes to find the tangent point and save the position of the tangent at the OL node. Compute the 2-median cost corresponding to the query.
- Return the 2-median cost with smallest value.

Program 4.6: A modified algorithm for cases 2 and 3

the optimal vertex to serve as facility translates to computing tangent points at $O(\log n)$ OL tree nodes, and the computation of the tangent point requires, under usual circumstances, $O(\log n)$ time. Observe that this amount to $O(\log^3 n)$ step for each choice of m_1 and p.

However, we can use the same trick we used for the easier problem MWD to reduce a logarithmic factor. We follow the steps of the algorithm from Program 4.5 to generate function $g(\alpha)$ at every SD node required by the choice of m_1 and p points. Instead of computing the tangent point determined by $g(\alpha)$ right away, we store the slope of $g(\alpha)$ in an array. After all $O(nh \log n)$ slopes are generated, we sort them in $O(nh \log^2 n)$ time. We then proceed to answer them in sorted order as usual, except that when we actually process an OL node to find the tangent, we use sequential search instead of binary search and we make sure we save the position of the most recently tangent point obtained. Since the convex hull at each OL tree node is assembled in $O(\log n)$ time [84], it follows that the total time consumed for answering all $O(nh \log n)$ queries is $O(nh \log^2 n)$. An outline of the algorithm described above is given in Program 4.6.

Lemma 4.3. *Cases 2 and 3 are implemented in $O(nh \log^2 n)$ time and $O(nh \log n)$ space.*

One can notice that we require a lot of storage space for the class of trees with height linear in n simply because we sort all queries in a pre-processing step. If we can afford an extra logarithmic factor in running time, we can reduce the storage space of the algorithm to $O(n \log n)$, the bound required by the storage of the cost functions only. In this case, we can no longer compute tangent points of lines to convex hulls by sequential search. We need

to use binary search and the data structure to store convex hull fragments becomes more complicated. However, we can afford to spend $O(\log n)$ time to assemble the hull fragments because $O(\log n)$ time is spent anyway for binary search. The results from all three lemmas can be collected in the following theorem.

Theorem 4.2. *The 2-median problem for trees with positive and negative weights under the objective function WMD is answered by our algorithm in $O(nh \log^2 n)$ time and $O(nh \log n)$ space, where h represents the height of the tree. Alternatively, we can use $O(n \log n)$ space but the running time becomes $O(nh \log^3 n)$.*

4.4 Conclusion

In this chapter, we have presented improved algorithms for a generalization of the k-median problem in trees in which client vertices are allowed to have negative weight. This generalization is more difficult than the usual k-median problem in which all clients are positively weighted. For example, if we consider a facility moving along a path in a tree, the 1-median cost for the facility when all vertices are positive is a convex function, but when negative vertices are allowed, this is no longer true.

Using techniques similar to those from Chapter 3, we have improved the running time of the algorithms for solving two instances of this problem, the 2-median with WMD objective and the 2-median with MWD objective. For the MWD problem, our algorithm has a running time of $O(n \log n)$ improving the previously known quadratic solution of Burkard *et al.* [21]. To improve this result is perhaps a difficult task since no algorithm better that $O(n \log n)$ exists so far for the problem with positive weights.

For WMD problem, we propose an algorithm with $O(nh \log^2 n)$ running time where h is the height of the tree, improving the cubic algorithm published in the same paper [21]. This instance is much more difficult than MWD because the optimal 2-median set might not necessarily lie on the vertices of the tree. Our approach is based on solving several cases that cover the entire set of candidate solutions. One of these cases, the "vertex-edge" case, can actually be solved in $O(nh \log n)$ time. For the future, we propose to investigate means to reduce the complexity of the other two cases we consider. Another direction of research worth taking is the design of an algorithm whose performance is independent on the height of the tree while still sub-quadratic in n. We believe that to be successful, a different approach must be used, one that either avoids the explicit enumeration of all local

optima, or one that uses more properties of the optimum solution like in the algorithm of Breton [20] for the case with all vertices positive.

When the number of facilities is three or more, nothing is known about this generalization in trees. We believe that it is possible to adapt the dynamic programming algorithm of Tamir [99] for the general k-median WMD problem with positive/negative weights. The obvious approach leads though to a blow-up in the running time of the algorithm (although the bound remains polynomial in n) since we might need to enumerate a quadratic number of values for each cost function. This quadratic number has its source in the enumeration of local optima configurations. We feel that further research could lead to interesting results on this problem.

Moreover, other graph structures for which the k-median problem with positive vertices is efficiently solvable have not yet been considered within the positive/negative framework. These too constitute possible directions for future research and test grounds for techniques similar to those used in this thesis.

Chapter 5

The collection depots facility location problem

In this chapter we consider another generalization of the k-median facility location problem in trees called *the round-trip collection depots* location problem. We are given a set of locations at which one of two types of objects are already placed, clients or collection depots. A third class of objects, the facilities, dispatch vehicles that serve clients. We are asked to find a placement for one or more facilities so that the following scenario is optimized.

- To serve a client, a vehicle executes a tour that starts at the facility, visits the client (to collect garbage, for example), then stops at a collection depot (to dump the garbage), and finally returns to the originating facility.
- All clients must be served in this way.
- The objective function to optimize is:
  - Minmax problem (also called the *center* problem): to minimize the cost of the most expensive tour.
  - Minsum problem (also called the *median* problem): to minimize the total cost of all tours.

The collection depots location problem was recently proposed by Drezner and Wesolowsky [37] who characterized some properties of the optimal solution for minsum problems on the line and in the plane with Euclidean and rectilinear distances. They also proposed a heuristic algorithm to solve the Euclidean distance version in the plane and presented an empirical study of its performance. One year later, Berman *et al.* [15] considered the

minsum and minmax problems in general graphs and trees, and they analyzed the properties of the optimal solution. In [16] Berman and Huang extended their results on the minsum problem in graphs by looking at the case of placing multiple facilities. They established some properties of the optimal solution in trees and in a cycle, and they also studied the problem of locating the collection depots simultaneously with the facilities. Their method uses a Lagrangean relaxation algorithm embedded in a branch and bound framework.

Finally, Tamir and Halman [104] focused on the minmax problem in the plane, in graphs, and in trees with an additional constraint that specifies the sets of collection depots allowed for each client. They also considered two extensions of the depots collection problem, the customer one way and the depot one way collection problem. In the latter two versions, the cost of the return trip to the facility is not important, however the order in which the client and the depot are visited becomes important. As an example, Tamir and Halman showed that the minmax customer one way k facility problem in which the depot is visited before the customer and a set of allowable collection depots is given for each client, is NP-hard even on path graphs, whereas the depot one way problem is polynomially solvable in trees using the $O(n \log^2 n)$ time algorithm by Megiddo and Tamir for the k-center problem [79].

