On the Approximability of Location and Network Design Problems

Dissertation zur Erlangung
des naturwissenschaftlichen Doktorgrades
der Bayerischen Julius-Maximilians-Universität Würzburg

vorgelegt von

Sven Oliver Krumke

aus Frankfurt am Main

Würzburg 1996

Dissertation zur Erlangung
des naturwissenschaftlichen Doktorgrades
der Bayerischen Julius-Maximilians-Universität Würzburg

Eingereicht am: 16. Dezember 1996

bei der Fakultät für Mathematik und Informatik

1. Gutachter: Univ.-Prof. Dr. H. Noltemeier

2. Gutachter: Univ.-Prof. Dr. J. Stoer

Tag der mündlichen Prüfung: 10. April 1997

Abstract

This thesis investigates the approximability of several NP-hard location and network design problems. We present polynomial time approximation algorithms with a performance guarantee on the worst-case quality of the output solution. We also establish lower-bound results (non-approximability results) which show that the performances of our algorithms are essentially the best unless $\mathsf{P} = \mathsf{NP}$ or $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$.

The problems considered in this thesis are formulated as multicriteria or k-criteria problems. A k-criteria optimization problem Π on a weighted graph is defined by specifying k minimization objectives, $f_1, \ldots, f_k$, and a membership requirement in a class of weighted subgraphs Γ (not necessarily weighted exactly the same way as the original graph). The problem specifies budget values $F_2, \ldots, F_k$ as upper bounds on the objectives $f_2, \ldots, f_k$. The goal is to find a subgraph from Γ obeying the constraints on the objectives $f_2, \ldots, f_k$ having minimum possible value for f_1. The performance of an approximation algorithm for a multicriteria problem is measured in a natural way as a vector: An algorithm with performance $(\alpha_1, \ldots, \alpha_k)$ finds a solution of objective function value at most α_1 times the optimum value and violating the constraints on f_i $(i \geq 2)$ at most by the respective factors α_i.

The main tools for deriving the approximation algorithms are relaxed decision procedures, solution based-decomposition methods and the application of parametric search techniques.

Da steh ich nun, ich armer Tor!
Und bin so klug als wie zuvor;
Heiße Magister, heiße Doktor gar,
Und ziehe schon an die zehen Jahr
Herauf, herab und quer und krumm
Meine Schüler an der Nase herum –
Und sehe, daß wir nichts wissen können!
Das will mir schier das Herz verbrennen.

Goethe, Faust (1806)

Contents

1

Introduction

Location and Network Design Problems

Location problems deal with the notion of making an optimal choice within a spatial context, such as deciding where to place a number of restaurant franchises, factories, hospitals, police stations or computer services. In the case of *network location problems* treated in this thesis facilities can be placed on the vertices of a graph. The task of solving a location problem is termed to make a *locational decision*. The optimality of a decision depends on the specific problem and, naturally, there is a wide range of criterions used.

Network design problems have been triggered with the advent of high-speed networks. Communication companies are approached by their clients with the requirement to interconnect cities housing the client's offices such as banks with a high transaction rate between the branches. One of the main concerns of the client is to build a communication network of minimum cost. Like network location problems, network design problems are formulated appropriately with the help of a graph theoretic framework. Also, the objective of optimization varies with the particular problem.

Many network design problems address the task of constructing a new network from scratch. In practice, it is often the case that already parts of a network or even a complete network does exist. In this case it is meaningful to ask how one can improve the old network by adding connections or increasing capacities rather than wasting the money already invested in the network. Particularly, with the advent of new communication technologies this aspect of network design has become rather important.

Quite a few problems which arise in practice are a synthesis of location and network design problems. As an example consider the task of placing computing service facilities in a communication network with the objective to speed up the communication between the existing nodes in the network. If a facility is placed at a certain node of the network, the communication links incident with that particular node will be sped up due to the node's larger capacity. On the other hand, only a limited number of new service facilities are available. The last requirement is that the communication in the network after installing the service facilities should only take place on a subnetwork of simple structure (e.g. a tree). This is motivated by the fact that in the long run maintaining a simple network will be less expensive. The optimization goal in the above example now consist of two aspects: One needs to make a locational decision where to place the new facilities and one must select a "good" subset of communication links in the modified network.

Intractability and Approximability

As motivated above, location and network design problems are of great practical interest. Unfortunately, it can be shown for many of these important problems that determining an optimal solution is extremely time consuming due to their computational complexity. The class of NP-complete problems represents a large collection of such problems, which are all related in the sense that a polynomial time solution of one of them implies the polynomial time solvability of the whole class. Up to now, no polynomial time algorithm for an NP-complete problem is known.

It is widely believed that NP-completeness, or, more general, NP-hardness of a problem is a certificate of intractability. Thus, especially for problems of practical interest one would like to compromise on the quality of the solution for the sake of computing a suboptimal solution quickly. This leads in a natural way to *approximation algorithms*. An approximation algorithm runs in polynomial time and outputs a solution with small error. The most common (but not the only meaningful) way to measure this deviation from optimality is by means of a multiplicative error. The maximum of the errors over all instances of a problem is called the *performance guarantee* or *performance ratio* of the algorithm.

The design of approximation algorithms with good performance guarantees has been a major focus of research for the last years. Sadly, for some of the NP-hard problems it is even computationally expensive to calculate a suboptimal solution where the error is bounded from above by a certain quantity. In these cases solving the problem approximately with the specific performance is as expensive as solving the problem exactly. Commonly, such lower bounds are referred to as *non-approximability* or *hardness results*. They vary from very weak to extremely strong depending on which degree of approximability they rule out.

Often it is meaningful to capture the quality of a solution to a location or network design problem by more than a single measure. Basically, one would desire to optimize all the objectives associated with a problem. A natural way to formulate such problems is to specify bounds on all but one of the objectives and to optimize the remaining objective subject to these budget constraints. These problems are commonly referred to as *multicriteria problems*. The corresponding approximation algorithms are called *multicriteria approximation algorithms*. Their performance is measured in terms of how much the corresponding approximate solution violates optimality and the constraints. In case that there is only one objective, we obtain the usual notion of a performance guarantee which was mentioned earlier.

One might ask whether *restricting* a hard problem to a simple subset of instances will make the problem easier. In the context of location and network design problems formulated with the help of a graph-theoretic framework this means to consider the problem restricted to a simple class of graphs. The answer whether such a restriction renders a hard problem to be solvable efficiently depends heavily on the problem. Sometimes a computationally complex problem can be solved in polynomial time, while in other cases the problem remains intractable.

Roadmap and Summary of Results

In this thesis, we investigate the *approximability of location and network design problems*. The main contribution is to devise efficient algorithms with performance guarantees that are typically a constant or a slow-growing function of the input size (such as the logarithm) and to show that the performances provided by these algorithms are nearly best-possible in the following sense: no polynomial time algorithm can provide a substantially better approximation, unless the complexity classes P and NP coincide or something nearly equivalent is true.

We also consider the restriction of location and network design problems to classes of graphs with a particularly simple or helpful structure. Examples of such a class are trees and graphs of bounded treewidth. The treewidth measures the complexity of a graph in a certain sense or how far the graph is away from being a tree itself. For graphs of bounded treewidth we can often improve the approximation results due to their underlying tree-like structure.

The presentation is organized as follows. In Chapter 2, we start with some basic terminology used throughout this thesis. Chapter 3 contains an introduction to approximation algorithms. We provide a framework for formulating multicriteria optimization problems on graphs and define the notion of the performance guarantee of an approximation algorithm. We also show that our framework is indeed meaningful: A good approximation algorithm for a multicriteria problem transforms into good approximation algorithms for related unicriterion problems where the objective function is a functional combination of the objectives of the multicriteria problem.

We start our analysis of location problems in Chapter 4. We study the approximability of a *generalization of the* p*-center problem*, which is a classical location problem. The goal of the problem is to select a number p of vertices in an edge-weighted network as facilities such as to minimize the maximum distance of a client to its α nearest of the p centers. We show that if the edge-weights do not obey the triangle inequality, then the problem is extremely hard to approximate. For the case that the triangle inequality is satisfied, we provide a fast approximation algorithm. The results obtained for the problem, which is called p-CENTER$^{(\alpha)}$, are summarized in Table 1.1. We note that for a problem Π we use Δ-Π to denote the subset of instances where the weights on the edges obey the triangle inequality.

Problem	Hardness Result	Performance Guarantee	Running Time
p-CENTER$^{(\alpha)}$	Not approximable within $\mathcal{O}(f(n))$, where f is any polynomial time computable function	—	—
Δ-p-CENTER$^{(\alpha)}$	Not approximable within $2-\varepsilon$ for any fixed $\varepsilon > 0$	2 for $\alpha = 1$ 4 for $\alpha \geq 2$	$\mathcal{O}(n^\tau \log n)$, where $\tau \approx 2.376$

Table 1.1: Results obtained in this thesis for the generalization of the p-CENTER problem on a graph with n nodes. The non-approximability results stated assume that $\mathsf{P} \neq \mathsf{NP}$.

In Chapter 5 we investigate the complexity of a number of problems, known as *compact location problems*. These problems bear their name from the goal to select a number p of facilities in a network minimizing a measure of the distances between them, so as to speak to select a "compact" placement of facilities. The measures that we consider are the diameter of a placement and the total sum of the edge weights between the facilities. The problems addressed in Chapter 5 are bicriteria problems. Again, for the case that the weights on the graph do not obey the triangle inequality, the compact location problems are shown to be extremely hard to approximate. Our approximation results for the case that the triangle inequality is satisfied are shown in Table 1.2. We postpone the precise description of the results until Chapter 5. We also note that Table 1.2

provides only an overview and does not contain the most general versions of the problems as discussed in Chapter 5.

Problem	Hardness Result	Performance Guarantee	Running Time
Δ-(DIA, DIA, p NODES)	Not approximable within $(\alpha, 2-\varepsilon)$ or $(2-\varepsilon, \alpha)$ for any $\alpha \geq 1$ and fixed $\varepsilon > 0$	$(2, 2)$	$\mathcal{O}(n^2)$
Δ-(SUM, DIA, p NODES)	Not approximable within $(\alpha, 2-\varepsilon)$ for any $\alpha \geq 1$ and fixed $\varepsilon > 0$	$(2-2/p, 2)$	$\mathcal{O}(n^2)$
Δ-(DIA, SUM, p NODES)	Not approximable within $(2-\varepsilon, \alpha)$ for any $\alpha \geq 1$ and fixed $\varepsilon > 0$	$(2, 2-2/p)$	$\mathcal{O}(n^2 \log n)$
Δ-(SUM, SUM, p NODES)	NP-hard	$((1+1/\varepsilon)(2-2/p), (1+\varepsilon)(2-2/p))$ for any fixed $\varepsilon > 0$	$\mathcal{O}(n^2 \log^2 n)$

Table 1.2: Results for bicriteria compact location problems in a complete graph with both the edge weights obeying the triangle inequality. The non-approximability results stated assume that $\mathsf{P} \neq \mathsf{NP}$. As discussed in Chapter 5, these results can be extended to handle node weights.

Several problems arising in areas such as communication networks and VLSI design can be expressed in the following general form: Enhance the performance of an underlying network by carrying out upgrades at certain nodes or edges of the network. Chapter 6 introduces such a problem which can be viewed as a bridge between location and network design problems. In this problem, we are given a network where nodes represent processors and edges represent bidirectional communication links. The processor at a node v can be upgraded at an expense of $c(v)$. Such an upgrade reduces the delay of each link emanating from v. The goal is to find a minimum cost set of nodes to be upgraded so that the resulting network has a spanning tree of bottleneck weight at most a given value. Due to its nature, we refer to this problem as a *node weighted network upgrade problem*. We derive approximation and hardness results for the problem. We also show that our approximation algorithms can be used to obtain good algorithms for the symmetric problem, where we are given a bound on the upgrading cost and the goal is to minimize the bottleneck weight. This “conversion” involves only a minor additional computational effort which is logarithmic in the number of nodes in the graph. Chapter 6 also contains improved (in terms of performance and running time) algorithms for the case when there are unit costs on the vertices. The basic results for this problem are summarized in Table 1.3.

Problem	Hardness Result	Performance Guarantee	Running Time
(NODE UPGRADING COST, BOTTLENECK, SPANNING TREE)	Not approximable within $(\alpha, f(n))$ for any $\alpha < \ln n$ and polynomial time computable function f	$(2\ln n, 1)$	$\mathcal{O}(nm \cdot \alpha(m,n))$, where $\alpha(m,n)$ is the inverse of Ackerman's function
(UPGRADING SIZE, BOTTLENECK, SPANNING TREE)	same as above	$(5 + 4\ln\Delta, 1)$, where Δ is the maximum degree in the graph	$\mathcal{O}(n+m)$
(BOTTLENECK, NODE UPGRADING COST, SPANNING TREE)	Not approximable within $(f(n), \alpha)$ for any $\alpha < \ln n$ and polynomial time computable function f	$(1, 2\ln n)$	$\mathcal{O}(nm \cdot \alpha(m,n) \cdot \log n)$
(BOTTLENECK, UPGRADING SIZE, SPANNING TREE)	same as above	$(1, 5 + 4\ln\Delta)$	$\mathcal{O}((n+m)\log n)$

Table 1.3: Results for node weighted upgrading problems. The hardness results assume that $\mathsf{NP} \not\subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$. In Chapter 6, we also present polynomial time algorithms for the problems if restricted to the class of treewidth-bounded graphs.

In Chapters 7 and 8 we continue our study of network design problems. The problems treated in these chapters aim at finding optimal strategies for improving a network under some cost measure subject to budget constraints. Therefore, we refer to them as *network edge improvement problems*. In the first problem addressed in Chapter 7, we consider the problem of reducing the edge lengths of a given network so that the modified network has a spanning tree of small total length. It is assumed that each edge of the given network has an associated function that specifies the cost of shortening the edge by a given amount and that there is a budget on the total reduction cost. The goal is to shorten the edges within the budget restrictions so that the total length of a minimum spanning tree in the modified network is as small as possible. The problem is shown to be NP-hard even on very restricted classes of graphs, contrasting to the polynomial time solvability of the classical minimum spanning tree problem. We provide efficient approximation algorithms, where the cost functions are allowed to be taken from a broad class. For the case when the costs are linear, we present improved algorithms in terms of performance and running time. These results are further improved for the class of treewidth-bounded graphs. We show that our methods can be extended to more general network design problems such as Steiner trees and generalized Steiner networks. An overview over the basic results in Chapter 7 is given in Table 1.4.

Steiner trees are also subject of our study in Chapter 8. We extend our results from Chapter 7 to the case of constrained Steiner trees. Hence, the problems this chapter are referred to as *network edge improvement problems with focus on constrained Steiner trees*. We tackle the problem of decreasing the edge weights of a given network so that the modified network has a constrained Steiner tree of small total length. The additional constraints on the Steiner tree considered are a diameter constraint on a weight function that does not change with the network modification and a constraint on the maximum degree in the Steiner tree. Table 1.5 shows our approximation and hardness results for the case of constrained Steiner trees treated in Chapter 8.

Acknowledgments

At this point, I am greatly indebted to all my friends and colleagues who contributed in various ways to this thesis. First of all, I would like to thank my advisor Professor Hartmut Noltemeier for many suggestions and improvements to this work which would not have been possible without his patient guidance. I also want to thank explicitly Madhav Vishnu Marathe and Sekharipuram Subramaniam Ravi for their interest, effort and enthusiasm when we did research together. I have also greatly benefited from the constructive advice of my colleague Knut Verbarg. Guido Braunreuther and Martin Väth did detailed reviews of this thesis. Riko Jacob gave very useful comments on the framework developed in Chapter 3. To all of them go my sincere thanks.

Problem	Hardness Result	Performance Guarantee	Running Time
(TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE)	NP-hard	General cost functions: $(1 + 1/\varepsilon, 1 + \varepsilon)$ for any fixed $\varepsilon > 0$ (the length of the tree is $1 + 1/\varepsilon$ times the optimum plus an additive ε' which can be made arbitrarily small)	$\mathcal{O}(T_{MST} \cdot \log(\frac{nL_{max}}{\varepsilon\varepsilon'}))$, where T_{MST} is the time required for an MST computation
		$(1 + 2/\varepsilon, 1 + \varepsilon)$ for any fixed $\varepsilon > 0$	$\mathcal{O}(\log \frac{nL_{max}}{\varepsilon} + T_{MST} \cdot \log\log \frac{nL_{max}}{\varepsilon})$
		Linear cost functions: $(1 + 1/\varepsilon, 1 + \varepsilon)$ for any fixed $\varepsilon > 0$	$\mathcal{O}(T_{MST} \cdot \log^2 n)$
(TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE)	Not approximable within $(\alpha, 1)$ or $(1, \beta)$ for any fixed $\alpha, \beta \geq 1$	$(2(1+2/\varepsilon), 2(1+\varepsilon))$ for any fixed $\varepsilon > 0$	$\mathcal{O}(n^2 \log n \cdot \log \frac{nL_{max}}{\varepsilon})$

Table 1.4: Results for network edge improvement problems. We note that for the class of treewidth-bounded graphs Chapter 7 also contains approximation algorithms with improved performances. The hardness results assume that $P \neq NP$.

Credits

The results reported in Chapter 5 on bicriteria compact location problems were obtained jointly with H. Noltemeier, S. S. Ravi and M. V. Marathe [KN+96c, KN+95a, KN+95b].

The results on the node weighted network upgrading problems in Chapter 6 are based on joint work with H. Noltemeier, M. V. Marathe, S. S. Ravi, R. Ravi and R. Sundaram [KN+96b].

The results on the network improvement problem in Chapter 7 were obtained jointly with H. Noltemeier, S. S. Ravi, M. V. Marathe and K. U. Drangmeister [KN+96d]. The extensions of these results to constrained Steiner trees pre-

Problem	Hardness Result	Performance Guarantee	Running Time
(TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE)	Not approximable within $(\alpha, \beta, 1)$ or $(1, \alpha, \beta)$ for any $\alpha, \beta \geq 1$	$(\mathcal{O}(\log \lvert K \rvert)$, $\mathcal{O}(\log \lvert K \rvert)$, $\mathcal{O}(\log \lvert K \rvert))$, where K is the set of terminals.	polynomial time
(TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE)	Not approximable within $(1, \beta, \gamma)$ or $(\alpha, 1, \gamma)$ for any $\alpha, \beta, \gamma \geq 1$	$(\mathcal{O}(\log \frac{\lvert K \rvert}{b})$, $\mathcal{O}(\log \frac{\lvert K \rvert}{b})$, $\mathcal{O}(\log \frac{\lvert K \rvert}{b}))$, where K is the set of terminals and b is the degree-bound	polynomial time

Table 1.5: Results for network edge improvement problems with objective to minimize the weight of a constrained Steiner Tree. The hardness results assume that $\mathsf{P} \neq \mathsf{NP}$.

sented in Chapter 8 were done jointly with H. Noltemeier, M. V. Marathe, R. Ravi and S. S. Ravi [KN+96a].

All the joint work mentioned above represents many hours of stimulating discussions with my colleagues. I am extremely grateful to all of them for their valuable time and for allowing me to include our joint work in this thesis.

2

Preliminaries

In general we assume that the reader is familiar with the most well-known notations and definitions from complexity theory (such as NP-completeness) and graph theory. However, to avoid ambiguities we have compiled a brief overview over the basic terminology used in this thesis. In the following we repeat important notions and cite corresponding literature, where full details can be found.

2.1 Basic Notation

Sets

By $\mathbb{R}$ ($\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$) we denote the set of real (rational, integral, natural) numbers. The set $\mathbb{N}$ of natural numbers does not contain zero. $\mathbb{R}_+$ ($\mathbb{Q}_+$, $\mathbb{Z}_+$) denotes the nonnegative real (rational, integral) numbers.

Let A and B be two sets. We write $A \subseteq B$ ($A \subset B$), if A is a (proper) subset of B. The set of subsets of A will be denoted by 2^A and, finally, $\varnothing$ is used to depict the empty set.