Other variations of the minmax problem in graphs exist. When the facilities are restricted to lie on the vertices of the network, we have the discrete version, as opposed to the continuous version where facilities can lie on the edges of the network. Also, depending on whether the clients are weighted or not and the cost of the tour is taken using weighted distances or not, we have weighted and unweighted minmax problems. Among other results concerning the minmax problem in the plane, Tamir and Halman gave algorithms for the following instances:

- For the unweighted 1-center collection depots problem in an arbitrary graph:

$$O(|E|(n \log n + S)) \text{ time,}$$

  where E is the set of edges, n the number of clients, and S the sum over all clients of the size of the set of collection depots allowed for that client. S can be between $\Theta(n)$ and $\Theta(n^2)$.

- For the weighted k-center collection depots problem in a tree:

$$O(n^2 \log n) \text{ time}$$

  in the discrete and continuous models.

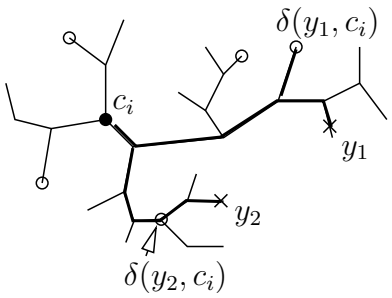


Figure 5.1: Example of trips from facilities y_1 and y_2 to client c_i

For both weighted and unweighted 1-center collection depots problem in a tree, Tamir [100] showed that an $O(n \log n)$ algorithm is possible.

In the following paragraphs, we study the 1-median collection depots location problem in trees. We show that using techniques similar to those employed in the previous chapters, we achieve an algorithm with $O(n \log n)$ running time and space. Unlike for the minmax objective, for this problem no previous results are known. For completeness, we also describe an obvious adaptation of Tamir's dynamic programming algorithm for the k-median collection depots problem in trees.

5.1 Notation and characterizations of the optimal solution

Let $T = (C \cup D, E)$ be a tree where $|C| = n_C$, $|D| = n_D$, and $|C \cup D| = n$. Let $C = \{c_1, c_2, \ldots, c_{n_C}\}$ be the customer nodes of T and let $D = \{\delta_1, \delta_2, \ldots, \delta_{n_D}\}$ be the depots nodes of T. It is allowed for a customer vertex and a depot vertex to coincide. Each customer $c \in C$ is associated with weight $w(c) \geq 0$. If T' is a subtree of T, we denote by $C(T')$ and $D(T')$ the client respectively depot set of T'.

We consider the tree as a network in which facilities can be located on edges. Let y be such a point in T. We denote the weighted trip distance from facility y to a client c_i by

$$r(y, c_i) = w(c_i)\Big(d(y, c_i) + \min_{1 \leq k \leq n_D} \big\{d(c_i, \delta_k) + d(\delta_k, y)\big\}\Big).$$

Figure 5.1 shows a typical example of two routes, one from facility y_1 to client c_i, the other from facility y_2 to client c_i. The depot locations are shown as empty discs and the optimal depot location for the route determined by y_j and c_i, denoted $\delta(y_j, c_i)$ where $j \in \{1, 2\}$, is not necessarily the closest depot to the client or to the facility, but it depends on the pair

facility-client. The k facility *minsum* (median) and *minmax* (center) problems are defined as follows:

minsum: Find k facilities y_1, y_2, ... y_k that give

$$\min_{y_1,y_2,\ldots y_k \in T} \Big(\sum_{i=1}^{n_C} \big(\min_{1\le h\le k} r(y_h, c_i)\big)\Big).$$

minmax: Find k facilities y_1, y_2, ... y_k that give

$$\min_{y_1,y_2,\ldots y_k \in T} \Big\{ \max_{1\le i\le n_C} \big\{ \min_{1\le h\le k} r(y_h, c_i)\big\}\Big\}.$$

Let $\delta(y, c_i) \in D$ denote the optimal depot used in the trip from y to c_i. Then,

$$r(y, c_i) = w(c_i)\Big(d(y, c_i) + d\big(c_i, \delta(y, c_i)\big) + d\big(\delta(y, c_i), y\big)\Big).$$

If we analyze the trip for c_i from y and keep in mind that T is a tree network, we notice that the trip from y to c_i can be partitioned into two parts. One part involves the path from y to c_i, denoted by $\pi(y, c_i)$, and the other involves the path from $\delta(y, c_i)$ to $\pi(y, c_i)$. Hence

$$r(y, c_i) = 2w(c_i)\Big(d(y, c_i) + d\big(\delta(y, c_i), \pi(y, c_i)\big)\Big). \tag{5.1}$$

The weighted distance of the trip from y to c_i is determined by the pair of elements $\big(d(y, c_i)$, $d(\delta(y, c_i), \pi(y, c_i))\big)$, *i.e.* by the length of two paths, one from the facility to the client (path facility-client) and the other from the depot to the path facility-client.

As in the case of the classic k-median problem, the k-median collection depots facility location problem admits an optimal solution with the set of facilities chosen from the set of client and depot vertices. This was shown by Berman and Huang in [16]. We restate their result here because it is essential for the correctness of the algorithms proposed in this chapter.

Lemma 5.1. *For the weighted minsum problem on a network, there always exists an optimal set of facility locations which is a subset of $C \cup D$.*

Sketch of proof. Consider Figure 5.2 where we assume that facility y_h is located inside edge uv in the optimal solution. Let $T(u)$ and $T(v)$ be the components of T obtained after the deletion of edge uv. We split the set of clients served by y_h in the optimal solution into four sets:

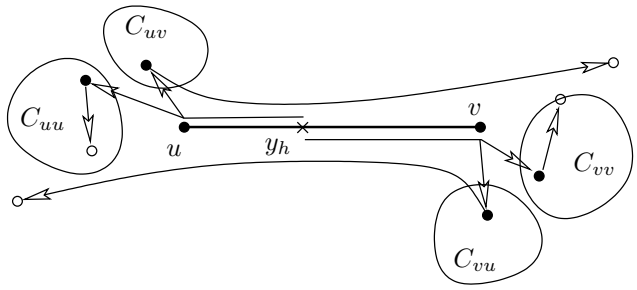


Figure 5.2: Vertex optimality for the k-median collection depots problem

- C_{uu}: the set of clients from $T(u)$ that use a depot also from $T(u)$.
- C_{uv}: the set of clients from $T(u)$ that use a depot from $T(v)$.
- C_{vu}: the set of clients from $T(v)$ that use a depot from $T(u)$.
- C_{vv}: the set of clients from $T(v)$ that use a depot also from $T(v)$.

Observe that the total cost for serving the clients from $C_{uv} \cup C_{vu}$ remains constant as y_h moves on edge uv because the clients served use a depot from the other side of the edge and the same distance is traversed no matter where the facility is placed on the edge. Let $w(C')$ denote the total weight of clients from a subset C'. If $w(C_{uu}) \geq w(C_{vv})$, then from (5.1) we see that y_h can move on u without increasing the total cost. □

The previous lemma suggests that the k-median collection depots problem has nice properties and might not be too difficult to solve. This is partially true. For example, unlike in the usual k-median problem, the optimal solution in the collection depots problem does not always split the client set into k connected components, and thus an algorithm based on the split edge technique is not suitable.