Growth of functions

Let g be a function from $\mathbb{N}$ to $\mathbb{N}$. The set $\mathcal{O}(g(n))$ contains all those functions $f : \mathbb{N} \to \mathbb{N}$ with the property that for constants $c > 0$ and $n_0 \in \mathbb{N}$: $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. A function f belongs to the set $\Omega(g(n))$, if and only if $g(n) \in \mathcal{O}(f(n))$. The notation $f(n) \in \Theta(g(n))$ means that $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$. Finally, we write $f(n) \in o(g(n))$, if $\lim\limits_{n\to\infty} f(n)/g(n) = 0$.

Particular Functions

We use $\log_a$ to denote the *logarithm* function to the basis of a. We omit the basis in the case of $a = 2$ for the sake of convenience. By $\ln n$ we denote the *natural* *logarithm* of a positive number n.

Let $\mathcal{A}$ be *Ackerman's function* which for integers $i, j \geq 1$ is defined by

$$\begin{aligned}
\mathcal{A}(1,j) &= 2^j && \text{for } j \geq 1, \\
\mathcal{A}(i,1) &= \mathcal{A}(i-1,2) && \text{for } i \geq 2, \\
\mathcal{A}(i,j) &= \mathcal{A}(i-1,\mathcal{A}(i,j-1)) && \text{for } i,j \geq 2,
\end{aligned}$$

The *inverse of Ackerman's function* is then defined by

$$\alpha(m,n) = \min\left\{ i \geq 1 : \mathcal{A}(i, \lfloor m/n \rfloor) > \log_2 n \right\}.$$

It should be noted that α is not an inverse of Ackerman's function in the true mathematical sense. Nevertheless, it captures the spirit of the inverse in its growth, which is as slow as Ackerman's function is fast. Details can be found for instance in [CLR90].

Another very slowly growing function is the function β, defined by

$$\beta(m,n) = \min\{ i : \log^{(i)} n \leq m/n \},$$

where $\log^{(i)}$ denotes the ith composition of the natural logarithm.

Alphabets

An alphabet Σ is a nonempty set of characters. By Σ^* we denote the set of all strings over Σ including the empty word.

2.2 Theory of Computation

Model of Computation

The *Turing machine* [GJ79] is the classical model of computation that was used to define the *computational complexity* of algorithms. However, for practical purposes it is fairly more convenient to use a different model. In the *random access machine* or *RAM model* [Pap94,MR95b] we have a machine which consists of an infinite array of *registers*, each capable of containing an *arbitrarily large integer*, possibly negative. The machine is capable of performing the following types of operations involving registers and main memory: input-output operations, memory-register transfers, indirect addressing, arithmetic operations and branching. The arithmetic operations permitted are addition, subtraction, multiplication and division of numbers. Moreover, the RAM can compare two numbers and evaluate the square root of a positive number.

There are two types of RAM models used in literature. In the *log-cost RAM* the execution time of each instruction takes time proportional to the logarithm of the size of its operands, whereas in the *unit-cost RAM* each instruction can be accomplished in one time step. A log-cost RAM is equivalent to the Turing machine under a polynomial time simulation [Pap94]. In contrast, in general there is no polynomial simulation for a unit-cost RAM, since in this model we can compute large integers too quickly by using multiplication. However, if the encoding lengths of the operands occurring during the run of an algorithm on a unit-cost RAM are bounded by a polynomial in the encoding length of the input, a polynomial time algorithm on the unit-cost RAM will transform into a polynomial time algorithm on a Turing machine [GLS88]. This argumentation remains valid in the case of nondeterministic programs.

For convenience, we will use the general unit-cost RAM to analyze the running time of our algorithms. This does not change the essence of our results, because the algorithms in which we are interested involve only operations on numbers that are not significantly larger than those in the input.

Complexity Classes

Classical complexity theory expresses the running time of an algorithm in terms of the "size" of the input, which is intended to measure the amount of data necessary to describe an instance of a problem. The *worst case time complexity* or simply *time complexity* of an algorithm is the function $T(n)$ which is the maximum over all inputs of size n of the sum of the time taken by each instruction executed (cf. [AHU74, GJ79, GLS88]).

We will assume that every problem Π has an (encoding independent) associated function *length* $: D_\Pi \to \mathbb{N}$, which is polynomially related to the inputs lengths that would result from a "reasonable encoding scheme". Here, $D_\Pi \subseteq \Sigma^*$ is the set of *instances* of the problem Π, expressed as words over the alphabet Σ. For a more formal treatment of the input length and also of the notion of a "reasonable encoding scheme" we refer the reader to [GJ79].

The most interesting *complexity classes* for this thesis are the classes P and NP, defined to contain those decision problems which can be solved in polynomial time by deterministic and nondeterministic Turing machines respectively (cf. [GJ79, GLS88, Pap94]).

For a nondecreasing function $f\colon \mathbb{N} \to \mathbb{N}$ the *deterministic time complexity class* $\mathsf{DTIME}(f(n))$ consists of the decision problems for which there exists a deterministic Turing machine deciding the problem in $\mathcal{O}(f(n))$ time. Its nondeterministic counterpart $\mathsf{NTIME}(f(n))$ is defined analogously.

Suppose we are given two decision problems Π and Π'. A *polynomial time transformation* is an algorithm t which, given an encoded instance I of Π, produces in polynomial time an encoded instance $t(I)$ of Π' such that the following holds: For every instance I of Π, the answer to Π is "yes" if and only if the answer to the transformation $t(I)$ (as an instance of Π') is "yes". A decision problem Π is called NP-*complete* if $\Pi \in \mathsf{NP}$ and every other decision problem in NP can be polynomial time transformed to Π.

To tackle also optimization problems rather than just decision problems it is useful to extend the notion of a transformation between problems. Informally, a *polynomial time Turing reduction* (or just *Turing reduction*) from a problem Π to a problem Π' is an algorithm A, which solves Π by using a hypothetical subroutine A' for solving Π' such that if A' were a polynomial time algorithm for Π', then A would be a polynomial time algorithm for Π. More precisely, a polynomial time Turing reduction form Π to Π' is a polynomial time oracle Turing machine (with oracle Π') solving Π.

An optimization problem Π is called NP-*hard* ("at least as difficult as any problem in NP"), if there is an NP-complete decision problem Π' such that Π' can be Turing reduced to Π. Results from complexity theory [GJ79] show that such an NP-hard optimization problem can not be solved in polynomial time unless $\mathsf{P} = \mathsf{NP}$. Details about NP-completeness, NP-hardness and transformations can be found e.g. in [GJ79, GLS88, Pap94].

We close this section with another important complexity class and a related special type of reduction between problems. The class MaxSNP was introduced by Papadimitriou and Yannakakis in [PY91] consisting of those optimization problems Π which can be written in the form

$$\max_S |\{\, \mathbf{x} : \Phi_\Pi(I, S, \mathbf{x}) \,\}|\,,$$

where Φ_Π is a quantifier free formula, I an instance of Π described by a finite structure and S a solution described as a finite structure. The above definition was inspired by Fagin's syntactic characterization of NP in [Fag74].

Let Π and Π' be two optimization problems. An L-*reduction* ("linear reduction") from Π to Π' consists of two polynomial-time computable functions f, g and two positive constants α, β such that for each instance I of Π the following conditions are satisfied:

(i) Algorithm f produces an instance I' of Π' such that the optima of I and I', $\mathrm{OPT}(I)$ and $\mathrm{OPT}(I')$ respectively, satisfy $\mathrm{OPT}(I') \le \alpha \cdot \mathrm{OPT}(I)$ and,

(ii) given any solution of I' with objective function value ζ', algorithm g produces a solution of I with objective value ζ such that the estimate $|\zeta - \mathrm{OPT}(I)| \le \beta \cdot |\zeta' - \mathrm{OPT}(I')|$ holds.

An optimization problem Π is called MaxSNP-*hard*, if every problem in the class MaxSNP can be L-reduced to Π. The problem is termed MaxSNP-*complete* if is MaxSNP-hard and contained in MaxSNP.

Papadimitriou and Yannakakis [PY91] prove that in fact there exist MaxSNP-complete problems and provide a list of several such problems. It is also shown [PY91] that L-reductions compose. Hence, to prove MaxSNP-hardness of a problem Π it suffices to show that one of the already known MaxSNP-hard problems L-reduces to Π.

2.3 Graph Theory

As a standard notation we use $G = (V, E)$ to denote the *undirected graph* (or simply *graph*) with vertex set V and edge set E. We usually denote by $n := |V|$ and $m := |E|$ the cardinalities of V and E respectively. Throughout the thesis we assume that both V and E are finite. The elements of V are called *vertices* or *nodes*, while we refer to the elements of E as *edges*. Each edge is an unordered pair (u, v) of distinct vertices $u \neq v$. The edge (u, v) is said to be *incident* with the vertices u and v. We call two vertices *adjacent*, if there is an edge which is incident with both of them. The *neighborhood* of a vertex v in the graph G, denoted by $N_G(v)$, is the set of vertices that are adjacent to v. Each vertex in $N_G(v)$ is also called a *neighbor* of v. The number $|N_G(v)|$ of neighbors is also called the *degree* of v and denoted $\deg_G(v)$. For a set $U \subseteq V$, we define its neighborhood $N_G(U)$ in G by $N_G(U) := \bigcup_{v \in U} N_G(v)$.

A *subgraph* of G is a graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. If $W \subseteq V$ is a subset of the vertices of G, then $G[W]$ denotes the subgraph *induced by the vertices in* W, i.e., $G[W]$ is the subgraph of G consisting of the vertices in W and all those edges of E with both endpoints in W. A graph $H = (V, E')$ with the same vertex set as $G = (V, E)$ and $E' \subseteq E$ is termed an *edge subgraph* of G.

A subset C of the vertices such that every pair of vertices is adjacent is called a *clique* of size $|C|$ in the graph G. A graph G whose vertex set forms a clique is said to be a *complete graph*. A clique C such that there is no other clique C' in G strictly containing C is a *maximal clique*. A *maximum clique* is a clique of maximum cardinality in G. The maximum size of a clique in G, denoted by $\omega(G)$, is called the *clique number* of G.

An *independent set* U is a subset of the vertices of G such that no pair u, v of nodes in U is adjacent. The notions *maximal* and *maximum* for independent sets are defined totally similar to the case of cliques.

A set $D \subseteq V$ is called a *dominating set*, if any node in V either belongs to D or has a neighbor in D.

A subset E' of the edges of G such that no two edges of E' share an endpoint is called a *matching* in G. The matching is termed a *maximal matching* if it is maximal with respect to inclusion. A *maximum matching* is a matching of maximum cardinality. Finally, a matching is called a *perfect matching*, if each vertex of G is adjacent to at least (and thus to exactly) one edge in the matching.

A *path* is an alternating sequence of vertices and edges $v_0, e_1, v_1 \ldots, e_r, v_r$, starting and ending with a vertex, in which every edge is incident with the two vertices immediately preceding and following it. This path joins v_0 and v_r and is sometimes also called a v_0-v_r path. For briefness, the path may also be denoted by $(v_0, v_1, \ldots, v_r)$ or $e_1, \ldots, e_r$. It is called a *simple path* if all the vertices are distinct. A path is *closed* if $v_0 = v_r$ and *open* otherwise. A *cycle* is a closed path where all the internal vertices are distinct.

A *tree* is a special type of graph, which is connected (i.e., every two vertices are connected by a path) and does not contain any cycle. A *spanning tree* $T = (V, E_T)$ of a graph $G = (V, E)$ is a tree which has the same vertex set as G. *Steiner trees* are a generalization of spanning trees. Given a subset $K \subseteq V$ of distinguished nodes called *terminals* , any subgraph T of G which is a tree and which contains all the terminals from K is said to be a of G for K. Clearly, in the special case that $K = V$, then T is a spanning tree of G.

A *tree-decomposition* of a graph $G = (V, E)$ is a pair $D = (S, T)$ with $S = \{X_i : i \in I\}$ a collection of subsets of V and $T = (I, F)$ a tree, which has one node for each subset of S, such that the following three conditions are satisfied:

(i) $\bigcup_{i \in I} X_i = V$,

(ii) for all edges $(u, v) \in E$, there exists a subset $X_i \in S$ containing both vertices u and v,

(iii) for each vertex $v \in V$, the set of nodes $\{i : v \in X_i\}$ forms a subtree of T.

The *width of a tree-decomposition* $D = (S, T)$ is defined to be $\max_{i \in I}(|X_i| - 1)$. The *treewidth* of a graph $G = (V, E)$ is the minimum width of a tree-decompositions of G. Figure 2.1 on the facing page displays a graph G and a corresponding tree-decomposition of width 2.

Treewidth-bounded graphs were introduced by Robertson and Seymour [RS90]. Independently, Bern, Lawler and Wong [BLW87] introduced the notion of *decomposable graphs*. Later, it was shown [AC+93] that the class of decomposable graphs and the class of treewidth-bounded graphs coincide. A class of *decomposable graphs* Γ is given by a set of recursive rules that satisfy the following conditions [BLW87]:

1. The rules define a finite number of primitive graphs.

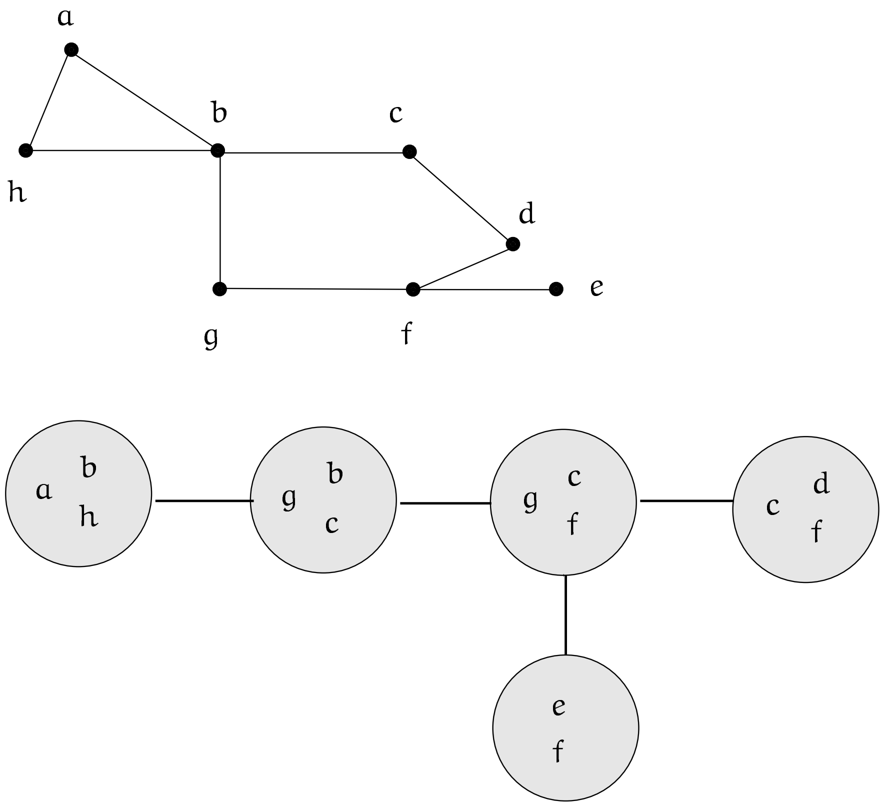


Figure 2.1: Example of a graph with tree-decomposition.

2. Each graph in Γ has an ordered (possibly empty) set of special nodes called *terminals*.[1] The number of terminals in each graph is bounded by a global constant.

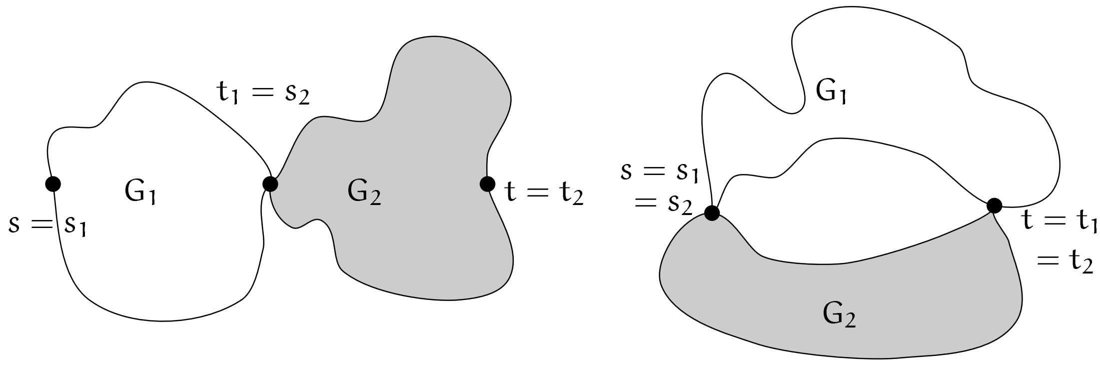
3. There is a finite collection of binary composition rules that operate only at terminals, either by identifying two terminals or adding an edge (called *attachment edge*) between terminals. A composition rule also determines the terminals of the resulting graph, which must be a subset of the terminals of the two graphs being composed.

As an example of a simple class of decomposable graphs we consider the class of *series-parallel graphs*, which can be defined by the following rules [BLW87].

1. The set of primitive graphs consists of the single graph P with vertex set $\{s, t\}$ and the single edge (s, t). The vertex s is the "start-terminal" of P and the vertex t is the "end-terminal" of P.

[1]The notion of a terminal in this context should not be confused with the notion of a terminal for Steiner trees.

2. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be series-parallel graphs with terminals s_1, t_1 and s_2, t_2 respectively. Then
   (a) The graph obtained by identifying t_1 and s_2 is a series-parallel graph, with s_1 and t_2 as its terminals. This graph is the *series composition* of G_1 and G_2 (see Figure 2.2(a)).
   (b) The graph obtained by identifying s_1 and s_2 and also t_1 and t_2 is a series-parallel graph, the *parallel composition* of G_1 and G_2. This graph has $s_1(= s_2)$ and $t_1(= t_2)$ as its terminals (see Figure 2.2(b)).



(a) Series composition (b) Parallel compositon

Figure 2.2: Composition rules for series-parallel graphs.

Let Γ be any class of decomposable graphs. Following [BLW87], we assume that a given graph $G \in \Gamma$ is accompanied by a parse tree specifying how G is constructed using the rules and that the size of the parse tree is linear in the number of nodes. Moreover, we may assume without loss of generality that the parse tree is a binary tree.

For an introduction to graph theory we refer the reader to [Har72, Nol75]. An introduction to treewidth can be found e.g. in [Bod92, Klo93, Klo94]. Additional definitions to the basic ones presented above will be given in the following chapters in the respective contexts.

There are two standard ways to represent a graph $G = (V, E)$ (see e.g. [CLR90]): the *adjacency-list representation* and the *adjacency-matrix representation*. The former consists of an array *Adj* of $|V|$ lists, one for each vertex in V. The list *Adj*(v) contains all the vertices $u \in V$ such that $(v, u) \in E$.

In the adjacency-matrix representation, the graph G is stored by its adjacency-matrix, $A(G)$. This 0/1-matrix of size $|V| \times |V|$ contains a 1 at position (i, j) if and only if $(v_i, v_j) \in E$.

It should be noted that both representations can be converted into each other in time $\mathcal{O}(|V|^2)$. Unless stated differently, we will work with the adjacency-list representation. A comprehensive list of ways how to represent graphs can be found in [CLR90].

3

Approximation Algorithms

In this chapter we will introduce *approximation algorithms* for NP-hard optimization problems. The *performance* of an algorithm is one of the central notions in this context. It measures the quality of a solution produced by the algorithm compared to an optimal solution of the problem. Under certain circumstances it is meaningful to allow the approximation algorithm also to output "slightly infeasible" solutions. Most of the approximation algorithms presented in this thesis fall into this category.

In Sections 3.1 and 3.2 we provide a formal framework for defining optimization problems and stating the performance of an approximation algorithm in terms of optimality and infeasibility of the solutions found by the heuristic. First, we show in Section 3.1 how to define the optimization problems appropriately. Section 3.2 then introduces the most important notion within this thesis: The *performance* of an approximation algorithm measures the error of the solution produced in terms of optimality and violation of constraints.

Section 3.3 provides a justification for formulating optimization problems as multicriteria problems. We establish a connection between multi- and unicriterion approximation: good multicriteria approximation algorithms transform into good approximation algorithms for certain related unicriterion problems. The converse is not true in general.

For some problems, providing a certain performance is as hard as finding an optimal solution. We address such *hardness* or *non-approximability results* in Section 3.4. Section 3.5 contains some fundamental techniques which are used throughout this thesis to design approximation algorithms.

We close the chapter with a brief discussion on alternatives to approximation algorithms, when dealing with hard problems.

3.1 Optimization Problems

While the class NP consists only of decision problems, many problems encountered in practice are optimization problems. A first definition of a class of such

NP-optimization problems was made by Johnson [Joh74]. Later, Crescenzi and Panconesi [CP91] introduced the class NPO, which is now commonly used if mainly dealing with generic NP-optimization problems. However, when considering a specific problem, formulating this problem in the framework from [CP91] is not very convenient and usually more intuitive formulations are given.

Below we will introduce the notion of a k-*criteria optimization problem* as a basis for the problems and approximation algorithms that are studied in this thesis. We sketch the idea behind this notion before giving a formal definition.

Suppose we want to upgrade a given network by replacing old links between the sites by faster links. Basically, our goal would be to maximize the performance of the network. It is clear that the more new links we install the better will be the performance of the network. However, building new links involves spending money. Thus, we could rephrase our goal as follows: "Spend little of our money while achieving a good performance of the network". A common way (see [Rav93, RM+93]) to formulate the problem is to specify a bound on the expenses and then to maximize the performance of the network.

We formalize the framework outlined above in the following definition.

Definition 3.1 (k-Criteria Optimization Problem)
*A **k-criteria minimization problem** Π on a weighted graph is defined by specifying k polynomial time computable minimization objectives, $f_1, \dots, f_k$, and a membership requirement in a class of weighted subgraphs Γ (not necessarily weighted exactly the same way as the original graph).*

The problem specifies budget values $F_2, \dots, F_k$ as upper bounds on the objectives $f_2, \dots, f_k$. The goal is to find a subgraph from

$$\{ x \in \Gamma : f_i(x) \leq F_i, i = 2, \dots, k \}, \tag{3.1}$$

having minimum possible value for f_1.

*Similarly, a **k-criteria maximization problem** consists of selecting a subgraph from* (3.1) *and maximizing f_1.*

Given an instance of a k-criteria problem, the *set of feasible solutions* for the instance is defined as the set given in (3.1).

In the case $k = 1$, the above definition falls back to finding a subgraph from the class Γ which minimizes (or maximizes) the objective f_1. We usually omit the index of one in this case.

Definition 3.1 might read complex. For better understanding, we illustrate the notion of a k-criteria problem with the help of some examples.

Example 3.2 Consider the *minimum spanning tree problem*. This is a classical network design problem and known to be solvable in polynomial time, e.g. by Kruskal's algorithm (see e.g. [Nol75, CLR90]). The input for the problem consists of a connected graph $G = (V, E)$ with edge-weights $d(e)$ $(e \in E)$. The goal is to find a spanning tree of G of minimum total weight.

The problem can be formulated with the help of Definition 3.1 as follows: We choose Γ to represent the class of spanning trees and select the only objective to be the sum of the weights of the edges that are in the tree. ◇

The following example shows an extension of the minimum spanning tree problem to a *bicriteria problem*.

Example 3.3 We are given a graph G and *two* weight functions c and d on its edges. Here, the weight function c models e.g. the construction cost and d the cost of using a particular edge (once it has been constructed). The idea behind the *constrained minimum spanning tree problem* to find a spanning tree of G that has low cost with respect to *both* edge weight functions. Formulated in the just given framework, this problem is stated as a *bicriteria problem* as follows: Given a bound B on the total d-weight of the tree, the goal is to find a tree of minimum c-cost subject to the budget constraint on its d-weight. ◇

To obtain good mnemonics for the problems in this thesis, we will adopt the following notational conventions: If a problem has not already been assigned a name in literature, we write a vector consisting of brief abbreviations for each objective f_i in the problem and a shortcut for the subgraph class Γ. Thus, the problem from Example 3.3 is denoted by (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE).

It should be noted that in contrast to the minimum spanning tree problem, the constrained version (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE) is NP-hard [AAN82].

Example 3.4 Another classical NP-hard problem from graph theory is the *minimum vertex cover problem* (see e.g. [GJ79, Problem GT1]), usually denoted MIN VERTEX COVER for short. Given an undirected graph G, the problem is to find a vertex cover in G of minimum cardinality, i.e., a vertex subset S of minimum cardinality such that each edge from G has at least one endpoint in S.

The problem is formulated as a unicriterion problem: We let Γ denote those subgraphs whose vertex set covers all edges in G in the sense defined above. The objective is the cardinality of the set of vertices in the subgraph, which is to be minimized. ◇

It should become clear from the previous examples how optimization problems on graphs can be cast into the framework given by Definition 3.1.

3.2 Performance Guarantees

In this section we will introduce the notion of the *performance* of an approximation algorithm for a multicriteria problem. The performance of a k-criteria approximation algorithm is given in a natural way as a vector $(\alpha_1, \ldots, \alpha_k)$, where α_1 measures the maximum (multiplicative) deviation from optimality and the α_i, $i \geq 2$ measure the maximum violation of the constraints posed in the problem.

Definition 3.5 (k-Criteria Approximation Algorithm)
*A **k-criteria approximation algorithm** for a k-criteria minimization problem Π with performance $(\alpha_1, \ldots, \alpha_k)$ is a polynomial time algorithm with the following properties:*

Given any instance of Π with nonempty set of feasible solutions, the algorithm finds an "almost feasible solution", that is a solution from

$$\{ x \in \Gamma : f_i(x) \leq \alpha_i \cdot F_i, i = 2, \ldots, k \} \tag{3.2}$$

such that $f_1(x) \leq \alpha_1 \cdot \mathrm{OPT}$. *Here* OPT *denotes the minimum value of a solution satisfying all the constraints.*

If there are no feasible solutions for an instance, the algorithm has the choice of producing an "almost feasible solution" from the set (3.2) *or providing the (correct) information that the set of feasible solutions is empty.*

In the case of a maximization problem $f_1(x) \leq \alpha_1 \cdot \mathrm{OPT}$ in the above definition is naturally replaced by the requirement that $f_1(x) \geq \mathrm{OPT}/\alpha_1$.

In general, each *performance factor* α_i depends on the input size and on the budget values $F_2, \ldots, F_k$. However, in most cases, the α_i will be constants. The performance of an approximation algorithm is measured in terms of both optimality and the constraint violation of the solution produced. Notice that any solution produced by the algorithm is contained in the set (3.2) and thus in particular belongs to the subgraph-class Γ.

We also use the term $(\alpha_1, \ldots, \alpha_k)$*-approximation algorithm* to refer to a k-criteria approximation algorithm with performance $(\alpha_1, \ldots, \alpha_k)$.

The minimum spanning tree problem (see Example 3.2) can be solved by Kruskal's algorithm in time $\mathcal{O}(m \log m)$ on a graph with m edges [Nol75, CLR90]. Thus, Kruskal's algorithm is a unicriterion approximation algorithm with a performance of 1 for the problem, although one would usually not speak of an approximation algorithm in this case. A non-trivial example of a unicriterion approximation algorithm is given in the next example.

Example 3.6 Consider the MIN VERTEX COVER problem presented in Example 3.4. It is easy to see that the endpoints of a maximal matching form a vertex cover of at most twice the size of the minimum cardinality vertex cover. We can find a maximal matching in a given graph in polynomial time, by simple choosing an edge and then repeatedly adding edges to our set until any further addition would destroy the matching property. Thus, by the observations from above the algorithm which first computes such a maximal matching and then returns the endpoints of the edges in the matching, is a unicriterion approximation algorithm for MIN VERTEX COVER with a performance of 2, or simply a 2-approximation algorithm. $\diamond$

It is more difficult to present brief results on k-criteria approximation algorithms for $k \geq 2$. Thus, we only give two very simple examples of two-criteria approximation algorithms for illustration.

Example 3.7 We give a $(1, n-1)$-approximation algorithm for the (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE) problem (cf. Example 3.3), where $n := |V|$ denotes the number of vertices in the input graph. The algorithm works as follows: It first removes all edges of d-weight strictly greater than B from the graph. Recall that B is the bound on the total d-weight of a tree. If the edge removal disconnects the graph, the algorithm outputs the information that there is no feasible solution. Otherwise, it computes a minimum spanning tree with respect to the c-weights in the resulting graph. It is easy to see that the edges removed by the algorithm can not belong to any feasible solution and that the total c-weight of the tree found is at most that of the optimal constrained tree (provided there exists a feasible tree). Moreover, since the d-weight of each edge in the tree found by the algorithm is at most B, the bound on the total d-weight will be violated by a factor of at most $n-1$. $\diamond$

Example 3.8 In this example we give a $(2, 2)$-approximation algorithm for the MAX KNAPSACK problem, which is the optimization version of the classical KNAPSACK problem (see [GJ79, Problem MP9]).

An instance of KNAPSACK is given by a finite set U, for each $u \in U$ a size $s(u) \in \mathbb{N}$ and a value $v(u) \in \mathbb{N}$ and positive integers K and B. The question is whether there is a subset $U' \subseteq U$ of size $s(U') := \sum_{u \in U'} s(u)$ at most B such that $v(U') := \sum_{u \in U'} v(u) \geq K$. The problem MAX KNAPSACK is to find a set $U' \subset U$ of size at most B which has maximum value $v(U')$.

Although MAX KNAPSACK is not formulated as a problem on graphs we can easily rephrase it to fit into our framework given by Definition 3.1. We omit this reformulation here.

Consider the simple heuristic shown in Algorithm 3.1. This algorithm first sorts the elements of U into decreasing order with respect to their "profit density" $v(u)/s(u)$. Then, starting with the first element, it keeps on adding items in their sorted order if they still fit into its knapsack of size $2B$. Thus, by construction of the algorithm, the budget constraint on the size of the elements in the knapsack will be violated by a factor of at most 2.

Let U' be the set generated by Algorithm 3.1 and let U^* be an optimal set of size $s(U^*) \leq B$ and maximum value OPT $:= \nu(U^*)$. We assume that the elements in $U^* = \{u_1^*, \ldots, u_r^*\}$ are numbered in decreasing order with respect to their profit density. We also assume without loss of generality that no item in U has size greater than B. What we have to show is that the profit $\nu(U')$ is at least $\nu(U^*)/2$.

Let $k \in \{1, \ldots, r\}$ be the minimum value such that u_k^* is not added to the set U' by Algorithm 3.1. If there is no such index k, then there is nothing left to prove. At the point when u_k^* was considered, it was not added to the set U' because at that point $s(U') \geq 2B - s(u_k^*)$. Since $s(u_k^*) \leq B$, it follows that $s(U') \geq B$. Since all the items in U' have profit density of at least $\nu(u_k^*)/s(u_k^*)$ by the fact that they have been considered *before* u_k^*, it follows that

$$\nu(U') \geq \frac{\nu(u_k^*)}{s(u_k^*)} \cdot s(U') \geq \frac{\nu(u_k^*)}{s(u_k^*)} \cdot B. \tag{3.3}$$

On the other hand, we have that $\nu(u_k^*)/s(u_k^*) \geq \nu(u_j^*)/s(u_j^*)$, or equivalently $\nu(u_j^*) \leq s(u_j^*) \cdot \nu(u_k^*)/s(u_k^*)$, for $j = k, \ldots, r$. Thus

$$\begin{aligned}
\text{OPT} = \nu(U^*) &= \nu(\underbrace{\{u_1^*, \ldots, u_{k-1}^*\}}_{\subset U'}) + \sum_{j=k}^{r} \nu(u_j^*) \\
&\leq \nu(U') + \frac{\nu(u_k^*)}{s(u_k^*)} \cdot \sum_{j=k}^{r} s(u_j^*) \\
&\leq \nu(U') + \frac{\nu(u_k^*)}{s(u_k^*)} \cdot B \\
&\stackrel{(3.3)}{\leq} 2 \cdot \nu(U').
\end{aligned}$$

This means that at the point when u_k^* was rejected by Algorithm 3.1 the profit of the current set U' was already at least OPT/2. ◇

We would like to note that the simple proof of the performance $(2, 2)$ given in the last example relies on the fact that the algorithm can pack items of size at most $2B$ into its knapsack. It is straightforward to see that the modification of the algorithm where we allow only a size of B in Step 4 does *not* have a performance of $(\alpha, 1)$ for any $\alpha \geq 1$. Nevertheless, there exists another easy modification of Algorithm 3.1 that improves the performance to $(2, 1)$ (cf. [Mot92]).

Algorithm 3.1 $(2, 2)$-Approximation algorithm for MAX KNAPSACK.

Input: A finite set U, for each $u \in U$ a size $s(u)$ and a value $v(u)$, a number B

Sort the elements in U according to decreasing "profit density" $v(u)/s(u)$.
$\{$assume now that $v(u_1)/s(u_1) \geq \cdots \geq v(u_n)/s(u_n)\}$
$U' \leftarrow \varnothing$
for $i \leftarrow 1, \dots, n$ **do**
    if $s(U') + s(u_i) \leq 2B$ **then**
        $U' \leftarrow U \cup \{u_i\}$
    end if
end for
return U'

A Note on the Infeasibility of Multicriteria Problems

A seeming drawback of our Definition 3.5 is that in case of the infeasibility of a problem instance, a k-criteria approximation algorithm might give us an "almost feasible solution" with a very bad objective function value (since in this case the algorithm is allowed to pick *just any* "almost feasible solution").

Of course, if we can check feasibility of an instance in polynomial time, then there are no severe problems. We can accomplish this check *before* running the k-criteria approximation algorithm. If we detect infeasibility, we simply inform about this fact and stop.

For the rest of this subsection assume that, given an instance of the k-criteria minimization problem Π, determining whether the set of feasible solutions is empty is NP-complete. In this case, the existence of any polynomial time algorithm that outputs a feasible solution if and only if a problem instance is feasible, will imply that $\mathsf{P} = \mathsf{NP}$. Thus, it seems unavoidable to allow an approximation algorithm to possess a certain kind of additional "freedom".

Let A be an $(\alpha_1, \dots, \alpha_k)$-approximation algorithm for the problem Π, where for simplicity the α_i are assumed to be constants greater or equal to 1. Suppose that given an instance I of Π the algorithm A returns the solution $\tilde{x}$ (and thus in particular does not inform about infeasibility). We denote by

$$\mathcal{S}_I = \{\, x \in \Gamma : f_i(x) \leq F_i, i = 2, \dots, k \,\}$$

the set of feasible solutions for I. We now run A again on the instance I' obtained from I by setting the bounds on the objectives f_i to $F_i' := \alpha_i F_i$ for $i = 2, \dots, k$. Thus, for I' the problem consists of finding a subgraph from the set

$$\begin{aligned}\mathcal{S}_{I'} &= \{\, x \in \Gamma : f_i(x) \le F_i',\, i = 2, \ldots, k \,\} \\ &= \{\, x \in \Gamma : f_i(x) \le \alpha_i F_i,\, i = 2, \ldots, k \,\}\end{aligned} \tag{3.4}$$

minimizing f_1. Notice that the set of feasible solutions defined in (3.4) is nonempty, since $\tilde{x} \in \mathcal{S}_{I'}$. Let the corresponding solution delivered by A be x'. We then output $\tilde{x}$ or x', depending on which of the two solutions has a better objective function value under f_1.

The solution $\hat{x}$ generated this way will violate the constraints $f_i(x) \le F_i$ at most by the factors α_i^2, $i = 2, \ldots, k$. Moreover, $\hat{x}$ is a good approximate solution for the "relaxed" instance I' corresponding to I and satisfies

$$f_1(\hat{x}) \le \alpha_1 \cdot \min\{\, f_1(x) : x \in \mathcal{S}_I \cup \mathcal{S}_{I'} \,\}.$$

Also, if the set of feasible solutions $\mathcal{S}_I$ of the original instance I is nonempty, then

$$f_1(\hat{x}) \le \alpha_1 \cdot \min\{\, f_1(x) : x \in \mathcal{S}_I \,\}.$$

The above arguments show that we can augment a k-criteria approximation algorithm to give us acceptably good solutions even if our original problem instance is infeasible. Instead of just outputting *any* "almost feasible solution" in any case we determine one that has a *good* objective function value f_1 for a relaxed problem instance. In view of the assumption that deciding feasibility is already NP-complete, this seems to be the best we can accomplish in polynomial time (assuming that $\mathsf{P} \neq \mathsf{NP}$).

Other Performance Measures for Approximation

The performance of an approximation algorithm as introduced in this section measures *multiplicative errors*. For unicriterion problems this measure is also called the *performance ratio* or the *approximation factor* of an algorithm.

It should be noted that the performance ratio is not the only way to evaluate the quality of an approximation algorithm, although it is the one which is most widely used in literature. In the sequel we briefly indicate alternative measures for the case of unicriterion problems. A comprehensive survey can be found e.g. in [Kan92].

A different method of measuring the quality of a solution is the *absolute error*. The absolute error of a feasible solution x with respect to the optimum value $\mathrm{OPT}(I)$ of the instance I of an optimization problem with objective function

f is defined to be $|f(x) - \mathrm{OPT}(I)|$. Only a few problems have an absolute approximation algorithm. In general, these are problems, where the value of the optimal solution can easily be pinned down within a small range.

As an example consider the problem of coloring the vertices of a planar graph such that no two adjacent vertices have the same color. It is well known that every planar graph can be colored with at most five[1] colors [Har72], while deciding whether three colors suffice is NP-complete [GJ79]. Consider now the problem of coloring a planar graph with the minimum number of colors possible. A polynomial time approximation algorithm for this problem with an absolute error of 4 would be to output the value 5 for each graph in the input.

Another measure is the *normalized relative error* $E^n(x, I)$, defined by

$$E^n(x, I) := \frac{|f(x) - \mathrm{OPT}(I)|}{|\mathrm{WORST}(I) - \mathrm{OPT}(I)|}.$$

Here, $\mathrm{WORST}(I)$ is the maximum objective function value of a feasible solution for the instance I in the case of a minimization problem. In the case of a maximization problem $\mathrm{WORST}(I)$ denotes the minimum objective function value over the set of feasible solutions, which is usually zero. Note that the normalized relative error satisfies $0 \leq E^n(x, I) \leq 1$. This measure is hardly ever used, perhaps because it is difficult to handle. The idea behind the notion of the normalized relative error can be explained best by an example.

Consider the problem MAX INDEPENDENT SET of finding a maximum size independent set in a graph $G = (V, E)$. Recall that a subset U of the vertices is termed *independent* if there is no edge between any pair of vertices from U. It is easy to see that U is an independent set if and only if $V - U$ is a vertex cover. An approximate solution for MAX INDEPENDENT SET on G with absolute error ϵ will also give us an approximation with the same absolute error for the instance of MIN VERTEX COVER on the same graph. Hence, we may deem it equally hard to obtain absolute approximations for MAX INDEPENDENT SET and MIN VERTEX COVER .

The case is different for the performance ratio. Recall from Example 3.6 that we can provide a polynomial time 2-approximation for MIN VERTEX COVER. In contrast, it can be shown [BS94] that, unless $\mathsf{P} = \mathsf{NP}$, for any fixed $\varepsilon > 0$ there can be no polynomial time approximation algorithm for MAX INDEPENDENT SET with a performance guarantee of $|V|^{1/6-\varepsilon}$, where V is the vertex set of the graph given in the instance. However, given an approximate solution U for the independent set problem on G with *normalized relative error* α, the set $V - U$

[1] The four-color Theorem [AH77, AHK77] states the stronger result that every planar graph can be colored by using at most four colors.

approximates the minimum size vertex cover in G with normalized relative error α.