Figure 5.3 illustrates a tree where the optimal 2-median collection depots solution determines three connected groups of client vertices served by the same facility. The two facilities are placed at A and C. We can force this placement by assigning a huge weight to vertices A and C as clients and connecting A with A' through a very long edge. A' is both a client and a depot, and its weight is slightly larger than the total weight of clients from path $\pi(A, B)$ served by A. As a result, there is no incentive to move the facility from A on a vertex from path $\pi(A, B)$, and of course, no incentive to move the facility over to A'. The collection

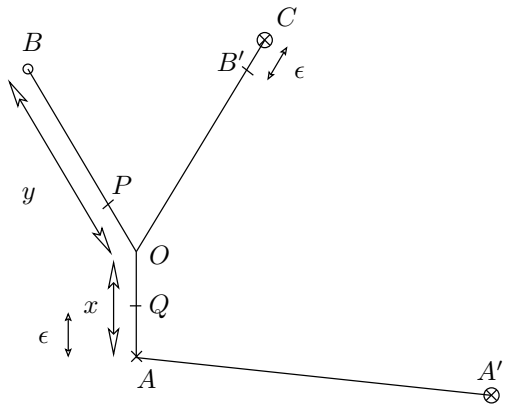


Figure 5.3: An optimal 2-median that split the clients into three connected sets, each served by one facility

depots are at B and C. We have

$$d(B,O) = d(B',O) = y, \qquad d(B',C) = \epsilon$$
$$d(O,P) = d(O,Q) = x - \epsilon, \qquad d(O,A) = x.$$

Note that for any client served by facility A on path $\pi(A,B)$, the best collection depot is B because C is too far and A' is the depot used only for A' as client. In the same time, the cost of the trip from A to any client on path $\pi(A,B)$ is the same, namely $x+y$ (assume all clients have weight $\frac{1}{2}$). However, for any client on path $\pi(P,Q)$, the cost of the trip from facility C is no more than $y+\epsilon+x-\epsilon$ and thus any client in $\pi(P,Q)$ is served by C and not A. In fact, the set of clients to be served by A is the disjoint set $\pi(P,B) \cup \pi(Q,A) \cup \{A'\}$.

Despite this behaviour, we will show in Section 5.3 that it is still not difficult to use the classic algorithm of Tamir [99] almost directly to obtain the optimal solution for the k-median collection depots problem. But first, we describe how to solve the 1-median case.

5.2 1-median collection depots problem

The general idea of the 1-median algorithm follows a pattern similar to the work of Rosenthal and Pino [90] who studied the location of one facility in trees, with several objective functions, in linear time. For this problem, the same framework seems not to lead to linear time algorithms very easily because of the particularities of our distance function. We propose here an algorithm with running time $O(n \log n)$.

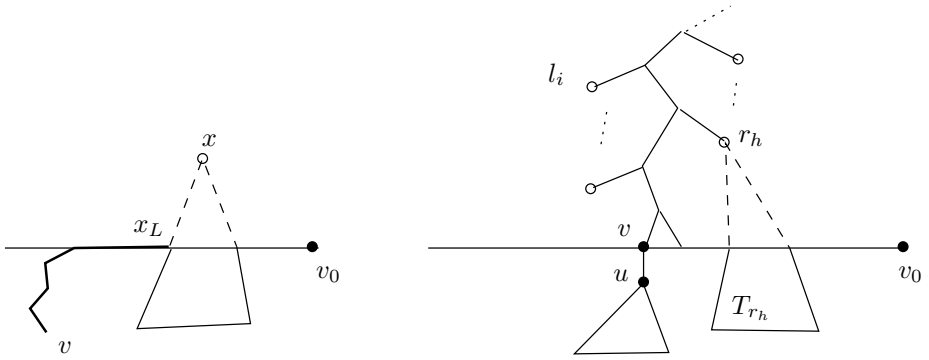


Figure 5.4: Computing the cost of the 1-median when v is the facility

From Lemma 5.1 we know that the optimal solution must be a vertex of the input tree, therefore we simply compute the 1-median cost with the facility at each vertex in the tree and return the one for which the value computed is smallest. The obvious algorithm to compute the cost for a given facility uses linear time for processing which leads to a quadratic 1-median algorithm. In the following paragraphs, we show that using the spine decomposition from Chapter 2 and pre-processing, we can compute the cost in logarithmic time.

Assume that at every node x of the SD we have the following information available:

(a) The clients of T_x are sorted in decreasing order of the distance from the path between the client and x_L, and the closest depot to the path. The sequence is z_1, z_2, ... $z_{|C(T_x)|}$ such that

$$d(\delta(x_L, z_j), \pi(x_L, z_j)) \geq d(\delta(x_L, z_{j'}), \pi(x_L, z_{j'}))$$

for any $j' > j$. Not that $\delta(x_L, z_j)$ is the closest depot to the path and is computed taking in consideration all depots in the whole tree. We will show later how to compute this efficiently.

(b) The weighted sum for the depot distance to path $\pi(z_j, x_L)$ for all clients starting with z_j to the last one in the order described above. We use subscript "L" because we compute another value $P_R(x, j)$ which returns the same weighted sum but using the ordering relative to paths $\pi(z_j, x_R)$.

$$P_L(x, j) = \sum_{i=j}^{|C(T_x)|} w(z_i) \cdot d\big(\delta(x_L, z_i), \pi(x_L, z_i)\big), \quad 1 \leq j \leq |C(T_x)|.$$

(c) The sum of the weights of the clients in the order described at (a) up to vertex z_j.

$$Q_L(x,j) = \sum_{i=1}^{j} w(z_i), \quad 1 \leq j \leq |C(T_x)|.$$

(d) The cost of all clients in T_x as if served by a facility at x_L but without adding the depot distance. To obtain the trip distance, one needs to add this value to the depot distance weighted sum.

$$M_L(x) = \sum_{i=1}^{|C(T_x)|} w(z_i) \cdot d(x_L, z_i).$$

A similar ordering of the clients in T_x is computed relative to the depot distance to the path from the client to x_R, $d\big(\delta(z_j, x_R), \pi(z_j, x_R)\big)$. Then, we define $P_R(x,j)$, $Q_R(x,j)$, and $M_R(x)$ exactly in the same way using the new ordering.

To compute the contribution of clients in T_x if served by some tree vertex v (Figure 5.4 (a)), we simply have to know the depot distance to path $\pi(v, x_L)$. Let this distance be d_{new},

$$d_{new} = d\big(\delta(v, x_L), \pi(v, x_L)\big).$$

Let j be the largest index in the ordering z_1, ... $z_{|C(T_x)|}$ of the client vertices in T_x relative to x_L for which the depot distance is larger than d_{new}, in other words, for which

$$d\big(\delta(z_j, x_L), \pi(z_j, x_L)\big) > d_{new}.$$

Then, the contribution of clients in T_x served by v is

$$2 \cdot \big(M_L(x) + w(T_x) \cdot d(v, x_L) + Q_L(x,j) \cdot d_{new} + P_L(x, j+1)\big). \tag{5.2}$$

Indeed, the first two terms represent the total cost for the trip between the clients and the facility and the last two the total cost for the trip between the optimal depot and the client-facility path. Note that we use d_{new} as depot distance for all clients for which the depot distance for the trip inside T_x is larger than d_{new}. Of course, if v is towards the root from x, we use the values P_R, Q_R and M_R in a similar way.