Multiobjective Optimization in Operations Research

In operations research literature we often find a different notion of multiobjective or multicriteria optimization (cf. [War92, Ham95]). In the sequel we will briefly compare this definition to our model of multicriteria optimization.

Suppose that $f : \mathcal{M} \to \mathbb{R}^k$ is a vector-valued function with domain $\mathcal{M}$. An element $x \in \mathcal{M}$ *dominates* $x' \in \mathcal{M}$ if

$$\begin{aligned} f_i(x) &\leq f_i(x') && \text{for } i = 1, \ldots, k \quad \text{and} \\ f_{i_0}(x) &< f_{i_0}(x') && \text{for at least one } i_0 \in \{1, \ldots, k\}. \end{aligned}$$

Here, $f_i(x)$ denotes the ith coordinate of the k-vector $f(x)$. The solution $x \in \mathcal{M}$ is called *efficient* if there is no $x' \in \mathcal{M}$ which dominates x. The problem of finding all[2] efficient solutions in $\mathcal{M}$ is then called a *multiobjective optimization problem* [War92, Ham95].

Thus, by its pure nature a multiobjective problems defined this way is different from a k-criteria problem as in Definition 3.1. While we are interested in finding one optimal (or nearly optimal) solution, the definition given above deals with finding a whole set of "good" solutions. Also, our definition allows some sloppiness for an optimal solution: as long as the bounds on the objectives $f_2, \ldots, f_k$ are fulfilled, we do not care about how much the solution stays below these bounds.

Let Π be a k-criteria problem as in our Definition 3.1. The goal is to find a subgraph from

$$\{ x \in \Gamma : f_i(x) \leq F_i, i = 2, \ldots, k \},$$

having minimum possible value for f_1. Let I be an instance of Π. We consider the instance I_P of the multiobjective problem P defined by the set $\mathcal{M} := \Gamma$ and the vector valued function $f = (f_1, \ldots, f_k)$. The multiobjective problem P consists of finding the efficient subgraphs in Γ. Any efficient solution to I_P which happens to obey the bounds $F_2, \ldots, F_k$ on the respective objectives $f_2, \ldots, f_k$ is clearly an optimal solution for the instance I of Π. However, an efficient solution for I_P need not be feasible for I. Conversely, an optimal solution for I is not necessarily an efficient solution for I_P. It should also be noted that the set of efficient solutions for I_P might be extremely large, i.e., exponential in the size of I_P (cf. [War92]).

[2] Sometimes one restricts the problem to finding all "lexicographically" efficient solutions (see [Ham95]).

3.3 Multicriteria and Unicriterion Approximation

One might ask how meaningful it is to measure the performance of an approximation algorithm as a vector and to allow constraints to be violated within a certain range. Also, the question arises whether there is a connection between multicriteria approximability and unicriterion approximability. In this section we will answer both questions in the affirmative. In case of the former we give an example of a classical unicriterion problem, which can not be approximated with a constant performance (unless $\mathsf{P} = \mathsf{NP}$), but which has polynomial time approximation algorithms with performance $1 + \varepsilon$ for any $\varepsilon > 0$ if we allow a constraint of the problem to be violated slightly. As far as the latter is concerned we show that, while k-criteria approximation algorithms can be used to obtain good approximation algorithms for unicriterion problems, the converse is not true in general. This emphasizes that it is indeed sensible to consider multicriteria approximation algorithms.

The p-*median problem* is a classical problem of location theory. We are given a complete graph $G = (V, E)$ on n vertices and a nonnegative edge-weight function d. In the network location context, for example, vertices may represent cities and edge weights may correspond to distances between pairs of cities. The problem consists of selecting a given number p of vertices, the *medians*, such that the sum of distances from each vertex to its nearest median is minimized.[3] The p-median problem is NP-hard [GJ79, Problem ND91]. Moreover, it has been shown in [LV92] that there is no polynomial time approximation for the problem with a constant performance ratio unless $\mathsf{P} = \mathsf{NP}$. In contrast, if a small violation of the cardinality of the median set is allowed, then the problem can be approximated within a factor of $1 + \varepsilon$ for any fixed $\varepsilon > 0$ [LV92].

We now discuss how approximation algorithms for multicriteria problems can be used to obtain approximation algorithms for certain related unicriterion problems. Let Π be a k-criteria minimization problem. For a fixed weight-vector $\boldsymbol{\sigma} = (\sigma_1, \dots, \sigma_k) \in \mathbb{N}^k$, we denote by Π_+^{σ} and $\Pi_\times^{\sigma}$ the unicriterion problems of finding a subgraph from Γ minimizing $f_+^{\sigma} := \sum_{i=1}^{k} \sigma_i f_i$ and $f_\times^{\sigma} := \prod_{i=1}^{k} f_i^{\sigma_i}$ respectively. Throughout this section we will assume that the objectives $f_1, \dots, f_k$ are integer valued. In case of rational objectives, the results can be extended easily with a small loss in the performance guarantees. The restriction to integral cost functions will help us to focus on the essential connections between multicriteria and unicriterion approximation. We will also make the following assumption about the *boundedness* of the objective function values of Π_+^{σ} and $\Pi_\times^{\sigma}$:

[3] The related problem of minimizing the *maximum* distance of a vertex to its closest facility is treated in Chapter 4

Assumption 3.9 *Let* $\Pi' \in \{\Pi_+^\sigma, \Pi_\times^\sigma\}$. *There is a polynomial* p *with the following property: for any instance* I *of* Π' *with encoding length* $|I|$ *there exist (polynomial time computable) numbers* $B_2, \dots, B_k$ *with* $B_i \leq 2^{p(|I|)}$ *for* $i = 2, \dots, k$ *such that the optimal solution* x^* *to the instance of* Π' *satisfies:*

$$1 \leq f_i(x^*) \leq B_i, \qquad \text{for } i = 2, \dots, k.$$

It should be noted here that Assumption 3.9 is in fact no severe restriction. If we assume a reasonable encoding scheme for numbers such as the usual binary encoding, then the assumption merely states that the encoding lengths of the single objective function values $f_i(x^*)$ of an optimal solution x^* remain polynomially bounded in the input length. If this were not the case, then the encoding lengths of $f_+^\sigma(x^*)$ and $f_\times^\sigma(x^*)$ would not be polynomially bounded (assuming again a reasonable encoding scheme). Hence, there could be no polynomial time algorithm, neither deterministic nor nondeterministic, that finds an optimal solution *and* outputs the optimal function value, since the encoding length of the objective function value is exponential in the input size.

Theorem 3.10 *Assume that there is an* $(\alpha_1, \dots, \alpha_k)$*-approximation algorithm for a* k*-criteria problem* Π. *Then for any* $\varepsilon > 0$ *and* $\boldsymbol{\sigma} = (\sigma_1, \dots, \sigma_k) \in \mathbb{N}^k$ *the following statements hold:*

(i) There is a polynomial time approximation algorithm for Π_+^σ *with a performance guarantee of* $(1 + \varepsilon)\max\{\alpha_1, \dots, \alpha_k\}$.

(ii) There is a polynomial time approximation algorithm for $\Pi_\times^\sigma$ *with a performance guarantee of* $(1 + \varepsilon)^{\sum_{i=2}^k \sigma_i} \prod_{i=1}^k \alpha_i^{\sigma_i}$.

In particular, if $\boldsymbol{\sigma} = (1, 1, \dots, 1)$*, then the performance of the approximation algorithm for* $\Pi_\times^\sigma$ *is* $(1 + \varepsilon)^{k-1} \prod_{i=1}^k \alpha_i$.

Proof: Let A be an $(\alpha_1, \dots, \alpha_k)$-approximation algorithm for the problem Π. We first consider Π_+^σ. The idea how to obtain an approximation algorithm for the unicriterion problem Π_+^σ is the following: Let x^* be an optimal solution for a given instance of Π_+^σ of value $\sum_{i=1}^k \sigma_i f_i(x^*)$. Then, x^* is a feasible solution to the instance of the k-criteria problem Π of minimizing $f_1(x)$ subject to the constraints that $f_i(x) \leq F_i := f_i(x^*)$. The feasibility of x^* allows us to argue that there is a "good" optimal solution to this instance of Π, meaning that its objective value under f_1 is at most that of x^*. Using the k-criteria algorithm A would enable us to find a "good" solution where also the linear combination of the objective values f_i, $i = 1 \dots, k$ via $\boldsymbol{\sigma}$ is not too bad compared to x^*. Unfortunately, we do not know the "right bounds" $f_i(x^*)$ to use as upper bounds. Thus, we construct an appropriate search space of bounds which is large enough to find good bounds and small enough to be searchable exhaustively in polynomial time.

We formalize the above idea in Algorithm 3.2. Since $\varepsilon > 0$ is fixed and k as well as the σ_i are constants, it follows easily that Algorithm 3.2 can be implemented to run in polynomial time.

Algorithm 3.2 Converting a k-criteria algorithm into a unicriterion algorithm.

Input: A k-criteria approximation algorithm A for Π, an instance of Π^{σ}_{+}

for $i \leftarrow 2, \dots, k$ **do**
  $r_i \leftarrow \lceil \log_{(1+\varepsilon)} B_i \rceil$
  $M_i \leftarrow \{ (1+\varepsilon)^j : j = 0, \dots, r_i \}$
end for
$M \leftarrow M_2 \times \cdots \times M_k$
For a vector $\mathbf{z} = (z_2, \dots, z_k)$ denote by $\mathsf{A}(\mathbf{z})$ the subgraph returned by the k-criteria algorithm A, when called with the bounds z_i on the values of the objectives f_i, $i = 2, \dots, k$.
Find $\mathbf{z}^* \in M$ such that $\sum_{i=1}^{k} \sigma_i f_i(\mathsf{A}(\mathbf{z}))$ is minimum among all $z \in M$.
return $\mathsf{A}(\mathbf{z}^*)$

Let x^* be an optimal solution to the unicriterion problem Π^{σ}_{+}. Consider the call in Algorithm 3.2 to the k-criteria algorithm A, when the parameter $\mathbf{z}$ satisfies:

$$f_i(x^*) \leq z_i < (1+\varepsilon) f_i(x^*), \qquad i = 2, \dots, k. \tag{3.5}$$

Observe that by construction of the algorithm, such a call will be issued. The algorithm A is hence applied to the k-criteria problem of finding a subgraph from

$$\{ x \in \Gamma : f_i(x) \leq z_i, i = 2, \dots, k \}, \tag{3.6}$$

having minimum possible value for f_1. Because of (3.5), the optimal solution x^* to the unicriterion problem Π^{σ}_{+} is contained in the feasible set (3.6), providing in particular that this set is nonempty. Thus, by the performance guarantee of A, the k-criteria approximation algorithm will return a subgraph $\hat{x} := \mathsf{A}(\mathbf{z})$ from

$$\{ x \in \Gamma : f_i(x) \leq \alpha_i \cdot z_i, i = 2, \dots, k \} \tag{3.7}$$

such that $f_1(\hat{x}) \leq \alpha_1 \cdot \mathrm{OPT}$. Here, OPT denotes the minimum objective value under f_1 of a solution from (3.6). Combining that $\hat{x}$ is contained in (3.7) with (3.5) yields

$$f_i(\hat{x}) \leq (1+\varepsilon)\alpha_i \cdot f_i(x^*), \qquad i = 2, \dots, k. \tag{3.8}$$

Moreover, since we have seen that the optimal solution x^* of the unicriterion problem is contained in the set (3.6) of feasible solutions of the instance of the k-criteria problem just constructed, we obtain that OPT $\leq f_1(x^*)$. Thus, again by the performance of A this implies

$$f_1(\hat{x}) \leq \alpha_1 \cdot \mathrm{OPT} \leq \alpha_1 \cdot f_1(x^*). \tag{3.9}$$

Combining (3.8) and (3.9) gives us:

$$\begin{aligned}\sum_{i=1}^{k} \sigma_i f_i(\hat{x}) &\leq \alpha_1 \cdot \sigma_1 f_1(x^*) + (1+\varepsilon) \sum_{i=2}^{k} \alpha_i \cdot \sigma_i f_i(x^*) \\ &\leq (1+\varepsilon) \cdot \max\{\alpha_1, \dots, \alpha_k\} \cdot \sum_{i=1}^{k} \sigma_i f_i(x^*).\end{aligned}$$

Since Algorithm 3.2 chooses the $\mathbf{z}^* \in M$ such that $\sum_{i=1}^{k} \sigma_i f_i(\mathsf{A}(\mathbf{z}))$ is minimum among all $\mathbf{z} \in M$, the claimed performance guarantee now follows in the case of Π_+^σ.

The proof for the problem $\Pi_\times^\sigma$ follows along the same lines. We just have to change Step 7 to select $\mathbf{z}^*$ such that $\prod_{i=1}^{k} f_i^{\sigma_i}(\mathsf{A}(\mathbf{z}^*))$ is minimum among all $\mathbf{z} \in M$. Then, using the same arguments as above, we get from (3.8) and (3.9) that

$$\begin{aligned}\prod_{i=1}^{k} f_i^{\sigma_i}(\hat{x}) &\leq \alpha_1^{\sigma_1} \cdot f_1^{\sigma_1}(x^*) \cdot \prod_{i=2}^{k} (1+\varepsilon)^{\sigma_i} \alpha_i^{\sigma_i} \cdot f_i^{\sigma_i}(x^*) \\ &= \left((1+\varepsilon)^{\sum_{i=2}^{k} \sigma_i} \prod_{i=1}^{k} \alpha_i^{\sigma_i} \right) \cdot \prod_{i=1}^{k} f_i^{\sigma_i}(x^*).\end{aligned}$$

This completes the proof. □

Theorem 3.10 establishes a connection between the approximability of a multicriteria problem and certain unicriterion problems. In particular, if a bicriteria problem is "well approximable" with performance (α_1, α_2), α_1 and α_2 constants, then there is also a "good" constant factor approximation for the problem of minimizing the sum of the two objectives.

The converse is not true in general. To see this, we will provide an example of a problem Π_+ of minimizing the objective $f := f_1 + f_2$, which is well-approximable, in fact polynomial time solvable. In contrast, the bicriteria problem Π of minimizing f_1 subject to a constraint on the value of f_2 turns out to be very hard to approximate.

Consider the bicriteria problem Π of finding an independent dominating set U in a graph $G = (V, E)$ minimizing $f_1(U) := |U|$ subject to the constraint that $f_2(U) := |V - U| \leq F_2$. The problem Π_+ where f_1 and f_2 are functionally combined to $f := f_1 + f_2$ consists of finding an independent dominating set U minimizing $f_1(U) + f_2(U) \equiv |V|$. Clearly, Π_+ can be solved in polynomial time, e.g. by the trivial algorithm that simply outputs *any* independent dominating set in the graph G. Thus, Π_+ can be deemed "well-approximable".

In contrast, Π shows a different behavior. If we set the bound F_2 on the cardinality of $|V - U|$ to be $F_2 := |V|$, then Π becomes the problem of finding an independent dominating set of minimum cardinality. This problem is hard to approximate [BS94] : Unless $\mathsf{P} = \mathsf{NP}$, for any $\varepsilon > 0$ there can be no polynomial time approximation algorithm with performance of $|V|^{1/6-\varepsilon}$ for this problem. Thus, the bicriteria problem Π contains as a special case a problem which is hard to approximate, while the unicriterion problem Π_+ is well-approximable.

The above discussion also shows that the generic transformation of an approximation algorithm for a k-criteria problem does not always yield the best possible performance for the unicriterion problems Π_+^σ and $\Pi_\times^\sigma$. This can also be seen by considering the constrained minimum spanning tree problem from Example 3.3. The related unicriterion problem Π_+ consists of finding a minimum spanning tree with respect to the edge weights $c(e) + d(e)$, $e \in E$. We could use the $(2, 2)$-approximation algorithm from [MR+95a] for the constrained minimum spanning tree problem to obtain an approximation algorithm with a performance of $2 + \varepsilon$ for Π_+. However, the problem can be solved exactly in polynomial time by applying Kruskal's algorithm where the weight of edge e is given by $c(e) + d(e)$.

3.4 Hardness of Approximation

The performance of an approximation algorithm can be used to judge the difficulty of an NP-hard optimization problem. Intuitively, if we have an approximation algorithm with a good performance for a problem, the problem might be deemed easier. This intuition can be formalized by establishing *lower bound results* on the approximability of a problem. Such lower bound results are commonly also referred to as *non-approximability* or *hardness results*. Hardness results vary from very weak to relatively strong depending on which degree of approximability they rule out. The most common assumption under which they are derived is that $\mathsf{P} \neq \mathsf{NP}$. Another widely used assumption is that $\mathsf{NP} \not\subseteq \mathsf{DTIME}(n^{OO(\log\log n)})$, which is regarded as almost equivalent to $\mathsf{P} \neq \mathsf{NP}$.

We have already mentioned a hardness result in the discussion about multicriteria versus unicriterion approximation in Section 3.3 above: For any fixed $\varepsilon > 0$ the problem of finding an independent dominating set of minimum cardinality can not be approximated within $|V|^{1/6-\varepsilon}$ unless $\mathsf{P} = \mathsf{NP}$. In this section we discuss typical forms of such hardness results.

The weakest hardness result as far as the (multiplicative) performance of an algorithm is concerned is to rule out the existence of a *polynomial time approximation scheme* (PTAS). For any fixed $\varepsilon > 0$, a PTAS runs in time polynomial in the input size and outputs a solution with multiplicative error at most $1 + \varepsilon$. If the running time of the PTAS can be bounded by a polynomial in the input size *and* $1/\varepsilon$, then it is called a *fully polynomial time approximation scheme* (FPTAS). Recently, by the means of interactive proof systems Arora et al. [AL+92] were able to establish the following non-approximability result.

Theorem 3.11 *For every* MaxSNP*-hard problem there exists some* $\varepsilon > 0$ *such that achieving an approximation with performance* $1 + \varepsilon$ *is* NP*-hard.*

Thus, the existence of a PTAS for a MaxSNP-hard problem implies that $\mathsf{P} = \mathsf{NP}$. Hence, showing that a problem is MaxSNP-hard is an (important) tool for ruling out the existence of a PTAS.

While a proof of MaxSNP-hardness for a problem Π provides the information that we can not approximate the problem better than $1+\varepsilon$ for some ε, the exact value of ε is not provided by means of Theorem 3.11. Thus, a stronger hardness result is to show that Π has no polynomial time approximation algorithm with performance $1 + \varepsilon$ and give a value of ε which is as large as possible with this property. Results of these types can be found in various chapters, e.g. Chapter 4 and Chapter 5.

The next stronger type of non-approximability result is to show that a problem can not be approximated with *any* constant performance or, even stronger, not within a logarithm of the input size. Such a hardness result was established by Lund and Yannakakis [LY93] for the MIN SET COVER problem, which can be stated as follows: Given a collection of sets over a finite ground set, the objective is to find a minimum size subcollection that contains all the ground elements. The hardness result from [LY93] for MIN SET COVER was recently improved by Feige [Fei96] in the following way:

Theorem 3.12 *Unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$*, the* MIN SET COVER *problem, with a ground set* Q *of size* $|Q|$*, can not be approximated in polynomial time within a performance factor of* $\beta < \ln |Q|$.

We will establish similar logarithmic lower-bound results to the one named for MIN SET COVER in Theorem 3.12 for the node-upgrading problems in Chapter 6.

The strongest type of non-approximability results are those which show that achieving performances nearly as bad as the trivial ones is already hard. A typical example is the MAX CLIQUE problem: Given an undirected graph, the problem is to determine a clique of maximum cardinality. Notice that the trivial algorithm which always outputs a single vertex has a performance of $|V|$, where V is the vertex set of the graph given in the input. We state the most recent result obtained by Bellare and Sudan [BS94].

Theorem 3.13 *Unless $\mathsf{P} = \mathsf{NP}$, for all $\varepsilon > 0$ the problem* MAX CLIQUE *does not have a polynomial time approximation algorithm with performance guarantee $|V|^{1/6-\varepsilon}$, where $|V|$ denotes the number of vertices in the input graph.*

Hardness results which basically rule out anything but a trivial ratio can be found for instance in Chapters 4 and 5. In Chapter 4, we show that the problem p-CENTER$^{(\alpha)}$ under study can not be approximated within $\mathcal{O}(f(|V|))$, where f is any polynomial time computable function and V is the vertex set of the input graph.

3.5 Algorithmic Techniques

This section starts with some bad news: no general approximation techniques are known which can be successfully applied to *all* hard problems. In contrast, the design of an approximation algorithm mostly depends heavily on the structure of the corresponding optimization problem. Nevertheless, there are some ideas that prove to be useful. We will sketch them below, since we will repeatedly adopt and apply them as tools in the design of the approximation algorithms presented in this thesis.

Relaxed Decision Procedures and Certificates of Failure

A very useful definition for the design of approximation algorithms was introduced by Hochbaum and Shmoys in [HS86]:

Definition 3.14 (Relaxed Procedure)
*A **ρ-relaxed procedure** TEST for a (unicriterion) minimization problem Π has the following features. Let $\mathrm{OPT}(I)$ denote the optimum value of any feasible solution for an instance I of Π. Given a candidate value M to be $\mathrm{OPT}(I)$, the procedure TEST(M) has two possible outcomes, namely "certificate of failure" ("COF") or success. These two outcomes have the following two implications:*

1. *If TEST returns a COF then it is certain that $\mathrm{OPT}(I) > M$.*

2. *If TEST succeeds, it outputs a solution whose objective function value is at most $\rho \cdot M$.*

The case of maximization problems is treated similarly.

Observe that a ρ-relaxed procedure, called with a test value $\mathrm{OPT}(I)/\rho \leq M < \mathrm{OPT}(I)$, has the choice of returning a COF or delivering an answer of objective value at most $\rho \cdot M$.

Let Π be a minimization problem. Suppose that, given an instance I of Π we know that $\mathrm{OPT}(I)$ is an integer contained in the interval $[m_I, M_I]$. Then the existence of a ρ-relaxed procedure TEST suggests the following approach to obtain an approximation algorithm for Π: We search the interval $[m_I, M_I]$ for $\mathrm{OPT}(I)$ by examining possible candidate values M with the help of Procedure Test. If the test fails, we can conclude that M is strictly smaller than the optimum value. On the other hand, if TEST succeeds, it guarantees a solution of value at most $\rho \cdot M$. Thus, if we find the minimum candidate value in the interval such that TEST succeeds, we will obtain a ρ-approximation for the optimum solution.

Of course, the above sketched idea is by no means a complete guide to a polynomial time approximation algorithm. To guarantee polynomiality of the algorithm obtained this way, we will have to bound the running time of TEST and the number of calls to the relaxed procedure TEST appropriately. Also, if $\mathrm{OPT}(I)$ is not necessarily an integer, one will have to undertake appropriate steps. Nevertheless, the idea of relaxed decision procedures often proves to be a very useful tool. We will use them for quite a few of our approximation algorithms.

One way to keep the number of calls to a relaxed test procedure small is to use *binary search*. However, we must make sure that such a binary search will work. Similarly as above, assume that $\mathrm{OPT}(I)$ is an integer in the interval $[m_I, M_I]$. The possible problem is that for a test parameter M satisfying $\mathrm{OPT}(I)/\rho \leq M < \mathrm{OPT}(I)$ we have no knowledge about whether the ρ-relaxed procedure TEST will deliver a solution or output COF.

Nevertheless, it turns out that this is not grave. We can still run a binary search as shown in Algorithm 3.3 simply ignoring the uncertainty caused by the relaxed procedure.

Lemma 3.15 *Suppose that* $\mathrm{OPT}(I)$ *is an integer in the interval* $[m_I, M_I]$ *and* TEST *is a* ρ*-relaxed procedure. Then the "binary search" in Algorithm 3.3 terminates with a value of high satisfying high* $\leq \mathrm{OPT}(I)$ *and* $\mathrm{TEST}(high) \neq$ *COF. The objective function value of the solution output at the end of the binary search is at most* $\rho \cdot \mathrm{OPT}(I)$.

Proof: If the test in Step 1 succeeds, i.e. if $\mathrm{TEST}(m_I) \neq \mathrm{COF}$, then by the assumption that $\mathrm{OPT}(I) \in [m_I, M_I]$ we have $\mathrm{OPT}(I) = m_I$ and Algorithm 3.3

Algorithm 3.3 Binary search guided by a relaxed procedure.

Input: A lower bound m_I and an upper bound M_I for the optimum value

if $\text{TEST}(m_I) \neq \text{COF}$ **then**
  return $\text{TEST}(m_I)$
end if
$high \leftarrow M_I$
$low \leftarrow m_I$
while $(high - low) > 1$ **do**
  $i \leftarrow \left\lceil \frac{high+low}{2} \right\rceil$
  if $\text{TEST}(i) = \text{COF}$ **then**
    $low \leftarrow i$
  else
    $high \leftarrow i$
  end if
end while
return $\text{TEST}(high)$

returns an optimal solution. Thus, we will assume for the rest of the proof that $\text{TEST}(m_I) = \text{COF}$.

As an invariant throughout the algorithm we have $\text{TEST}(low) = \text{COF}$ and $\text{TEST}(high) \neq$COF. In particular, upon termination $\text{TEST}(high)$ is also not a COF.

To establish the rest of the lemma we distinguish between two cases. In the first case, the procedure TEST is called for some M in the critical range, i.e. $\text{OPT}/\rho \leq M < \text{OPT} := \text{OPT}(I)$ and does not return a COF for this test value. Then, by construction of the algorithm, $high \leq M \leq \text{OPT}$ upon termination, which shows the claim. In the second case, $\text{TEST}(M) = \text{COF}$ for *all* $\text{OPT}/\rho \leq M < \text{OPT}$ that are probed during the search. In this case, it is easy to see that $high = \text{OPT}$, when the algorithm terminates.

The fact that the objective function value of the solution output at the end of the search is at most $\rho \cdot \text{OPT}$ is a direct consequence of $\text{TEST}(high) \neq \text{COF}$ and the fact that TEST is ρ-relaxed. □

Lemma 3.15 enables us to perform a binary search even if we are dealing with relaxed decision procedures instead of exact ones. Thus, in the sequel we can restrict ourselves to showing that a certain procedure is relaxed (for some value ρ) and then simply refer to the binary search technique in order to make a fast search of a certain range possible.

Bottleneck Graphs and Powers of Graphs

Another important technique which we will use a couple of times is the notion of the *power of a graph* also introduced by Hochbaum and Shmoys [HS86], which is defined in the sequel:

Definition 3.16 (Power of a Graph)
Let $H = (V, E_H)$ *be a subgraph of the complete graph* $G = (V, E)$*. For* $t \in \mathbb{N}$ *the* **k*th power*** $H^k = (V, E_H^k)$ *of* H *contains an edge between the vertices* u *and* v *if and only if there is a path containing at most* k *edges in* H *connecting* u *and* v*.*

Definition 3.17 (Bottleneck Graph,Bottleneck Weight)
Let $G = (V, E)$ *be a complete graph with nonnegative weights* $d(e)$ *on the edges* $e \in E$*. The* ***bottleneck graph*** *of* G *with respect to the weights specified by* d *and a given bound* D*, denoted by* $\text{Bottleneck}(G, d, D)$*, is defined to be the graph consisting of the same set of vertices as* G *but containing only those edges* $e \in E$ *such that* $d(e) \leq D$*.*

Given an edge weighted graph as above, we denote by $\max(G)$ *its* ***bottleneck weight****, that is* $\max(G) := \max\{\, d(e) : e \in E \,\}$*.*

As is standard in the literature, we say that a nonnegative edge-weight function d satisfies the *triangle inequality*, if we have $d(v, w) \leq d(v, u) + d(u, w)$ for all $(v, w), (v, u), (u, w) \in E$.

Observation 3.18 *Let* H *be a subgraph of the complete graph* $G = (V, E)$ *with edge-weights* $d(e)$ *(*$e \in E$*) satisfying the triangle inequality. Then the weight of the heaviest edge in* H^k *is at most* k *times the weight of the heaviest edge in* H*, i.e.,*

$$\max(H^k) \leq k \cdot \max(H).$$

Powers of a graph (in conjunction with bottleneck graphs) are usually very helpful if dealing with so called *bottleneck problems*. Following [HS86], these are problems where we wish to find a subgraph (of a complete graph) satisfying certain constraints such that the weight of the heaviest edge included in the subgraph is minimized.

The (very) basic idea behind the definitions given above is the following: Assume that we are given a bottleneck problem on a complete graph $G = (V, E)$ with edge weights $d(e)$ ($e \in E$) satisfying the triangle inequality. The problem is to find an edge subgraph H of G from the class Γ of minimum bottleneck weight $\max(H) = \max\{\, d(e) : e \in H \,\}$. It is clear that the optimal solution value must be contained in the set $\{\, d(e) : e \in E \,\}$. Suppose that we can construct a procedure TEST with the following properties: If $\text{TEST}(M)$ is a COF, then the optimum value satisfied $\mathrm{OPT}(I) > M$. If TEST succeeds, it outputs an edge subgraph of the kth power of the bottleneck graph $\mathrm{Bottleneck}(G, d, M)$. Using Observation 3.18 it is easy to show that under these circumstances TEST is a k-relaxed decision procedure (see Definition 3.14). Thus, if we find the minimum $M \in \{ d(e) : e \in E \}$ such that $\mathrm{TEST}(M)$ succeeds, this will give us a k-approximation for the bottleneck problem.

Solution Based Decomposition Methods

The idea of a *solution based decomposition method* can be sketched as follows. Decompose the optimal solution of a hard optimization problem into one or more solutions of a simpler problem. The approximation algorithm then uses an efficient algorithm to obtain good solutions for the simpler problems. The basic idea is to use the existence of an optimal solution to prove that for each of the single pieces it is possible to find a sufficiently good solution.

An example for the application of a solution based decomposition method is the classical approximation algorithm of Christofides [Chr76] for the traveling salesperson problem with triangle inequality. In this problem, one is given a complete graph $G = (V, E)$ and a nonnegative edge-weight function d obeying the triangle inequality. The problem is to find a closed tour of minimum total weight visiting each vertex exactly once.

Christofides' algorithm first computes a minimum spanning tree in the graph. Then, a minimum total weight perfect matching is computed for the subgraph $G[D]$ induced by the set D of those nodes in the tree that have odd degree. We can shortcut the optimal tour to obtain a tour of at most the same length on the vertices of D. Then, this tour *decomposes* into two perfect matchings on D, where the lighter one of these has weight at most half that of the optimum tour length. Thus, the minimum total weight perfect matching found by the algorithm has weight at most one half times the optimal tour length. Since the weight of the minimum spanning tree computed in the first step can not exceed that of the optimal tour, adding the edges of the matching to the edges of the tree will produce a graph of total edge weight at most $3/2$ times the optimal solution. The graph obtained this way is Eulerian.[4] Using shortcuts to convert an Eulerian cycle into a feasible tour, Christofides' algorithm finally produces a $3/2$-approximation to the optimal tour.

We use solution based decomposition methods in the design of our approximation algorithms in Chapters 6 and 8.

[4]A graph is Eulerian, if it contains an Eulerian cycle, i.e., a cycle containing each edge of the graph exactly once. By Euler's Theorem (see e.g. [PS82]) a graph is Eulerian if and only if it is connected and each vertex has even degree. Moreover, an Eulerian cycle in an Eulerian graph can be found in polynomial time (see e.g. [PS82]).

3.6 Alternatives to Approximation Algorithms

If a problem is NP-hard, this means that (if P $\neq$ NP) we can not devise a *polynomial time* algorithm which finds an *optimal solution* for *every instance* of the problem. As emphasized in the last sentence, there are a few spots where we might relax the requirements for an algorithm.

One way to do this are approximation algorithms, where we sacrifice optimality of a solution for the sake of a short running time. Another possibility is to *restrict* a problem to a certain subset of the possible instances. In the context of network and graph problems this usually means to consider the problem on a class of graphs with a simple structure. For instance, the MIN VERTEX COVER problem (see Example 3.4) can be solved in polynomial time on trees and, more generally, on graphs of bounded treewidth [BLW87].

We may also drop the requirement that the algorithm runs in *polynomial time*. There are algorithms, e.g. using dynamic programming techniques, which run in time slightly super-polynomial and are reasonably fast in practice [Mot92]. Yet another way to cope with hardness is to drop the requirement that all instances of a problem are treated *equally*. For some problems it is possible to show that certain heuristics will solve the problem with *high probability* [Coo94].

As the title suggests, the main focus of this thesis is on *approximation algorithms*. In some cases we will use the restriction of a problem, mainly to trees or treewidth-bounded graphs, to obtain improved approximation or polynomial time exact algorithms.

4

A Generalization of the p-Center Problem

4.1 Introduction

In this chapter we begin our study of *location problems*. The ubiquitous notion in the context of location problems is that of making an *optimal choice within a spatial context*. That means, the problems deal with the task of placing a number of facilities so as to achieve a desired property, such as minimizing the set up costs or a function of the distances between the facilities. The location problems studied in this thesis are all *discrete location problems* on graphs, also called *network location problems*. The decision maker can place facilities only at the nodes of a given network.

In contrast, *continuous location problems* consider the case where a number of facilities can be placed in the plane $\mathbb{R}^2$ or three-dimensional space $\mathbb{R}^3$, where distances are usually measured by an l_q-metric[1] for some $q \geq 1$.

In the following two chapters, we will investigate the complexity and approximability of a number of location problems on graphs. Our main goal is to devise efficient approximation algorithms for the problems under study. We counterbalance our approximation results with hardness results which for most cases show that the performances provided by our algorithms are best possible unless $\mathsf{P} = \mathsf{NP}$.

The present chapter is organized as follows. Section 4.2 contains the formal statement of the generalization of the p-center problem which is subject to our study. In Section 4.3 we discuss related work. Section 4.4 will present basic lemmas on k-independent and k-dominating sets in a graph G and its square G^2. In Section 4.5 we give our approximation algorithm for p-CENTER$^{(\alpha)}$ and establish its performance. Section 4.6 contains the hardness results.

[1]For $0 < q < \infty$, the l_q-distance between $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$ and $y = (y_1, \ldots, y_d) \in \mathbb{R}^d$ is defined by $l_q(x, y) := \left(\sum_{i=1}^{d} |x_i - y_i|^q\right)^{1/q}$. Moreover, $l_\infty(x, y) := \max_{1 \leq i \leq d} |x_i - y_i|$.

4.2 Preliminaries and Problem Definition

The p-center problem (p-CENTER for short) is one of the classical location problems (cf. [MF90]). The objective is to select a set of p centers such that the maximum distance of a non-center to its nearest center is minimized. The problem is used e.g. to model the placement of emergency facilities such as fire stations or hospitals, where the aim is to have a minimum guaranteed response time between a client and its center.

In this chapter, we study a generalization of p-CENTER, which we call the α-*neighbor* p-*center problem* or p-CENTER$^{(\alpha)}$ for short. Given an edge-weighted network, the target is to minimize the maximum distance of a client to its α nearest neighbors in the set of p centers. Clearly, for $\alpha = 1$, p-CENTER$^{(\alpha)}$ is identical to the classical p-center problem.

The generalized problem introduced in this chapter can be used to model the case when facilities are prone to failure. Since the center problems usually model the placement of emergency facilities, this issue becomes particularly important. Assume we have found a solution for p-CENTER$^{(\alpha)}$ such that each client has α centers (or facilities) within a distance of d^*. Then, if at most $\alpha - 1$ of the centers fail, still each client can be served by at least one facility within a distance of d^*. We now define the problem under study formally:

Definition 4.1 (α-Neighbor p-Center Problem)
An instance of the α***-neighbor*** p***-center problem***, p-CENTER$^{(\alpha)}$, *is given by an undirected complete graph* $G = (V, E)$, *nonnegative edge weights* $d(e)$ *(*$e \in E$*) and two integers* $1 \leq \alpha \leq p \leq |V|$. *The problem consists of finding a set* $P \subseteq V$ *of* p *nodes such that*

$$\mathrm{rad}^{(\alpha)}(P) = \max_{v \in V - P} d^{(\alpha)}(v, P)$$

is minimized, where

$$d^{(\alpha)}(v, P) = \min_{\substack{S \subseteq P \\ |S| = \alpha}} \max_{s \in S} d(s, v).$$

The subset of instances such that the distances obey the triangle inequality will be denoted by Δ-p-CENTER$^{(\alpha)}$. Notice that for any subset $P \subseteq V$ of p nodes we have that $\mathrm{rad}^{(1)}(P) \leq \mathrm{rad}^{(2)}(P) \leq \cdots \leq \mathrm{rad}^{(p)}(P)$.

The problem p-CENTER$^{(\alpha)}$ defined above is a *unicriterion problem*. We have stated it in a natural way as a problem of selecting a subset of the vertex set of a given graph G. The set Γ of subgraphs from Definition 3.1 consists of copies G_P of the input graph G, one for each p-element subset $P \subseteq V$, with one additional marking function on the vertices stating whether a vertex belongs to P or not. The objective function on G_P takes the value $\mathrm{rad}^{(\alpha)}(P)$.

As can be seen the rigid formulation of p-CENTER$^{(\alpha)}$ in the framework of Definition 3.1 is fairly close to the more intuitive one just given above. For convenience, we will speak of selecting a vertex subset P instead of choosing a graph G_P.

4.3 Related Work

Problems involving the placement of p facilities so as to minimize suitable cost measures have been studied extensively in the literature (see [AI+91, RK+93, BE95] and the references cited therein). These problems can roughly be divided into two main categories. The first category of problems involves selecting a set of p facilities so as to minimize (or maximize) the distance (cost) from the unselected sites to the selected sites. Problems that can be cast in this framework include the p-center problem [DF85, HS86], the p-cluster problem [Gon85, HS86, FG88] and the p-median problem [MF90, LV92]. The second category consists of problems where the goal is to select p facilities so as to optimize a certain cost measure defined on the set of selected facilities. Problems that can be cast in this framework include the p-dispersion problem [EN89, Tam91, RRT94, CH96], the p-minimum spanning tree problem [RS+94, GH94, AA+95, BCV95] and the p-compact location problem [AI+91, RK+93].

4.4 Some Basic Lemmas

Our first step towards an approximation algorithm for p-CENTER$^{(\alpha)}$ is to establish some key results about independent and dominating sets in a graph $G = (V, E)$ and its square G^2.

To this end we first extend the notion of independence in graphs. Recall that a set $U \subseteq V$ is called *independent*, if for any pair u, v of nodes from U there is no edge connecting u and v.

Definition 4.2 *A* ***k-independent set*** *is a subset $U \subseteq V$ such that every node $v \in U$ has at most $k-1$ neighbors in U.*

Consequently, a 1-independent set is a classical independent set. We use the term *maximal k-independent set* to denote a k-independent set that is maximal with respect to inclusion. Given a graph $G = (V, E)$ with $n := |V|$ nodes, we can always find a maximal k-independent set in polynomial time, in fact in $\mathcal{O}(n^2)$ time, simply by choosing a node and then adding nodes repeatedly, until any further addition of a single node would destroy the k-independence of our set.

As the definition of k-independence extends the classical notation of independence, there is an analogon for dominating sets. Recall that a set $D \subseteq V$ is called a *dominating set*, if any node in V either belongs to D or has a neighbor in D.