Now, we have all the ingredients needed to compute the 1-median cost when some vertex $v \in T$ is the facility. Consider Figure 5.4 (b) where $v \in T$ is the facility for which we need the cost. Let r_0, r_1, ... r_a and l_0, l_1, ... l_b be the SD nodes adjacent to path $\sigma(v, s_{SD})$. At each of these SD nodes including v, we use (5.2) to evaluate the contribution of the client vertices in the respective components, and we report the total as the result.

- Compute the SD and any information required for maintaining the sorted lists of vertices at any node.
- Traverse the SD bottom up; at every node x with children y and t do:
  1: Compute the sorted lists for x, compute values P, Q, and M for x.
  2: Traverse the sorted list of y and generate queries in t. Answer the queries by sequential search in the list at t.
  3: Store the result incrementally in an array indexed by the tree vertex corresponding to the query.
  4: Repeat steps 2 and 3 with the roles for y and t interchanged.
  5: Discard the lists stored at y and t.
- Traverse the array indexed by tree vertices and output the entry with smallest value.

Program 5.1: Algorithm to solve the 1-median collection depots problem in trees

Observe that the evaluation of (5.2) is done in constant time once value j is determined. If we use binary search with d_{new} over the ordering of clients, we spend $O(\log n)$ time at each SD node, and thus $O(\log^2 n)$ time for each vertex. This gives an algorithm with $O(n \log^2 n)$ running time. However, we can use the ideas from the previous chapter and replace the binary search step with sequential search. This can be done easily. Let y be the SD node sibling of x. Node x is used in the computation from (5.2) only when $v \in T_y$. But at y, we already have the sorted list of all vertices (we will compute the sorted list for all vertices and not only for the clients) relative to their depot distance to the path to either y_L or y_R. This gives a set of queries with d_{new} in sorted order.

Normally, if we keep the lists of sorted vertices at all nodes in the spine decomposition, the storage space of the algorithm is $O(n \log n)$. However, we can use the idea of Chapter 3 where we reduced the storage space of the k-median algorithm with a logarithmic factor by discarding lists that were no longer needed. In our present case, we can combine pre-processing with the computation steps as follows. Let x be the current SD node in the bottom-up pre-processing phase, and let y and t be its two children. Nodes y and t have all the required data available which we use to obtain the sorted list and values P, Q, and M for node x. Then, we simply generate all queries with $v \in T_y$ for t, and $v \in T_t$ for y, after which we discard the lists at y and t. The algorithm is sketched by Program 5.1.

Assuming that the pre-processing phase is executed in $O(n \log n)$ time with $O(n)$ space,

we can state the following result.

Theorem 5.1. *The* 1*-median collection depots problem in trees can be solved in* $O(n \log n)$ *time and* $O(n)$ *space.*

5.2.1 Preprocessing

We now show how to obtain the lists of tree vertices $z_j \in T_x$ for SD node x, sorted by $d(\delta(x_L, z_j), \pi(x_L, z_j))$. Values M_L, P_L, Q_L and M_R, P_R, Q_R are not difficult to obtain recursively once this ordering is determined. Similar operations are already illustrated in the previous chapters. Suppose we know the distance $d(\delta(z, z), z)$ for all vertices z in the tree. This value represents the distance to the closest depot from z. Then, we can use a bottom-up traversal to obtain our lists of distances, as described below.

Consider Figure 3.4 from page 46 where node x has children y and t with t towards the root. To obtain the list at x with x_L as reference, we can simply merge the lists of sorted vertices from y and t updating the depot distance as follows:
– If $z \in T_y$, use the entry from y with the depot distance unchanged because

$$d\big(\delta(z, y_L), \pi(z, y_L)\big) = d\big(\delta(z, x_L), \pi(z, x_L)\big).$$

– If $z \in T_t$, use the entry from t and update the depot distance,

$$d\big(\delta(x, y_L), \pi(z, x_L)\big) = \min\Big\{d\big(\delta(z, t_L), \pi(z, t_L)\big), d\big(\delta(y_L, y_R), \pi(y_L, y_R)\big)\Big\}.$$

In the last expression, we have to use the closest depot to the path from z to y_L, and since we only have this distance for the part $\pi(z, t_L)$, we need to explicitly compare the key of every entry with the distance to the depot closest to the remaining part, $\pi(y_L, y_R)$.

It follows immediately that the merging step is executed in time linear in the size of the lists. The depot $\delta(z, z)$ is needed to startup the bottom-up procedure and to maintain $\delta(x_L, x_R)$. It can be obtained easily, in amortized constant time for each z, without the spine decomposition.

The idea is to traverse the tree twice. The first time we walk bottom-up and compute, for every vertex $z \in T$, the value $\delta'(z, z)$ which returns the depot closest to z but only from subtree $T(z)$. Recall that $T(z)$ is the subtree rooted at z. Obviously, $\delta'(r_T, r_T) = \delta(r_T, r_T)$. The second traversal is performed from root to leaf, and we use $\delta(v, v)$ for the parent of z to find $\delta(z, z)$. It is an easy exercise to verify all the details.

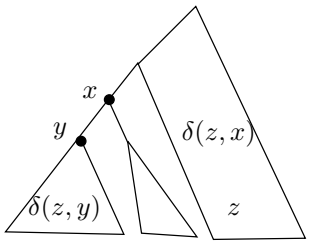


Figure 5.5: Obtaining the sorted trip distances for all vertices in total $O(n^2)$ time

5.3 k-median collection depots problem

In this section, we consider the general k-median collection depots problem in trees. We show that the algorithm of Tamir [99] can be used almost directly for solving the collection depots problem. This contrasts the k-median problem with positive/negative weights for which a direct translation is not possible.

Since none of our previous techniques are needed here, we will be brief in our exposition. We mentioned the algorithm of Tamir [99] in Section[1] 1.4.1. Two cost functions are used in the dynamic programming algorithm,

- $G(v_j, p, r_j^i)$ that returns the optimal cost in $T(v_j)$ if at most p facilities are located in $T(v_j)$ and at least one within distance r_j^i from v_j. r_j^i represents the i-th distance to a vertex in T from v_j when the distances are sorted in increasing order. Thus $r_j^0 = 0$ and represents the distance between v_j and itself.
- $F(v_j, p, r)$ is the cost in $T(v_j)$ if at most p facilities are in $T(v_j)$ and an external median is at distance exactly r from v_j.

The collection depots k-median problem has a strong similarity with the usual k-median problem. The difference lies in the distance function used. For the classic k-median problem, the distance between two vertices is simply the vertex distance. In the collection depots problem, it is the trip distance with one of the vertices being the facility. Note that the trip distance is symmetric and monotone but not uniformly monotone. We call the trip distance monotone because it never decreases as the client moves away from the serving facility. The trip distance is not uniformly monotone because given two facilities placed on the same side of the client vertex, as the client moves away from both facilities, the

[1]The description of the functions in Section 1.4.1 is a bit different because we were not interested in the details at that point.

difference in length between the trips using the two facilities is not a monotone function. Figure 5.3 provides an example. The difference is constant if pure distance is used instead of trip distance. However, in Tamir's algorithm, the monotonicity property is not needed since all pairs of distances are explicitly computed and sorted. In order to apply the algorithm on the collection depots problem, we need to compute and sort the trip distances originating from every vertex in the tree. Then, we replace parameter r_j^i above with the sorted trip distance and everything else is carried out in the usual manner. Without detailing the steps of the dynamic programming algorithm, we just discuss how to obtain the sorted list of trip distances.

Notice that in total $O(n^2)$ time, we can compute all pairwise trip distances by traversing the tree n times, every time using a different vertex as origin. We call $v \in T$ an origin if we compute $r(v, z)$, $\forall z \in T$. In $O(n \log n)$ time we can sort directly the list of distances with one vertex as origin, which gives a total $O(n^2 \log n)$ time for the entire process. However, the sorting time would dominate the computation time when k is constant, and we wish to avoid this.

To sort all lists in total $O(n^2)$ time, we use merging in a similar way to Tamir [99]. First we compute and sort the trip distances at the root node r_T directly as in the previous paragraph. Assume we know the sorted trip distance at node x and we want the sorted list for one of its children, y, as in Figure 5.5. Consider vertices $z \in T \setminus T(y)$ for which the depot for the trip from x belongs to $T \setminus T(y)$, but the new depot for the trip from y is in $T(y)$. Denote this set by V_{xy}. The vertices from V_{xy} use the same new depot from $T(y)$ with the same depot distance equal to $d(\delta(y, y), y)$. Their relative trip order is given then by the distance to x order in $T \setminus T(y)$. This distance order can be computed with the algorithm of Tamir [99]. Therefore, the relative order of vertices in V_{xu} is known.

Consider now elements from set V_{yx} of vertices in $T(y)$ whose depot for the trip from x lies in $T \setminus T(y)$, but the new depot for the trip originating at y is in $T(y)$. These vertices used the same depot $\delta(x, x)$ from $T \setminus T(y)$, but now, each might use some other depot within $T(y)$. Their relative order is given by the trip distance order in $T(y)$ that uses only depots from $T(y)$. This distance is not difficult to compute bottom-up in total $O(n^2)$ time for all vertices $y \in T$. Therefore, the relative order of vertices in V_{yx} is also known.

For all other vertices, the depot distance does not change, and the relative order is the same as that for parent node x. We thus have, at node y, three sorted lists which can be merged in linear time to give the sorted trip distance for vertex y. This gives a total of

$O(n^2)$ time to obtain the sorted lists at every vertex in the tree. We can therefore state the following result.

Theorem 5.2. *The k-median collection depots location problem can be solved directly by the dynamic programming algorithm of Tamir [99] in $O(kn^2)$ time and $O(n^2)$ space.*

The storage space analysis is similar to that from Chapter 3. We do not need to maintain the cost functions at all nodes in the tree during the bottom-up traversal. We simply discard the storage space for the cost functions of the children vertices once we used their information entirely. This means that the storage used by the cost functions at any moment in time is $O(kn)$, which is dominated by the memory requirement for the lists of sorted trip distances.

5.4 Conclusion

In this chapter, we considered the collection depots location problem in trees. Using the same set of techniques as for earlier applications, we propose an algorithm for the location of one facility. We also show that the k-median algorithm for the simple distance function in a tree can be used directly to solve collection depots instances.

For the 1-median collection depots problem, our solution has the same running time as the algorithm for the 1-center collection depots problem. A gap still exists between the performance of the best known algorithm for collection depots and the simple pure 1-median and 1-center linear time algorithms. For the k-median problem, both the collection depots generalization and the pure distance problem can be solved with the same algorithm. This indicates that the dynamic programming algorithm is quite powerful since it can accommodate a wide palette of distance functions. Unfortunately, the undiscretized version of the k-median algorithm cannot be used directly with collection depots. The trip distance depends on two components with no apparent relationship. To predict the behaviour of the optimal solution by pre-computing useful information, leads quickly to complexity quadratic in n.

Chapter 6

Conclusion

In this thesis we study several optimization problems from location science. These problems aim to establish an optimal placement of objects called facilities in order to minimize a certain objective function. The setting is a tree network where vertices are assigned weights, and the objective function to be minimized is based on the weighted tree distance. The cardinality of the set of facilities to be located is constrained.

We propose new algorithms to solve these problems that either improve or supplement the results known in the literature, or are the first ones for particular instances. Table 6.1 summarizes their performance.

How significant are these results? With our algorithm for the k-median problem in trees, we answer a question remained open since 1996 when Tamir [99] gave a careful analysis of the dynamic programming algorithm and showed that it runs in time $O(kn^2)$. Until now, nobody has given algorithms sub-quadratic in n when the number of facilities is more than 2, or when the trees used for input are restricted to particular classes. Of course, we exclude paths from this discussion because on paths, many optimization problems have very efficient algorithms. For example, Hassin and Tamir [56] and later Auletta *et al.* [8] proposed an $O(kn)$ dynamic programming algorithm for the k-median on paths (the real line). In fact, we were able to extend the methods used for the k-median problem on paths to arbitrary trees.

Our general idea is simple. We use a decomposition of trees that we named *the spine decomposition* which enables us to apply intuitive geometric techniques in the processing of the information managed by our dynamic programming algorithms. These techniques are nothing but computation of convex hulls and tangent points. Similar recipes have been

	Running time	Storage space	Prev. running time
k-median, T–arbitrary	$O(n \log^{k+2} n)$	$O(n \log^{k} n)$	
k-median, T–balanced	$O(n \log^{k-1} n)$	$O(n \log^{k-2} n)$	$O(kn^2)$ [99]
k-median, T–directed	$O(n \log^{k-1} n)$	$O(n \log^{k-2} n)$	
3-median, T–arbitrary	$O(n \log^3 n)$	$O(n \log n)$	$O(n^2)$ [99]
2-median (+/– MWD)	$O(n \log n)$	$O(n \log n)$	$O(n \log^2 n)$ [20]
2-median (+/– WMD)	$O(nh \log^2 n)$	$O(nh \log n)$	$O(n^3)$ [21]
or	$O(nh \log^3 n)$	$O(n \log n)$	
1-median (collect.)	$O(n \log n)$	$O(n)$	unknown
k-median (collect.)	$O(kn^2)$	$O(n^2)$	the same [100]

Table 6.1: Summary of problems solved and algorithm complexity for tree T and for constant k

successfully used in solving optimization problems on the line, as in the afore mentioned paper of Hassin and Tamir. With this combination of spine decomposition and geometric techniques, we were able to improve several results on the k-median problem in trees and two of its generalizations. This makes us believe that many other problems could be solved efficiently by the same approach.