Definition 4.3 *A* ***k-dominating set*** *is a set $D \subseteq V$ such that each node $v \in V - D$ has at least k neighbors in D.*

It is easy to see that a maximal independent set U is also a dominating set. For $k > 1$, in general it is not true that a maximal k-independent set is also k-dominating, see e.g. the simple example depicted in Figure 4.1. The set $U = \{x, y, z\}$ is maximal 3-independent, because v can not be added without destroying the 3-independence, but not 3-dominating, since v has only one neighbor in U.

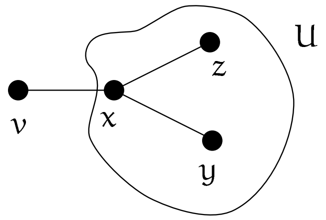


Figure 4.1: A maximal 3-independent set which is not 3-dominating.

We will now prove that, although a maximal k-independent set need not be k-dominating in G, we can deduce this property for the same set in the square graph G^2.

Lemma 4.4 *Let U be a maximal k-independent set such that $|U| \geq k$. Then U is a k-dominating set in G^2.*

Proof: We show that each node $v \in V - U$ has a neighbor w_1 in U such that this neighbor is adjacent to $k - 1$ nodes $w_2, \ldots, w_k$ in U. It follows that the nodes $w_1, \ldots, w_k$ will be neighbors of v in G^2 and this establishes the claim.

In fact, if there existed a node v that is not adjacent to *any* node in U with $k - 1$ neighbors in U, we could add v to U without destroying the property of k-independence contradicting the fact that U is maximal. □

The following lemma states a key relation between the k-dominating sets in G and the k-independent sets in the square G^2:

Lemma 4.5 *Let D be a k-dominating set in G. Then $|U| \leq |D|$ for any k-independent set U in G^2.*

Proof: If $U \subseteq D$ then the claim of the lemma is trivial.

If U is not contained in D, then choose an arbitrary node $u \in U - D$ and let $S := N_G(u) \cap D$. Clearly $|S| \geq k$, because D is a k-dominating set. Define $C := N_G(S) \cap (V - D)$ to be the set of vertices in $V - D$ that are adjacent to the set S (see Figure 4.2 for an illustration). Then any node in $C \cup S$ is adjacent to u in G^2 and thus U can contain at most k vertices from $C \cup S$. On the other hand, we have seen that $|D \cap (C \cup S)| = |S| \geq k$.

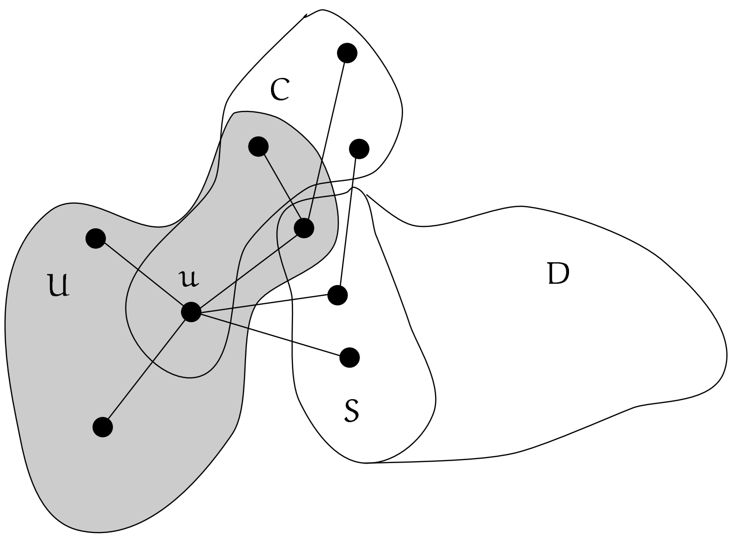


Figure 4.2: The sets S and C in the proof of Lemma 4.5.

Now consider the graph $\hat{G} := G[V - (C \cup S)]$. We claim that $\hat{D} := D - (C \cup S) = D - S$ is a k-dominating set in $\hat{G}$. To see this, consider an arbitrary node v from $\hat{G}$ that is not contained in $\hat{D}$. Then $v \in V - D$. The node v has at least k neighbors in G that are contained in D, since again D is a k-dominating set in G. None of these neighbors can be contained in S, because otherwise we would have $v \in N_G(S) \cap (V - D) = C$ and thus v were not contained in $\hat{G}$. Hence $N_G(v) \cap D \subseteq D - S = \hat{D}$ and all the neighbors of v in D are still present in $\hat{D}$.

The set $\hat{U} := U - (C \cup S)$ is clearly k-independent in $\hat{G}$. Thus, we can repeat the above construction for $D := \hat{D}$ and $U := \hat{U}$ until we obtain that $\hat{U} \subseteq \hat{D}$. Since in each step we delete at most k nodes from U and at least k nodes from D it follows that $|U| \leq |D|$. □

4.5 An Approximation Algorithm

In this section we will present our algorithm to approximate Δ-p-CENTER$^{(\alpha)}$ and use the results of the two lemmas from the last section to analyze its performance guarantee. The algorithm relies on *bottleneck graphs* (see Definition 3.17) and *powers of graphs* (see Definition 3.16).

In the sequel we will use the following notation: Given an instance of Δ-p-CENTER$^{(\alpha)}$, we let $P^* \subseteq V$ denote an optimal placement of p nodes and use OPT for the optimal solution value, that is OPT := $\mathrm{rad}^{(\alpha)}(P^*)$.

First there is one very important observation: By definition of the objective function $\mathrm{rad}^{(\alpha)}$, the optimal function value OPT must equal the weight of some edge. Assume that the edges of G satisfy $d(e_1) \leq d(e_2) \leq \cdots \leq d(e_m)$, where $m = \frac{n(n-1)}{2}$.

The algorithm now searches the m element set $\{d(e_1), \ldots, d(e_m)\}$ with the help of a relaxed procedure (see Definition 3.14). Given a number M, the relaxed procedure either tells us that OPT $> M$ by returning a COF ("certificate of failure") or delivers a solution of cost at most $4M$ ($2M$ for $\alpha = 1$). The algorithm returns the minimum value i such that the outcome of the relaxed procedure if called with $M := d(e_i)$ is not a COF. This is done by applying the binary search of Algorithm 3.3.

Algorithm 4.1 shows the main procedure, the relaxed test procedure is depicted in Algorithm 4.2 on the facing page.

Algorithm 4.1 Approximation algorithm for p-CENTER$^{(\alpha)}$.

Input: A complete graph $G = (V, E)$, an edge weight function d, an integer p and an integer α

Sort the edges of G such that $d(e_1) \leq d(e_2) \leq \cdots \leq d(e_m)$
Perform a binary search to find the minimum $i \in \{1, \ldots, m\}$, such that Algorithm 4.2 called with the test value $d(e_i)$ does not return COF
Let U be the set found by the binary search
return U

Algorithm 4.2 Relaxed test procedure for p-CENTER$^{(\alpha)}$.

Input: A complete graph $G = (V, E)$, an edge weight function d, an integer p, and a test value M

$G' \leftarrow \text{Bottleneck}(G, d, M)$
$U \leftarrow$ a maximal α-independent set in $(G')^2$
if $|U| > p$ **then**
  return COF
end if
if $|U| < p$ **then**
  Add nodes arbitrarily to make $|U| = p$
end if
return U

Performance Guarantee

We will now prove that the test procedure from Algorithm 4.2 is in fact a valid 4-relaxed decision procedure for $\alpha > 1$ and 2-relaxed for $\alpha = 1$. To this end, we first establish the following lemma.

Lemma 4.6 *If Algorithm 4.2 called with some number M returns a COF, then* $\text{OPT} > M$.

Proof: Assume that the algorithm returns a COF, but nonetheless $\text{OPT} \leq d(e_i)$. Let $P^* = \{v_1^*, \dots, v_p^*\}$ be a set of p centers in G such that $\text{rad}^{(\alpha)}(P^*) = \text{OPT}$. By definition of the solution value OPT it follows that P^* is an α-dominating set in the bottleneck graph G', defined by $G' := \text{Bottleneck}(G, d, M)$.

Algorithm 4.2 can only return a COF, if it finds an α-independent set U in $(G')^2$ that contains more than p elements. But according to Lemma 4.5 such a set can not exist in $(G')^2$. □

Thus, we have shown that the first part from Definition 3.14 is satisfied for our test procedure. We now show that also the second part holds.

Lemma 4.7 *Assume that Algorithm 4.2 returns a set S of nodes and not a COF. Then* $\text{rad}^{(\alpha)}(S) \leq 4M$. *For $\alpha = 1$, the better estimate* $\text{rad}^{(1)}(S) \leq 2M$ *holds true.*

Proof: By definition of the bottleneck graph $G' = \text{Bottleneck}(G, d, M)$, each edge weight in G' is at most M, i.e., $\max(G') \leq M$. By Observation 3.18, we have that the maximum edge weights in $(G')^2$ and $(G')^4$ are $2M$ and $4M$, respectively.

If $\alpha \geq 2$, we can use Lemma 4.4 to deduce that S is an α-dominating set in $((G')^2)^2 = (G')^4$. Thus, each vertex from $V - S$ is adjacent to at least α nodes from S via edges of weight at most $4M$.

For $\alpha = 1$, it follows that S is a dominating set in $(G')^2$ as was remarked at the beginning of the last section (page 46). The claim of the lemma now follows. □

Corollary 4.8 *Algorithm 4.2 is a* 4*-relaxed procedure. If* $\alpha = 1$*, then the procedure is even* 2*-relaxed.*

Proof: Follows from Lemmas 4.6 and 4.7. □

From Lemma 3.15 we now obtain the following lemma on the performance of our main approximation Algorithm 4.1.

Lemma 4.9 *Algorithm 4.1 as applied to* Δ-p-CENTER$^{(\alpha)}$ *has a performance of* 4*. For* $\alpha = 1$*, the performance improves to* 2*.*

Running Time

We now estimate the running time of Algorithm 4.1. As before, let n denote the number of nodes and $m := \frac{n(n-1)}{2}$ denote the number of edges in the input graph.

Sorting the edges according to their weights in Step 1 of the the main Algorithm 4.1 takes $\mathcal{O}(m \log m)$ time. During the binary search, $\mathcal{O}(\log m)$ calls to the test procedure Algorithm 4.2 are issued. Each time Algorithm 4.2 is called, we have to compute a bottleneck graph G'. This can be accomplished almost trivially in $\mathcal{O}(m)$ time by comparing the weight of each edge with the bound M. Then, the square graph $(G')^2$ has to be computed. We have the following simple observation.

Observation 4.10 *Let* G *be a graph and let* A *denote its adjacency-matrix with added ones on the diagonal. Then the adjacency-matrix of* G^2 *is given by* $A \oplus (A \otimes A)$*, where* $\oplus$ *and* $\otimes$ *denote the boolean addition and multiplication of two 0/1-matrices respectively.*

Thus, it follows that by using a fast 0/1-matrix multiplication, we can compute $(G')^2$ efficiently. Currently, the fastest known algorithm to compute the product of two 0/1-matrix of size $n \times n$ runs in time $\mathcal{O}(n^\tau)$, where $\tau \approx 2.376$ [CW87]. We have already noted that a maximal α-independent set can be found in $\mathcal{O}(n^2)$ time. Thus, the total time effort for a single call to Algorithm 4.2 is in $\mathcal{O}(n^\tau)$.

This leads to an overall running time of $\mathcal{O}(m \log m + n^\tau \log m) \subseteq \mathcal{O}(n^\tau \log n)$, where $\tau \approx 2.376$.

We summarize the results of this section in the following theorem.

Theorem 4.11 *Algorithm 4.1 is a $\mathcal{O}(n^\tau \log n)$ time approximation algorithm for Δ-p-CENTER$^{(\alpha)}$ with a performance of 4. For $\alpha = 1$, the performance improves to 2.* □

4.6 Hardness Results

In this section we will establish a non-approximability result for p-CENTER$^{(\alpha)}$, which shows that the algorithm presented in Section 4.5 provides essentially the best possible approximation guarantee for the problem under the assumption that $\mathsf{P} \neq \mathsf{NP}$.

Theorem 4.12 *Unless $\mathsf{P} = \mathsf{NP}$, for any fixed $\alpha \leq p$ there can be no polynomial time approximation algorithm for Δ-p-CENTER$^{(\alpha)}$ with a performance guarantee of of $2 - \varepsilon$ for any fixed $\varepsilon > 0$.*

Moreover, p-CENTER$^{(\alpha)}$ can not be approximated with a performance of $f(|V|)$, where f is any polynomial time computable function.

Proof: We will establish the hardness result by providing a polynomial time reduction from DOMINATING SET, a well known NP-complete problem (cf. [GJ79, Problem GT2]). An instance of DOMINATING SET is given by an undirected graph $G' = (V', E')$ and an integer s. The problem is to decide, whether there is a dominating set of size at most s, i.e., whether there is a set $D \subseteq V'$ of cardinality at most s such that every vertex in V' either belongs to D or has a neighbor in D.

Assume that A is an algorithm for p-CENTER$^{(\alpha)}$ with a performance guarantee of $f(|V|)$, where f is a polynomial time computable function.

Let I' be any instance of DOMINATING SET. We now construct an instance I of p-CENTER$^{(\alpha)}$ in the following way: We choose $|V'|$ pairwise disjoint sets $N_v := \{w_v^{(1)}, \ldots, w_v^{(\alpha-1)}\}$ $(v \in V')$ with $N_v \cap V' = \emptyset$. We then let $V := V' \cup \left(\bigcup_{v \in V'} N_v\right)$, $p := s + (\alpha - 1)|V'|$ and define $G = (V, E)$ to be a complete graph on $|V|$ nodes. The edge-weights $d(e)$ are given by

$$d(u, v) := \begin{cases} 1 & \text{if } u, v \in V' \text{ and } (u, v) \in E', \\ 1 & \text{if } v \in V' \text{ and } u \in N_v \text{ or vice versa}, \\ f(|V|) + \varepsilon' & \text{otherwise}, \end{cases}$$

where we choose $\varepsilon' > 0$ arbitrarily. Figure 4.3 illustrates the transformation from G' to G. All edges shown have weight 1, the edges not drawn in the figure have weight $f(|V|) + \varepsilon'$.

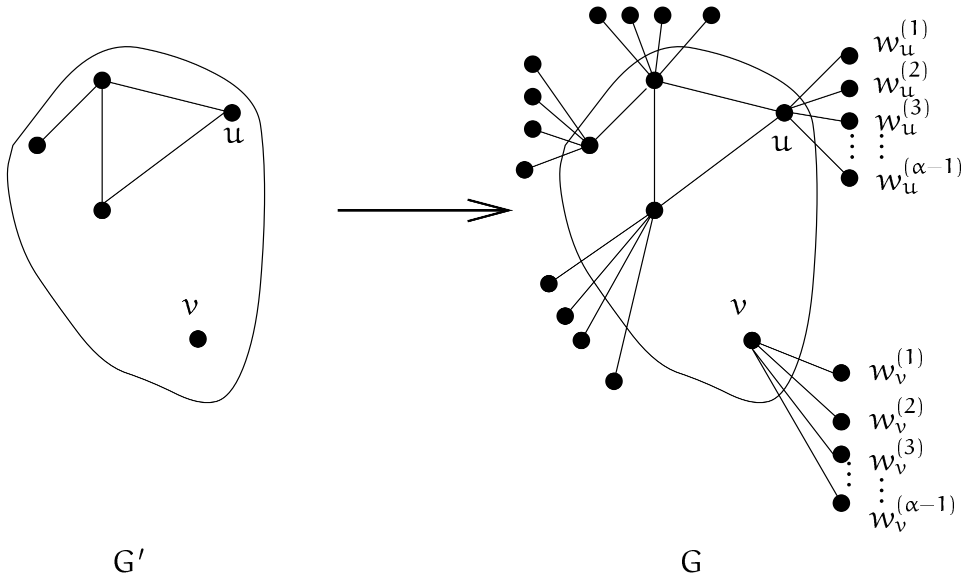


Figure 4.3: Mapping the instance of DOMINATING SET to p-CENTER$^{(\alpha)}$.

Observe that by definition of the edge-weights in G, any set P of p centers has either radius $\mathrm{rad}^{(\alpha)}(P) = 1$ or $f(|V|) + \varepsilon'$.

Observe further that in the special case when f is the constant function $f \equiv 2-\varepsilon$ for some $\varepsilon > 0$, we can choose $\varepsilon' := \varepsilon$ and obtain the distances 1 and 2 in G, as a consequence of which the triangle inequality will be satisfied by the distances defined above.

In the sequel we will show that A delivers a solution of value 1 if and only if G' has a dominating set of size s.

First, assume that G' has a dominating set D of size s. Then $P := D \cup \left(\bigcup_{v \in V} N_v\right)$ is a set of $s + (\alpha - 1)|V'| = p$ centers with solution value $\mathrm{rad}^{(\alpha)}(P) = 1$. In that case, since A has a performance guarantee of $f(|V|)$, the radius of the set of centers returned by A must also be 1.

Now suppose that G' does not have a dominating set of size s. We claim that in this case any selection of $p = s+(\alpha-1)|V|$ centers will have radius $f(|V|)+\varepsilon'$. To see this, assume that P is a set of p centers with solution value 1. First observe that P must include all the nodes from $\bigcup_{v \in V'} N_v$.

If $\alpha = 1$, this claim is trivially satisfied, because in that case $N_v = \emptyset$ for all $v \in V'$. Hence, it suffices to consider the case $\alpha \geq 2$. Any node $w \in N_v$ has v as the only neighbor in V, which is within a distance of 1 and, if not included in the set of centers, must be covered by at least α servers within a distance of 1. Hence, w must be included in the set of centers.

It follows that $P_{V'} := P \cap V' \subseteq V'$ consists of $|P| - (\alpha - 1)|V| = s$ nodes. We will now show that $P_{V'}$ is a dominating set in V'. If we take $v \in V' - P_{V'}$, then v must have at least α nodes from P within a distance of 1. Only $\alpha - 1$ from these nodes can be from N_v. Thus by definition of the distances in G', there must be $z \in P_{V'}$ such that $d(v, z) = 1$, i.e., $(v, z) \in E$.

Hence, $P_{V'}$ is a dominating set of size s in G as a contradiction to the assumption that G does not contain any dominating set of size at most s.

Thus, A delivers a solution of value 1 if and only if G' has a dominating set of size s. □

It should be noted that in the proof of the last theorem $f \in \Theta(2^{q(|V|)})$ for some polynomial q is the largest we can do in polynomial time, since otherwise the length of the binary representation for $f(|V|)$ is no longer polynomially bounded in the input size.

5

Compact Location Problems

5.1 Introduction

Several fundamental problems in location theory [HM79, MF90] involve finding a placement of facilities obeying certain "covering" constraints. Generally, the goal of such a location problem is to find a placement of minimum cost that satisfies all the specified constraints. The cost of a placement may be specified in several ways including the total cost of all the edges in the placement (i.e., the total cost of constructing the network of facilities) and the maximum cost of any edge in the placement (i.e., the *diameter*, which models the maximum communication cost between any pair of facilities). Finding a placement of sufficient generality minimizing even one of these measures is often NP-hard.

In this chapter we consider several problems dealing with the placement of a specified number of facilities on the nodes of a given network so as to minimize some function of the distances between the facilities. Because of the nature of the placement desired, we refer to these problems as *compact location problems*. For example, consider the following processor allocation problem which arises in the context of multiprocessor systems. We are given a computational task consisting of a number of communicating subtasks. At a given time, some of the processors may already be allocated and the remaining processors are available. The problem is to select a subset of processors from the currently available set of processors, one processor per subtask, such that the cost of communication among the processors executing the subtasks is minimized. In this application, the processors must be allocated quickly, and this may conflict with the goal of minimum communication cost among the selected processors.