The spine decomposition is a structure we designed to overcome several disadvantages of the centroid decomposition. Although it can be argued that many other decompositions can be used instead, we feel that the particular structure of the spine decomposition makes certain algorithms simpler. In fact Boland [18] has independently proposed an almost identical data structure and has applied it to the circular ray shooting problem. Therefore, there is evidence that the spine decomposition could be successfully used with many other algorithms not necessarily limited to optimization problems in trees.

Another feature of the spine decomposition is that both the structure of the tree and the relationship between the components of the decomposition are well represented. For example, given any two nodes in the decomposition, we can easily determine the tree path between the components of the two nodes. Conversely, for any tree vertices, it is easy to view the succession of adjacent decomposition nodes starting at a component containing one vertex and ending at the component containing the other. The same information is not so clearly represented in the centroid decomposition for example, where the relationship

between the structure of the decomposition and that of the tree is more blurred. The top-trees of Holm [61] overcome many of the problems of the centroid decomposition, however they are more suitable for applications where the parent-child relationship in the tree is not essential. The spine decomposition can be immediately used with both rooted and un-rooted trees as demonstrated by our algorithm for the k-median problem in directed trees from Chapter 3. Finally, it is important to note that the spine decomposition can be constructed easily in linear time as opposed to the linear time construction of the centroid decomposition which is much more complex [48].

Regarding the algorithm for k-median problems in trees, our bound on both the running time and the storage space is based on the recurrence relation obtained from the recursive computation of cost functions controlled by the dynamic programming framework. This bound is proportional to a constant exponential in k which explains the need to assume that parameter k is not part of the input. However, a conjecture by Chrobak *et al.* [28] states that the cost functions have a much tighter bound than the one obtained through the recurrence relation, in fact, it is believed that their size is linear in n. If this is true, then the complexity of our algorithms would be drastically improved. Our intuition favours a positive answer for Chrobak's conjecture, however, we were not able yet to settle this issue.

We should also mention that our k-median algorithm is practical. Although it is more complex than the classic $O(kn^2)$ dynamic programming algorithm, its implementation does not pose significant challenges, except perhaps if one implements fractional cascading. The calculation of the recursive functions is direct and solving the j-median subproblem amounts to a repeated application of convex hull computations. The computation of the spine decomposition is straightforward, but our undiscretized dynamic programming framework is flexible and can be used with other decompositions of trees too. For example, we argued in Chapter 2 that the centroid decomposition is not suitable because a logarithmic number of versions of the same cost function must be calculated for every component. This also means that, if one doesn't mind at least a logarithmic blow-up of storage space and running time, the centroid decomposition can be used as well.

Directions for future research

The results presented in this thesis open several possibilities for further study. First, it would be extremely useful if we could answer Chrobak's conjecture either in the affirmative or in the negative. Second, the two k-median generalizations that we studied here are relatively

new problems, and many issues are still open. For instance, no algorithms are known for the computation of k medians in trees for positive and negative weights, or in more general graphs. It is also not clear what changes are necessary in the different algorithms that solve the usual k-median problem in order to accommodate negative weights in both WMD and MWD formulations. It might also be possible to improve the running time of the WMD 2-median problem which currently depends on the height of the tree.

The k-median collection depots problem is not harder than the usual k-median problem, at least when speaking of Tamir's dynamic programming algorithm. However the computation of one median is still super-linear, as opposed to the simple linear time algorithm for the classic problem. Is a linear time algorithm possible for the one facility collection depots problem? At a more general level, many issues are still open. Is it possible to design an algorithm linear in n for the 1-center problem? Can we develop efficient algorithms for other classes of graphs, such as interval or circular arc graphs? Can we design sub-quadratic algorithms for the k-median collection depots problem in trees? For the last question, we feel the task is much more difficult than designing sub-quadratic algorithms for the positive/negative problem. The reason for this statement is that the cost of the solution in the collection depots problem is determined by two parameters independent of each other, the client-facility distance and the depot-path distance. Any undiscretized dynamic programming approach we could think of so far, involved both of these parameters as arguments of cost functions and we couldn't avoid the quadratic complexity arising from here. Perhaps by considering only two facilities to locate, one can achieve sub-quadratic performance more easily.

Our results on the k-median problem in trees could also further the research towards solving the k-median more efficiently on other classes of graphs, such as the graphs with bounded tree-width. These are graphs with a structure similar to a tree, the similarity being controlled by a parameter of the graph, the tree-width. The idea to exploit the structure of graphs that are not too different from trees is not new. Gurevich *et al.* [49] looked at solving the k-cover problem on graphs and proposed an algorithm whose running time depends on the number of edges that should be removed to transform the graph into a tree. Their idea is to recursively decompose the graph into connected components assembled into a tree-like structure on which the k-cover algorithm is applied. The components are obtained by removing vertices with large degree. Then, all possible interactions that could take place through the removed vertex are enumerated in order to compute the solution on the original

graph from the solution of the tree decomposition (the tree decomposition mentioned here is a decomposition of a *general graph* into components). A similar approach but using the tree decomposition of a graph in the sense of Robertson and Seymour [89], could be also used for solving the k-median problem. One possibility is to combine our framework for the k-median problem in trees with a method inspired from the algorithm of Chaudhuri and Zaroliagis [25] for computing shortest paths in trees with bounded tree-width.

Finally, we believe that the simple techniques we used for solving the k-median in trees can be used on optimization problems with a much broader application range. We hope that the results from this thesis will have impact on areas not necessarily limited to facility location.

Bibliography

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

[2] S. Alstrup, J. Holm, K. de Lichtenbarg, and M. Thorup, "Minimizing diameters of dynamic trees," in *Proc. ICALP'97*, pp. 270–280, 1997.

[3] S. Alstrup, J. Holm, and M. Thorup, "Maintaining center and median in dynamic trees," in *In Proc. 7-th SWAT*, vol. 1851 of *LNCS*, pp. 46–56, 2000.

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "TSP cuts which do not conform to the template paradigm," in *Computational combinatorial optimization: optimal or provably near optimal solutions* (M. Jünger and D. Naddef, eds.), vol. 2241 of *Lecture Notes in Computer Science*, pp. 261–303, Springer, 2001.

[5] 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)*, pp. 106–113, 1998.

[6] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Mungala, and V. Pandit, "Local seach heuristic for k-median and facility location problems," in *Proc. 33rd Annual ACM Symposium on Theory of Computing*, pp. 21–29, 2001.

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

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

[9] R. Benkoczi and B. Bhattacharya, "Spine tree decomposition," Tech. Rep. 09, School of Computing Science, Simon Fraser University, Canada, 1999.

[10] R. Benkoczi and B. Bhattacharya, "The 2-median problem on a tree with positive and negative weights." unpublished, 2004.

[11] R. Benkoczi and B. Bhattacharya, "New results regarding the k-median problem on trees." submitted to FOCS, 2004.

[12] R. Benkoczi, B. Bhattacharya, and D. Breton, "Efficient computation of 2-medians in a tree network with positive/negative weights." to appear in Discrete Mathematics, 2004.

[13] R. Benkoczi, B. Bhattacharya, M. Chrobak, L. Larmore, and W. Rytter, "Faster algorithms for k-median problems in trees," in *Proc. 28th International Symposium on Mathematical Foundations of Computer Science* (B. Rovan and P. Vojtáš, eds.), vol. LNCS 2747, pp. 218–227, 2003.

[14] R. Benkoczi, B. Bhattacharya, and Q. Shi, "Minsum and minmax collection depots problems in trees." unpublished, 2004.

[15] O. Berman, Z. Drezner, and G. O. Wesolowsky, "The collection depots location problem on networks," *Naval Research Logistics*, vol. 49, no. 1, pp. 15–24, 2002.

[16] O. Berman and R. Huang, "The minisum collection depots location problem with multiple facilities on a network." submitted to Jnl. Operational Research Society, 2003.

[17] B. Bhattacharya and R. Benkoczi, "On computing the minimum bridge between two convex polygons," *Information Proc. Letters*, vol. 79, no. 5, pp. 215–221, 2001.

[18] R. Boland, *Polygon visibility decompositions with applications*. PhD thesis, University of Ottawa, Ottawa, Canada, 2002.

[19] D. Bovet and P. Crescenzi, *Introduction to the Theory of Complexity*. Prentice-Hall, 1993.

[20] D. Breton, "Facility location optimization problems in trees," Master's thesis, School of Computing Science, Simon Fraser University, Canada, 2002.

[21] R. Burkard, E. Çela, and H. Dollani, "2-medians in trees with pos/neg weights," *Discrete Applied Mathematics*, vol. 105, pp. 51–71, 2001.

[22] R. Burkard and J. Krarup, "A linear algorithm for the pos/neg-weighted 1-median problem on a cactus," *Computing*, vol. 60, pp. 193–215, 1998.

[23] 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)*, pp. 378–388, 1999.

[24] 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)*, pp. 1–10, 1999.

[25] S. Chaudhuri and C. Zaroliagis, "Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms," *Algorithmica*, vol. 27, pp. 212–226, 2000.

[26] B. Chazelle and L. Guibas, "Fractional cascading: I. A data structuring technique," *Algorithmica*, vol. 1, no. 2, pp. 133–162, 1986.

[27] B. Chazelle and L. Guibas, "Fractional cascading: II. Applications," *Algorithmica*, vol. 1, no. 2, pp. 163–191, 1986.

[28] M. Chrobak, L. Larmore, and W. Rytter, "The k-median problem for directed trees," in *Proc. 26th International Symposium on Mathematical Foundations of Computer Science (MFCS'01)*, no. 136 in Lecture Notes in Computer Science, pp. 260–271, 2001.

[29] R. Cole, M. Farach-Colton, R. Hariharan, T. Przytycka, and M. Thorup, "An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees," *SIAM Journal on Computing*, vol. 30, pp. 1385–1404, 2000.

[30] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time," *Algorithmica*, vol. 3, pp. 329–346, 1988.

[31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. MIT Press, 1999. ISBN 0-262-03141-8.

[32] G. Cornuéjols, M. Fisher, and G. Nemhauser, "Location of bank accounts to optimize float: an analytic study of exact and approximate algorithms," *Management Science*, vol. 23, pp. 789–810, 1977.

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

[34] E. Demaine, F. Fomin, M. T. Hajiaghayi, and D. Thilikos, "Fixed-parameter algorithms for the (k, r)-center in planar graphs and map graphs," in *Lecture Notes in Computer Science*, vol. 2719, (Heidelberg), pp. 829–844, Springer-Verlag, 2003.

[35] R. Diestel, *Graph Theory*, vol. 173 of *Graduate Texts in Mathematics*. New York: Springer-Verlag, 2000.

[36] R. Downey and M. Fellows, *Parameterized complexity*. Heidelberg: Springer-Verlag, 1998.

[37] Z. Drezner and G. O. Wesolowsky, "On the collection depots location problem," *European Journal of Operational Research*, vol. 130, no. 3, pp. 510–518, 2001.

[38] U. Feige, "A threshold of $\ln n$ for approximating set cover," in *Proceedings of the 28-th ACM Symposium on Theory of Computing*, pp. 314–318, 1996.

[39] G. Frederickson, "Parametric search and locating supply centers in trees," in *Proceedings 2-nd Workshop on Algorithms and Data Structures*, vol. 519 of *Lecture Notes in Computer Science*, (Ottawa), pp. 299–319, Springer, 1991.

[40] G. Frederickson and D. Johnson, "Finding k-th paths and p-centers by generating and searching good data structures," *Journal of Algorithms*, vol. 4, pp. 61–80, 1983.

[41] Z. Galil and K. Park, "A linear time algorithm for concave one-dimensional dynamic programming," *Information Processing Letters*, vol. 33, pp. 309–311, 1989.

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

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

[44] A. Goldman, "Optimal center location in simple networks," *Trans. Sci.*, vol. 5, pp. 212–221, 1971.

[45] A. Goldman and C. Witzgall, "A localization theorem for optimal facility placement," *Trans. Sci.*, vol. 1, pp. 106–109, 1970.

[46] T. Gonzalez, "Clustering to minimize the maximum intercluster distance," *Theoretical Computer Science*, vol. 38, pp. 293–306, 1985.

[47] D. Granot and D. Skorin-Kapov, "On some optimization problems on k-trees and partial k-trees," *Discrete Applied Mathematics*, vol. 48, no. 2, pp. 129–145, 1994.

[48] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, "Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons," *Algorithmica*, vol. 2, no. 2, pp. 209–233, 1987.

[49] Y. Gurevich, L. Stockmeyer, and U. Vishkin, "Solving NP-hard problems on graphs that are almost trees and an application to facility location problems," *Journal of the ACM*, vol. 31, no. 3, pp. 459–473, 1984.

[50] S. Hakimi, "Optimum locations of switching centers and the absolute centers and medians of a graph," *Operations Research*, vol. 12, pp. 450–459, 1964.

[51] S. Hakimi, "Optimum distribution of switching centers in a communication network and some related graph-theoretic problems," *Operations Research*, vol. 13, pp. 450–459, 1965.

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

[53] G. Handler, "Minimax network location theory and algorithms," Tech. Rep. 107, MIT, Cambridge, Mass., Nov. 1974.

[54] G. Handler, "Finding two centers of a tree: the continuous case," *Transportation Science*, vol. 12, no. 2, pp. 93–106, 1978.

[55] R. Hassin and A. Tamir, "Efficient algorithms for optimization and selection on series-parallel graphs," *SIAM Journal of Algebraic Discrete Methods*, vol. 7, pp. 379–389, 1986.

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

[57] J. Hershberger and S. Suri, "A pedestrian approach to ray shooting: shoot a ray, take a walk," *Journal of Algorithms*, vol. 18, pp. 403–431, 1995.

[58] D. Hirschberg and L. Larmore, "The least weight subsequence problem," *SIAM Journal on Computing*, vol. 16, pp. 628–638, 1987.

[59] D. Hochbaum, "Various notions of approximations: good, better, best, and more," in *Approximation Algorithms for NP-hard Problems*, pp. 346–398, Boston, MA: PWS Publishing Company, 1997.