Such location problems are commonly modeled as problems on undirected graphs. The nodes of the graph represent the available sites. A cost is associated with each pair of sites, and it is specified as the weight of the edge joining the corresponding pair of nodes. Depending on the problem, this cost represents one of several parameters such as the cost of transporting components between the pair of sites, the cost of setting up a communication link between the pair of sites, the time required to communicate between the pair of sites, and so forth. In some problems, there is also a weight associated with each node. This node weight may reflect the cost of setting up a facility at the corresponding site.

Under this graph theoretic setting, a placement is a subset of the nodes in the network of a given cardinality. The cost of a placement is a problem-specific function of the weights of the nodes and edges in the subgraph induced by the placement. Examples of such cost functions are the sum of the weights of all the edges in the placement (which may reflect the total cost of setting up communication links between each pair of chosen sites), the maximum weight of an edge in the placement (i.e., the *bottleneck cost*, which may reflect the maximum time needed to communicate between any pair of chosen sites), etc. The goal of a compact location problem is to find a placement of minimum cost.

In practice, it is often the case that a chosen placement must minimize a certain cost measure subject to budget constraints on other cost measures. Since such problems involve two cost functions, we refer to them as *bicriteria compact location problems*. As an example, consider once again the scenario presented above in the context of high performance computing. If the processors need to communicate with each other frequently to exchange data, then these data communication delays increase the time needed to complete a task. Further, the computation may require the processors to be synchronized often, thus adding to the time needed to complete the task. Therefore, it is desirable to select a subset of processors so that the total communication cost among the processors is minimized and the maximum delay due to synchronization during the computation does not exceed a given bound.

Bicriteria compact location problems arise in a number of applications such as allocation of manufacturing sites for the components of a system to minimize the cost distributing raw material and the cost of transporting components, distributing the activities of a project among geographically dispersed offices so as to minimize the transportation and communication costs among the offices, statistical clustering, pattern recognition, load-balancing in distributed systems, etc. (see [HM79, MF90, KN$^+$95a] and the references cited therein).

This chapter is organized as follows. Section 5.2 contains preliminaries and the formal definitions of the problems considered. In Section 5.3 we discuss the related research done in this area. Some basic Lemmas are presented in Section 5.4. Section 5.5 contains our approximation algorithms for the diameter constrained problems. In Section 5.6 we give the approximation algorithms for the sum constrained problems. Section 5.7 discusses some extensions of our approximation algorithms. Section 5.8 contains non-approximability results for the problems under study. In Section 5.9 we consider the restriction of the compact location problems to tree networks. We also take a brief look at geometric versions in Section 5.10.

5.2 Preliminaries and Problem Definitions

We consider a complete undirected n-vertex graph $G = (V, E)$ with one or two distance functions specified on the edges. Given an integer p, a *placement* is a subset $P \subseteq V$ of p vertices.

Let δ be a nonnegative edge-weight function on G. We use $\mathcal{D}_\delta(P)$ to denote the *diameter*, $\mathcal{S}_\delta(P)$ to denote the *sum of the distances* and $\mathcal{Q}_\delta(P)$ to denote the *sum of the squares of the distances* between the nodes in the placement P, that is

$$\mathcal{D}_\delta(P) = \max_{\substack{(v,w)\in E\\ v,w\in P}} \delta(v,w)$$

$$\mathcal{S}_\delta(P) = \sum_{\substack{(v,w)\in E\\ v,w\in P}} \delta(v,w)$$

$$\mathcal{Q}_\delta(P) = \sum_{\substack{(v,w)\in E\\ v,w\in P}} \delta^2(v,w).$$

Note that the average length and the variance of an edge in a placement P equals $\frac{2}{p(p-1)}\mathcal{S}_\delta(P)$ and $\frac{1}{p}\mathcal{Q}_\delta(P)$ respectively. Since these differ from the total length and the sum of the squared distances of all the edges in the placement, respectively, by only a scaling factor, finding a placement of minimum average length or minimum variance is equivalent to finding a placement of minimum total length and minimum total sum of the squared distances, respectively. We use this fact throughout our presentation.

Definition 5.1 (Unicriterion Compact Location Problems)
An instance of the ***minimum diameter placement problem****,* (DIA, p NODES)*, is given by an undirected complete graph* $G = (V, E)$*, a positive edge weight functions* c*, and an integer* $2 \le p \le |V|$*. The problem consists of finding a placement* P *minimizing* $\mathcal{D}_c(P)$*. Formulated with the framework given by Definition 3.5, the goal of* (DIA, p NODES) *is to find a subgraph of* G *from the set*

$$\{\, G[P] : P \subseteq V \textit{ contains } p \textit{ vertices} \},$$

minimizing the objective $\mathcal{D}_c(P)$*.*

The ***minimum average distance placement problem****,* (SUM, p NODES)*, and the* ***minimum variance placement problem****,* (VAR, p NODES)*, are defined analogously of minimizing* $\mathcal{S}_c$ *and* $\mathcal{Q}_c$ *respectively.*

It should be noted that the (VAR, p NODES) problem is less meaningful in the graph theoretic setting presented here than in a *geometric setting*, where the nodes are points in space and the distance between a pair of nodes is their Euclidean distance. Thus, we will only sketch briefly how our approximation results carry over to the minimum variance. We will discuss related work in Section 5.3 and show how to solve (SUM, p NODES) in the one-dimensional case efficiently in Section 5.10.

Definition 5.2 (Bicriteria Compact Location Problems)
An instance of the ***diameter constrained minimum diameter placement problem****,* (DIA, DIA, p NODES)*, is given by an undirected complete graph* $G = (V, E)$ *with two positive edge weight functions* c, d*, an integer* $2 \leq p \leq |V|$ *and a positive number* B*. The problem consists of finding a placement with* d*-diameter at most* B *minimizing the diameter with respect to the* c*-weights. Formulated with the framework given by Definition 3.5, the goal of* (DIA, DIA, p NODES) *is to find a subgraph of* G *from the set*

$$\{\, G[P] : P \subseteq V \textit{ contains } p \textit{ vertices and } \mathcal{D}_d(P) \leq B \,\},$$

minimizing the objective $\mathcal{D}_c(P)$*.*

For the ***diameter constrained minimum average placement problem****,* (SUM, DIA, p NODES)*, the goal is to find a placement* P *which minimizes* $\mathcal{S}_c(P)$ *subject to the constraint* $\mathcal{D}_d(P) \leq B$*.*

The ***sum constrained minimum diameter placement problem****,* (DIA, SUM, p NODES)*, and the* ***sum constrained minimum average placement problem****,* (SUM, SUM, p NODES)*, are defined analogously of finding a placement* P *minimizing* $\mathcal{D}_c(P)$ *and* $\mathcal{S}_c(P)$ *respectively subject to the budget constraint that* $\mathcal{S}_d(P) \leq B$*.*

Given one of the above problems Π, we use Δ-Π to denote the problem Π restricted to graphs in which both the edge weight functions satisfy the triangle inequality.

One of the goals of this chapter is to find good approximation algorithms for the compact location problems introduced above. We formulate our approximation algorithms for the bicriteria problems. Naturally, they can also be used for the respective unicriterion problems: Given an instance of a unicriterion compact location problem one can easily construct an instance of a bicriteria problem with the same objective function by setting the second weight function d to be identically zero. The corresponding (α, β)-approximation for the bicriteria problem will then give us an approximation algorithm for the unicriterion problem with performance α.

As mentioned above, the approximation algorithms which we will deal with are all *bicriteria approximation algorithms*. Recall that a bicriteria approximation algorithm for one of the compact location problems Π stated in Definition 5.2

with a performance of (α, β) is a polynomial-time algorithm with the following properties: If there is a feasible placement for the instance, then it produces a placement P of p nodes which is within α times the optimal value with respect to the first distance function c, violating the constraint with respect to the second distance function d by a factor of at most β. If the set of feasible solutions for the instance is empty, the algorithm has the choice of returning an "almost feasible solution", that is a solution which violates the constraint by a factor of at most β, or providing the correct information that there is no feasible placement at all.

5.3 Related Work

As mentioned earlier, problems involving the placement of p facilities have been studied extensively. In contrast, not much work has been done in finding optimal location of facilities when there is more than one constraint. A notable work in this direction is by Bar-Ilan, Kortsarz and Peleg [BIKP93] who considered the problem of assigning network centers with a bound imposed on the number of nodes that any center can service. We refer the reader to [MR+95a, RM+93] for a survey of the work done in the area of algorithms for bicriteria network design and location theory problems.

In contrast to the NP-hardness results for the (unicriterion) compact location problems on graphs studied in this chapter, geometric variants of the (DIA, p NODES) and (VAR, p NODES) problem were shown to be solvable in polynomial time in [AI+91]. In the geometric versions of these problems, the nodes are points in space and the distance between a pair of nodes is their Euclidean distance. For points in the plane, [AI+91] contains an $\mathcal{O}(p^{2.5}n \log p + n \log n)$ algorithm for the (DIA, p NODES) problem and an $\mathcal{O}(p^2 n \log n)$ algorithm for the (VAR, p NODES) problem. It is also observed in [AI+91] that these algorithms, which are based on the construction of pth order Voronoi diagrams [Lee82, PS85], extend to higher dimensions.

5.4 Some Basic Lemmas

Observe that for a given placement $P \subseteq V$ of p nodes we need $\Theta(p^2)$ time to calculate $\mathcal{S}_c(P)$, $\mathcal{D}_c(P)$ and $\mathcal{Q}_c(P)$. In this section, we are going to establish lemmas which will enable us to use related measures that can be computed faster.

Lemma 5.3 *Suppose that the edge weight function* c *obeys the triangle inequality. Let* $v \in V$ *be an arbitrary node and let* $N \subseteq V \setminus \{v\}$ *be a set of* $p - 1$

nodes. Define

$$\hat{\mathcal{S}}_c(v, N) := \sum_{w \in N} c(v, w). \tag{5.1}$$

Then the inequality

$$\mathcal{S}_c(N \cup \{v\}) \leq (p-1) \cdot \hat{\mathcal{S}}_c(v, N)$$

holds true.

Proof: Let $w \in N$ be arbitrary. Then

$$\begin{aligned}
\sum_{u \in N \cup \{v\} \setminus \{w\}} c(w, u) &= c(w, v) + \sum_{u \in N \setminus \{w\}} c(w, u) \\
&\leq c(w, v) + \sum_{u \in N \setminus \{w\}} (c(w, v) + c(v, u)) \\
&= (p-1)c(w, v) + \sum_{u \in N \setminus \{w\}} c(v, u) \\
&= (p-2)c(v, w) + \sum_{u \in N} c(v, u) \\
&= (p-2)c(v, w) + \hat{\mathcal{S}}_c(v, N).
\end{aligned} \tag{5.2}$$

Now using (5.2) we obtain

$$\begin{aligned}
\mathcal{S}_c(N \cup \{v\}) &= \frac{1}{2}\left(\sum_{u \in N} c(v, u) + \sum_{w \in N} \sum_{u \in N \cup \{v\} \setminus \{w\}} c(w, u)\right) \\
&= \frac{1}{2}\hat{\mathcal{S}}_c(v, N) + \frac{1}{2}\sum_{w \in N} \sum_{u \in N \cup \{v\} \setminus \{w\}} c(w, u) \\
&\overset{(5.2)}{\leq} \frac{1}{2}\hat{\mathcal{S}}_c(v, N) + \frac{1}{2}\sum_{w \in N} \left((p-2)c(v, w) + \hat{\mathcal{S}}_c(v, N)\right) \\
&= \frac{1}{2}\hat{\mathcal{S}}_c(v, N) + \frac{p-2}{2}\hat{\mathcal{S}}_c(v, N) + \frac{p-1}{2}\hat{\mathcal{S}}_c(v, N) \\
&= (p-1)\hat{\mathcal{S}}_c(v, N).
\end{aligned}$$

This proves the lemma. □

Lemma 5.4 *Suppose that the edge weight function* c *obeys the triangle inequality. Let* $v \in V$ *be an arbitrary node and let* $N \subseteq V \setminus \{v\}$ *be a set of* $p-1$ *nodes. Define*

$$\hat{\mathcal{D}}_c(v, N) := \max_{w \in N} c(v, w). \tag{5.3}$$

Then

$$\mathcal{D}_c(N \cup \{v\}) \le 2 \cdot \hat{\mathcal{D}}_c(v, N).$$

Proof: Observe that $N \cup \{v\}$ will form a clique in the square of the bottleneck graph $\mathrm{Bottleneck}(G, c, \hat{\mathcal{D}}_c(v, N))$. The claim now follows immediately from Observation 3.18. □

Lemma 5.5 *If the distances in a network satisfy the triangle inequality, then for all nodes* u, v, w*, the following inequality holds true:*

$$[c(u, w)]^2 \le 2\left([c(u, v)]^2 + [c(v, w)]^2\right).$$

Proof: By the triangle inequality, we have $c(u, w) \le c(u, v) + c(v, w)$. By squaring both sides we obtain

$$[c(u, w)]^2 \le [c(u, v)]^2 + [c(v, w)]^2 + 2c(u, v)c(v, w). \tag{5.4}$$

Moreover, we obtain from

$$0 \le (c(u, v) - c(v, w))^2 = [c(u, v)]^2 + [c(v, w)]^2 - 2c(u, v)c(v, w)$$

that

$$2c(u, v)c(v, w) \le [c(u, v)]^2 + [c(v, w)]^2. \tag{5.5}$$

Thus by (5.4) and (5.5)

$$[c(u, w)]^2 \le [c(u, v)]^2 + [c(v, w)]^2 + 2c(u, v)c(v, w) \le 2\left([c(u, v)]^2 + [c(v, w)]^2\right)$$

which proves the lemma. □

Using the results of Lemma 5.5 one can now establish the following result analogously to the proof of Lemma 5.3.

Lemma 5.6 *Suppose that the edge weight function* c *obeys the triangle inequality. Let* $v \in V$ *be an arbitrary node and let* $N \subseteq V \setminus \{v\}$ *a set of* $p - 1$ *nodes. Define*

$$\hat{\mathcal{Q}}_c(v, N) := \sum_{w \in N} [c(v, w)]^2. \tag{5.6}$$

Then the inequality

$$\mathcal{Q}_c(N \cup \{v\}) \le (2p - 3) \cdot \hat{\mathcal{Q}}_c(v, N)$$

holds true. □

5.5 Algorithms for Diameter Constrained Problems

Basic Approximation Technique

We first discuss the basic technique used in approximating the diameter constrained compact location problems. The approach uses the ideas from Hochbaum and Shmoys [HS86] which we have introduced in Section 3.5. We combine this technique with a local search heuristic.

Let us first consider the problem Δ-(SUM, DIA, p NODES). Let B be the bound on diameter of the placement with respect to the d-cost. It is clear that in the subgraph induced by an optimal placement no edge has d-cost more than B. Thus we can prune all the edges in the graph with d-costs more than B. Call the resulting graph G'. We then find a near optimal placement in G' with respect to the c-costs. This placement will turn into a clique in the square graph $(G')^2$. Because the edge weights satisfy the triangle inequality, it will follow that the placement does not violate the constraint by a factor of more than 2.

Although the basic idea is simple, the method yields near tight bounds on the performance guarantees of the resulting algorithms as the hardness results in Section 5.8 will show.

We start presenting an approximation algorithm for the problem Δ-(SUM, DIA, p NODES) that provides a performance guarantee of $(2-2/p, 2)$. Algorithm 5.1 displays the approximation algorithm where the cost measure $\mathcal{M}_c$ is set to $\hat{\mathcal{S}}_c$ which is defined in Lemma 5.3.

Performance Guarantee for Δ-(SUM, DIA, p NODES)

First we will show that a placement returned by the algorithm will be almost feasible in the sense that it violates the diameter constraint with respect to the d-weights by a factor of at most 2.

Lemma 5.7 *Any placement which is considered by Algorithm 5.1 in Step 8 has d-diameter at most $2B$. If Algorithm 5.1 returns a vertex set P and not the information that no feasible solution exists, then the d-diameter of P is at most $2B$.*

Proof: Observe that any placement $P(v)$ which is considered by the algorithm consists of a vertex v and $p-1$ vertices $w_1, \ldots, w_{p-1}$ that are adjacent to v in the bottleneck graph $G' := \text{Bottleneck}(G, d, B)$, which does not contain edges of d-weight greater than B. The placement will form a clique in $(G')^2$. By Observation 3.18 all the edges between the vertices in this clique have d-weight at most $2B$. Thus, the d-diameter of $P(v)$ is at most $2B$. The lemma now follows. □

Algorithm 5.1 Approximation algorithm for Δ-(SUM, DIA, p NODES).

Input: A complete graph $G = (V, E)$ with two edge weight functions c, d, an integer p and a number B, a cost measure $\mathcal{M}_c$

1 $G' \leftarrow \text{Bottleneck}(G, d, B)$
2 $V_{\text{cand}} := \{ v \in G' : \deg_{G'}(v) \geq p - 1 \}$
3 **if** $V_{\text{cand}} = \varnothing$ **then**
4   **return** "There is no feasible solution"
5 **end if**
6 **for all** $v \in V_{\text{cand}}$ **do**
7   {assume that $N_{G'}(v) = \{w_1, \ldots, w_r\}$ with $c(v, w_1) \leq \cdots \leq c(v, w_r)$}
    $N(v) \leftarrow \{w_1, \ldots, w_{p-1}\}$
8    $P(v) \leftarrow \{v\} \cup N(v)$
9 **end for**
10 **return** the placement $P(v) = \{v\} \cup N(v)$ with smallest value $\mathcal{M}_c(v, N(v))$

We are now ready to establish the performance of Algorithm 5.1.

Lemma 5.8 *Algorithm 5.1 with* $\mathcal{M}_c := \hat{\mathcal{S}}_c$ *has a performance of* $(2 - 2/p, 2)$.

Proof: By Lemma 5.7 we know that any solution output by the algorithm will violate the constraint on the d-diameter by a factor of at most 2. Assume that the algorithm informs about the infeasibility of an instance in Step 4. Then, indeed, no feasible solution to the instance can exist, since any feasible solution with d-diameter at most B will form a clique in the bottleneck graph G'. In particular there will be at least p nodes of degree p or greater in G' and V_{cand} can not be empty.

It remains to show that for any instance of Δ-(SUM, DIA, p NODES) with nonempty set of feasible solutions the algorithm will find a good placement with respect to the objective $\mathcal{S}_c$.

Consider an optimal solution P^* such that $\mathcal{D}_d(P^*) \leq B$ and let OPT $:= \mathcal{S}_c(P^*)$ be the optimal objective function value. By definition, this placement forms a clique of size p in $G' := \text{Bottleneck}(G, d, B)$. Hence, for any node $v \in P^*$ we have $|N_{G'}(v)| \geq p$ and $P^* \subseteq V_{\text{cand}}$.

Define for each node $v \in P^*$:

$$R_v := \sum_{\substack{w \in P^* \\ w \neq v}} c(v, w).$$

We have $\mathcal{S}_c(P^*) = \frac{1}{2}\sum_{v \in P^*} R_v$. Now let $v^* \in P^*$ be so that R_{v^*} is a minimum among all nodes in P^*. Then clearly

$$\mathrm{OPT} = \mathcal{S}_c(P^*) \geq \frac{p}{2} R_{v^*}. \tag{5.7}$$

As mentioned earlier, $v^* \in V_{\mathrm{cand}}$. Consider the step of Algorithm 5.1 in which it examines v^*. Let $N(v^*) := P(v^*) \setminus \{v^*\}$ denote the set of $p-1$ nearest neighbors of v^* in G' with respect to c. Then we have

$$\hat{\mathcal{S}}_c(v^*, N(v^*)) = \sum_{w \in N(v^*)} c(v^*, w) \leq R_{v^*}, \tag{5.8}$$

by definition of $N(v^*)$ as the set of nearest neighbors. By construction, Algorithm 5.1 chooses a placement $P(v) = N(v) \cup \{v\}$ such that $\hat{\mathcal{S}}_c(v, N(v))$ is minimized. Consequently, we have $\hat{\mathcal{S}}_c(v, N(v)) \leq \hat{\mathcal{S}}_c(v^*, N(v^*))$ and obtain that

$$\hat{\mathcal{S}}_c(v, N(v)) \leq \hat{\mathcal{S}}_c(v^*, N(v^*)) \overset{(5.8)}{\leq} R_{v^*} \overset{(5.7)}{\leq} \frac{2}{p} \cdot \mathrm{OPT}. \tag{5.9}$$

By Lemma 5.3 we have

$$\mathcal{S}_c(P(v)) \leq (p-1) \cdot \hat{\mathcal{S}}_c(v, N(v)) \overset{(5.9)}{\leq} \left(2 - \frac{2}{p}\right) \cdot \mathrm{OPT}.$$

This establishes the claimed performance guarantee. □

Running Time

We now address the running time of Algorithm 5.1. Let $n := |V|$ be the number of nodes in the graph G. Then the number m of edges is $m = \frac{n(n-1)}{2} \in \Theta(n^2)$. Thus, the bottleneck graph G' in Step 1 of the algorithm can be computed in time $\mathcal{O}(m)$ by simply inspecting the weight $d(e)$ for each edge $e \in E$. The set of candidate nodes V_{cand} can then be determined in $\mathcal{O}(n)$ time. The main effort of the algorithm occurs during the loop in Steps 6 to 9.