[60] D. Hochbaum and D. Shmoys, "A unified approach to approximation algorithms for bottleneck problems," *Journal of the ACM*, vol. 33, pp. 533–550, 1986.

[61] J. Holm and K. de Lichtenberg, "Top-trees and dynamic graph algorithms," Tech. Rep. 17, Univ. of Copenhagen, Dept. of Computer Science, 1998.

[62] W. Hsu, "The distance-domination numbers of trees," *Operation Research Letters*, vol. 1, pp. 96–100, 1982.

[63] W. Hsu and G. Nemhauser, "Easy and hard bottleneck location problems," *Discrete Applied Mathematics*, vol. 1, pp. 209–216, 1979.

[64] K. Jain and V. Vazirani, "Primal-dual approximation algorithms for metric facility location and k-median problems." Manuscript, March 1999.

[65] A. Kang and D. Ault, "Some properties of a centroid of a free tree," *Information Processing Letters*, vol. 4, pp. 18–20, 1975.

[66] O. Kariv and S. Hakimi, "An algorithmic approach to network location problems I: The p-centers," *SIAM Journal on Applied Mathematics*, vol. 37, pp. 513–538, 1979.

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

[68] A. Kolen and A. Tamir, "Covering problems," in *Discrete Location Theory* (B. Mirchandani and R. Francis, eds.), pp. 263–304, Wiley-Interscience, 1990.

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

[70] M. Labbe, D. Peeters, and J. Thisse, *Location on networks*, vol. 8 of *Handbooks in Operations Research and Management Science*. Elsevier, 1995.

[71] L. Larmore and B. Schieber, "On-line dynamic programming with applications to the prediction of rna secondary structure," in *Proc. 1st Ann. ACM Symposium on Discrete Algorithms (SODA)*, pp. 00–00, 0000.

[72] H. Lenstra, "Integer programming with a fixed number of variables," *Mathematics of Operations Research*, vol. 8, no. 4, pp. 538–548, 1983.

[73] 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)*, pp. 485–495, 1998.

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

[75] J.-H. Lin and J. Vitter, "Approximation algorithms for geometric median problems," *Information Processing Letters*, vol. 44, pp. 245–249, 1992.

[76] J.-H. Lin and J. Vitter, "ϵ-approximations with minimum packing constraint violation," in *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp. 771–782, 1992.

[77] A. Lozano and J. Mesa, "Location of facilities with undesirable effects and inverse location problems: a classification," *Studies in Locational Analysis*, vol. 14, pp. 253–291, 2000.

[78] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *Journal of the ACM*, vol. 31, pp. 114–127, 1984.

[79] N. Megiddo and A. Tamir, "New results on the complexity of p-center problems," *SIAM J. on Computing*, vol. 12, pp. 751–758, 1983.

[80] N. Megiddo and K. J. Supowit, "On the complexity of some common geometric location problems," *SIAM Journal on Computing*, vol. 13, pp. 182–196, 1984.

[81] E. Meyr, H. Promel, and A. Steger, *Lectures on Proof Verification and Approximation Algorithms*, vol. 1367 of *Lecture Notes in Computer Science*. Springer, 1998.

[82] E. Minieka, "The m-center problem," *SIAM Review*, vol. 12, pp. 138–139, 1970.

[83] P. Mirchandani and A. Oudjit, "Localizing 2-medians on probabilistic and deterministic tree networks," *Networks*, vol. 10, pp. 329–350, 1980.

[84] M. Overmars and J. van Leeuwen, "Maintenance of configurations in the plane," *J. Comput. Syst. Sci.*, vol. 23, pp. 166–204, 1981.

[85] C. Papadimitriou, *Computational Complexity*. Addison-Wesley, 1994.

[86] J. Plesnik, "On the computational complexity of centers locating in a graph," *Aplikace Matematiky*, vol. 25, pp. 445–452, 1980.

[87] J. Plesnik, "A heuristic for the p-center problem in graphs," *Discrete Applied Mathematics*, vol. 263-268, pp. 263–268, 1987.

[88] F. Preparata and M. Shamos, *Computational geometry*. New York: Springer-Verlag, 1985.

[89] N. Robertson and P. Seymour, "Graph minors. I. excluding a forest," *J. Combin. Theory Ser. B*, vol. 35, pp. 39–61, 1983.

[90] A. Rosenthal and J. A. Pino, "A generalized algorithm for centrality problems on trees," *Journal of the ACM*, vol. 36, no. 2, pp. 349–361, 1989.

[91] G. Sabidussi, "The centrality index of a graph," *Psychometrika*, vol. 31, pp. 581–603, 1966.

[92] R. Shah, *Undiscretized dynamic programming and ordinal embeddings*. PhD thesis, The State University of New Jersey – Rutgers, 2002.

[93] R. Shah and M. Farach-Colton, "Undiscretized dynamic programming: faster algorithms for facility location and related problems on trees," in *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pp. 108–115, 2002.

[94] R. Shah, S. Langerman, and S. Lodha, "Algorithms for efficient filtering in content-based multicast," in *Proc. 9th Annual European Symposium on Algorithms (ESA)*, pp. 428–439, 2001.

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

[96] H. Sherali and F. Nordai, "A capacitated balanced 2-median problem on a tree network with a continuum of link demands," *Transportation Science*, vol. 22, no. 1, pp. 70–73, 1988.

[97] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences*, vol. 26, pp. 362–391, 1983.

[98] A. Tamir, "Improved complexity bounds for center location problems on networks by using dynamic data structures," *SIAM Journal of Disrete Mathematics*, vol. 3, pp. 377–396, 1988.

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

[100] A. Tamir, "The 1-center collection depots problem in trees," 2004. personal communication.

[101] A. Tamir, "The k-median problem in unweighted balanced binary trees," 2004. private communication.

[102] A. Tamir, D. Pérez-Brito, and J. Moreno-Pérez, "A polynomial algorithm for the p-centdian problem on a tree," *Networks*, vol. 32, pp. 255–262, 1998.

[103] A. Tamir and E. Zemel, "Locating centers on a tree with discontinuous supply and demand regions," *Mathematics of Operations Research*, vol. 7, no. 2, pp. 183–197, 1982.

[104] A. Tamir and N. Halman, "One-way and round-trip center location problems." submitted, 2003.

[105] B. Tansel, R. Francis, and T. Lowe, "Duality: Covering and constraining p-center problems on trees," in *Discrete Location Theory* (B. Mirchandani and R. Francis, eds.), pp. 349–386, Wiley-Interscience, 1990.

[106] M. Thorup, "Quick k-median, k-center, and facility location for sparse graphs," in *28th International Colloqium on Automata, Languages and Programming*, vol. 2076 of *Lecture Notes in Computer Science*, (Crete, Greece), pp. 249–260, 2001.

[107] 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*, vol. 74, pp. 81–88, 2000.

[108] R. Wilber, "The concave least-weight subsequence problem," *Journal of Algorithms*, vol. 9, pp. 418–425, 1988.

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

[110] B. Zelinka, "Medians and peripherians of trees," *Arch. Math. (Brno)*, vol. 4, pp. 87–95, 1968.

Index