For each of the nodes in V_{cand} we must determine the nearest neighbors with respect to the c distance function. This can be done in $\mathcal{O}(n+p)$ time as follows. We first use a linear time selection algorithm (see e.g. [CLR90]) to find the node w with the $(p-1)$st smallest distance from v. Then, by performing one linear pass over the $\mathcal{O}(n)$ neighbors of v and comparing their distance to $c(v, w)$, we can then extract the $p-1$ nearest neighbors of v in $\mathcal{O}(n+p)$ time.

Then, the algorithm computes the sum $\hat{\mathcal{S}}_c(v, N(v))$ for each of the $\mathcal{O}(n)$ placements $P(v) = N(v) \cup \{v\}$. Evaluating $\hat{\mathcal{S}}_c$ for one placement needs $\mathcal{O}(p)$ time.

Choosing the best placement $P(v)$ (with respect to $\hat{\mathcal{S}}_c$) can then be accomplished in $\mathcal{O}(n)$ time. Thus, the total time complexity of Algorithm 5.1 is $\mathcal{O}(n^2)$ which is polynomial in the input size. Hence, we have shown the following theorem:

Theorem 5.9 *Algorithm 5.1 with $\mathcal{M}_c := \hat{\mathcal{S}}_c$ is an approximation algorithm for* Δ-(SUM, DIA, p NODES) *with a performance of* $(2-2/p, 2)$ *and a running time of* $\mathcal{O}(n^2)$. □

Along the same lines as above but using the results of Lemmas 5.5 and 5.6 one can establish the analogous result for minimizing $\mathcal{Q}_c$ subject to the diameter constraint.

Theorem 5.10 *Algorithm 5.1 with $\mathcal{M}_c := \hat{\mathcal{Q}}_c$ is an approximation algorithm for* Δ-(VAR, DIA, p NODES) *with a performance of* $(4-6/p, 2)$ *and a running time of* $\mathcal{O}(n^2)$. □

Worst Case Example Algorithm 5.1

We provide an example of an instance of Δ-(SUM, DIA, p NODES) where the $(2-2/p, 2)$-performance of our heuristic is asymptotically tight. In this example, the node set V consists of $p+1$ subsets $S_0, \ldots, S_p$. The set $S_0 = \{v_1, \ldots, v_p\}$ forms an optimal placement of p nodes and the distances between the nodes in S_0 are $c(v_i, v_j) = 1$ and $d(v_i, v_j) = 1$. All other sets S_i $(i \geq 1)$ consist of $p-1$ nodes and we have that $c(u, w) = 2-2\varepsilon$ and $d(u, w) = 2$ $(u, w \in S_i)$.

Let v_i be any node in the optimum placement S_0. We set $c(v_i, w) = 1-\varepsilon$ and $d(v_i, w) = 1$ for all $w \in S_i$, while we let $c(v_i, w') = 2-\varepsilon$ and $d(v_i, w) = 2$ for all $w' \in S_j, j \neq i$. Finally, for each edge (u, v) such that u and v belong to distinct sets S_i, we set $c(u, v) = 3-2\varepsilon$ and $d(u, v) = 2$. It is easily verified that the distances c and d obey the triangle inequality. The instance just described is illustrated in Figure 5.1. The edges are labeled by their c-weights.

For the bound $B := 1$, the set S_0 forms an optimum placement with objective function value $\mathcal{S}_c(S_0) = \frac{p(p-1)}{2}$ and diameter $\mathcal{D}_d(S_0) = 1$. In the first step, the algorithm calculates the bottleneck graph $G' = \text{Bottleneck}(G, d, 1)$. This graph now only contains the edges between the nodes in S_0 and the edges of the form (v_i, w) with $w \in S_i$. It is now easy to see that $V_{cand} = S_0$.

For each node $v_i \in S_0$ the set of $p-1$ nearest neighbors with respect to the c-weights consists of the set S_i. Since $\mathcal{D}_d(\{v_i\} \cup S_i) = 2 = 2B$, *any* placement considered by the algorithm will violate the diameter constraint by a factor of 2. Moreover

$$\mathcal{S}_c(\{v_i\} \cup S_i) = (p-1)(1-\varepsilon) + \frac{(p-1)(p-2)}{2}(2-2\varepsilon).$$

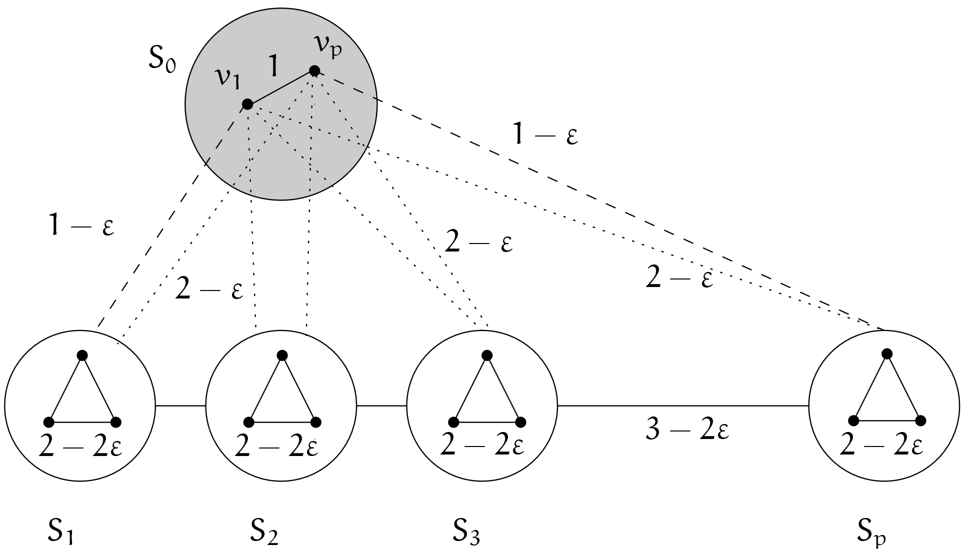


Figure 5.1: Worst case example for Algorithm 5.1.

Thus, $\lim_{\varepsilon \to 0} \frac{\mathcal{S}_c(\{v_i\} \cup S_i)}{\mathcal{S}_c(S_0)} = 2 - 2/p$.

Approximating Δ-(DIA, DIA, p NODES)

Our approximation algorithm for Δ-(DIA, DIA, p NODES) is the same as Algorithm 5.1, except that the measure $\mathcal{M}_c$ is set to $\mathcal{M}_c := \hat{\mathcal{D}}_c$, where $\hat{\mathcal{D}}_c$ is defined in (5.3). The algorithm obtained this way has a performance of $(2, 2)$, which will be proved in the sequel. The hardness results (Theorem 5.21) will show that this is the best approximation we can expect to obtain in polynomial time.

Theorem 5.11 *Algorithm 5.1 (with $\mathcal{M}_c := \hat{\mathcal{D}}_c$) is an approximation algorithm for* Δ-(DIA, DIA, p NODES) *with a performance of* $(2, 2)$. *It can be implemented to run in* $\mathcal{O}(n^2)$ *time.*

Proof: Consider an optimal solution P^* such that $\mathcal{D}_d(P^*) \leq B$. It can be seen that $P^* \subseteq V_{cand}$, and thus V_{cand} is non empty. Consequently, the heuristic will not output "There is no feasible solution" in Step 4. Any placement considered by Algorithm 5.1 in Step 8 will turn into a clique in $(G')^2$, where $G' = \text{Bottleneck}(G, d, B)$. Thus as in the proof of Lemma 5.8, we conclude again that any placement considered by the heuristic will violate the diameter constraint by a factor of at most 2.

Now consider an arbitrary node $v \in P^*$. Clearly, $v \in V_{cand}$ and consequently it will be considered in the loop from Step 6 to Step 9. By definition of $N(v)$, for any $w \in N(v)$ we have $c(v, w) \leq \mathrm{OPT}$, hence $\hat{\mathcal{D}}_c(v, N(v)) \leq \mathrm{OPT}$. Thus by the triangle inequality

$$\mathcal{D}_c(N(v) \cup \{v\}) \leq 2 \cdot \hat{\mathcal{D}}_c(v, N(v)) \leq 2 \cdot \mathrm{OPT}.$$

The same running time analysis as in the case of Δ-(SUM, DIA, p NODES) (see page 64) shows that the algorithm runs in time $\mathcal{O}(n^2)$. □

The lower bound example in Figure 5.1 can be used to show that the approximation ratio indicated in Theorem 5.11 is tight for the heuristic. Again, the optimum placement consists of the nodes in S_0 which has both c- and d-diameter 1. The placement returned by the heuristic is one of the sets $\{v_i\} \cup S_i$, which has c-diameter $2 - 2\varepsilon$ and d-diameter 2. For $\varepsilon > 0$ small enough the performance of the algorithm for the instance is as close to $(2, 2)$ as one likes.

5.6 Algorithms for Sum Constrained Problems

In this section, we study approximation algorithms for bicriteria compact location problems where the objective is to minimize either the diameter $\mathcal{D}_c$ or the sum of the distances $\mathcal{S}_c$ subject to constraints of sum type.

Approximating Δ-(DIA, SUM, p NODES)

We proceed to present a heuristic for Δ-(DIA, SUM, p NODES). This algorithm uses our approximation algorithm for Δ-(SUM, DIA, p NODES) (Algorithm 5.1) from Section 5.5 as a relaxed decision procedure to search for the optimum diameter with respect to c. This will involve interchanging the c- and d-weights on the edges. For simplicity, we will use A to denote the approximation algorithm for Δ-(SUM, DIA, p NODES) where we are given a bound on the c-diameter and want to minimize the sum $\mathcal{S}_d$ of the d-distances (in the original formulation in Definition 5.2 the roles of c and d were interchanged). Thus, given a bound M on the c-diameter A will either provide the information that no placement of c-diameter M does exist or output a placement P such that $\mathcal{D}_c(P) \leq 2M$ and $\mathcal{S}_d(P)$ is at most $2 - 2/p$ times the minimum value subject to the constraint on the c-diameter.

It is straightforward to see that A can be viewed as a 2-relaxed procedure where we interpret the outcome of A to be a COF if A does not output a placement P such that $\mathcal{S}_d(P) \leq (2 - 2/p)B$.

We can sort the set $\{c(e) : e \in E\}$ in $\mathcal{O}(m \log m)$ time, where $m := |E| = \frac{n(n-1)}{2}$, and perform a binary search to find the minimum value such that the outcome of A is not a COF by the above conventions. This will give us an approximation for Δ-(DIA, SUM, p NODES) with performance $(2, 2 - 2/p)$ by issuing $\mathcal{O}(\log m) \subseteq \mathcal{O}(\log n)$ calls to A.

Theorem 5.12 *There exists an $\mathcal{O}(n^2 \log n)$ time approximation algorithm for* Δ-(DIA, SUM, p NODES) *with performance of* $(2, 2 - 2/p)$. □

Again, a similar theorem can be established for the case when the constraint is with respect to the sum of the squared distances $\mathcal{Q}_d$ rather than with respect to $\mathcal{S}_d$. The performance becomes then $(2, 4 - 6/p)$.

As a lower bound example take again the instance depicted in Figure 5.1 but interchange the distances c and d and set the bound B to be $\frac{p(p-1)}{2}$. It is easy to see that by choosing $\varepsilon > 0$ small enough, the performance for this instance is arbitrarily close to $(2, 2 - 2/p)$.

Approximating Δ-(SUM, SUM, p NODES)

We now present a heuristic for Δ-(SUM, SUM, p NODES). The basic idea behind the approximation algorithm is to use a *parametric search* technique to reduce the problem to that of solving the minimum sum problem for *one* modified weight function. Then, by appropriate scaling and rounding techniques, this solution can be transformed back into a near optimal solution for the original bicriteria problem.

The presentation of our approximation algorithm is organized as follows. We first present our heuristic and show that it provides a performance guarantee of $((2 - 2/p)(1 + 1/\varepsilon), (2 - 2/p)(1 + \varepsilon))$ for any fixed $\varepsilon > 0$. We then show how to improve the running time of the heuristic using an elegant technique due to Megiddo [Meg83].

Performance Guarantee

The main procedure shown in Algorithm 5.2 uses the test procedure from Algorithm 5.3. Step 4 of the test procedure, that is, computing a $(2 - 2/p)$-approximation for the (unicriterion) (SUM, p NODES)-instance with distances given by h, can be done by using our approximation algorithm for Δ-(SUM, DIA, p NODES) (Algorithm 5.1), where we set $c := h$, $d :\equiv 0$ and $B := 0$. We also note that ε is a fixed quantity that specifies the accuracy requirement.

For a value of K let OPT_{h_K} denote the sum of the distances of an optimal placement of p nodes with respect to the edge weight function $h_K(v, w) :=$

Algorithm 5.2 Approximation Algorithm for Δ-(SUM, SUM, p NODES).

Input: A complete graph $G = (V, E)$ with two edge weight functions c, d, an integer p, and a number B;
a *fixed* accuracy parameter $\varepsilon > 0$

Use a binary search to find the smallest integer

$$K \in \left[0, \frac{p(p-1)}{2} \cdot \max\{\, c(e) : e \in E \,\}\right]$$

such that Algorithm 5.3 called with the test parameter K does not return a COF.
if the search terminates with the information that there is no such integer **then**
  return "There is no feasible solution"
end if
Let P be the placement generated by Algorithm 5.3.
if $\mathcal{S}_d(P) > (2 - 2/p)B$ **then**
  return "There is no feasible solution"
else
  return P
end if

Algorithm 5.3 Test procedure for Δ-(SUM, SUM, p NODES).

Input: A complete graph $G = (V, E)$ with two edge weight functions c, d, an integer p, a number B, and a test parameter K;
a *fixed* accuracy parameter $\varepsilon > 0$

for all pairs (v, w) of nodes **do**
  $h(v, w) \leftarrow c(v, w) + \frac{K}{B} d(v, w)$
end for
Compute a $(2-2/p)$-approximation P for the instance of (SUM, p NODES) given by the graph G, distances $h(e)$, $e \in E$ and the number p.
if $\mathcal{S}_h(P) \leq (1 + \varepsilon)(2 - 2/p)K$ **then**
  return P
else
  return COF
end if

$c(v, w) + \frac{K}{B}d(v, w)$, that is

$$\mathrm{OPT}_{h_K} = \min_{\substack{P \subseteq V \\ |P|=p}} \mathcal{S}_{h_K}(P).$$

Then we have the following lemma:

Lemma 5.13 *The function* $F(K) = \frac{\mathrm{OPT}_{h_K}}{K}$ *is monotonically nonincreasing on* $\mathbb{R}_+ \setminus \{0\}$.

Proof: Let K_1 and K_2 be given numbers with $K_1 < K_2$. Let P_1 and P_2 denote optimal placements of p nodes under h_{K_1} and h_{K_2}, respectively. For $i \in \{1, 2\}$, let C_i and D_i denote the costs of placement P_i under c and d respectively. Thus, we have that $F(K_i) = \frac{C_i}{K_i} + \frac{D_i}{B}$ for $i \in \{1, 2\}$.

Consider the cost of the placement P_1 under h_{K_2}. By the definition of C_1 and D_1, it follows that the cost of P_1 is $C_1 + \frac{D_1 \cdot K_2}{B}$. Thus the value of $F(K_2)$ is at most this cost divided by K_2 which is $\frac{C_1}{K_2} + \frac{D_1}{B}$. This in turn is less than $\frac{C_1}{K_1} + \frac{D_1}{B}$, since $K_1 < K_2$. But $\frac{C_1}{K_1} + \frac{D_1}{B}$ is exactly $F(K_1)$, and hence $F(K_1) \geq F(K_2)$. □

The next corollary proves that the binary search in Algorithm 5.2 can work correctly. Before we state and prove the corollary, observe that Step 7 of Algorithm 5.2 ensures that, if the algorithm outputs a placement, this placement will violate the constraint on the sum of the d-distances by a factor of at most $2 - 2/p$. Thus, in the sequel we can restrict ourselves to instances with nonempty set of feasible solutions. Given such an instance, let $\mathrm{OPT} = \mathcal{S}_c(P^*)$ denote the function value of an optimal placement P^* of p nodes. To simplify the analysis, we assume that OPT/ε is an integer. This can be enforced by first scaling the cost function c so that all values are integers and then scaling again by ε.

Corollary 5.14 *The test procedure Algorithm 5.3 returns a placement for all* $K > \mathrm{OPT}/\varepsilon$. *Thus, the binary search in Algorithm 5.2 works correctly and either finds a value* $K^* \leq \mathrm{OPT}/\varepsilon$ *or provides the information that Algorithm 5.3 returns COF for all values of* K.

Proof: We first show that the procedure will return a placement (and not a COF) if called with $K' = \mathrm{OPT}/\varepsilon$. Notice that K' is an integer by our assumption. We estimate the sum of the $h_{K'}$-distances between the nodes in the optimal placement P^*. This sum is then $\mathrm{OPT} + \frac{K'}{B}B = \mathrm{OPT} + K' = (1 + \varepsilon)K'$. Thus it follows that $\mathrm{OPT}_{h_{K'}} \leq (1 + \varepsilon)K'$ and the $(2 - 2/p)$-approximation P that is computed in Step 4 will satisfy

$$\mathcal{S}_{h_{K'}}(P) \leq (2 - 2/p)\mathrm{OPT}_{h_{K'}} \leq (1 + \varepsilon)(2 - 2/p)K'.$$

Thus, we observe that the procedure will not return a COF. Moreover, since $\mathrm{OPT}_{h_{K'}} \leq (1+\varepsilon)K'$, it follows that $F(K') \leq (1+\varepsilon)$, where F is the function defined in Lemma 5.13. By the results of this lemma we then have $F(K) \leq (1+\varepsilon)$ for all $K \geq K'$.

This is equivalent to saying that for all $K \geq K'$ the corresponding optimal placement P^*_K minimizing $\mathcal{S}_{h_K}$ satisfies $\mathcal{S}_{h_K}(P^*_K) \leq (1+\varepsilon)K$. Hence for all these values of K, the approximation P computed in Step 4 of the test procedure will satisfy $\mathcal{S}_{h_K}(P) \leq (2-2/p)(1+\varepsilon)K$. But this means that Algorithm 5.3 will not return COF for any $K \geq K'$. □

Now we are ready to complete the proof of the performance of our approximation algorithm.

Lemma 5.15 *For any fixed* $\varepsilon > 0$ *Algorithm 5.2, as applied to* Δ-(SUM, SUM, p NODES)*, has a performance of* $((2-2/p)(1+1/\varepsilon), (2-2/p)(1+\varepsilon))$.

Proof: By Corollary 5.14, the binary search in Algorithm 5.2 will successfully end with a value of K^* satisfying $K^* \leq \mathrm{OPT}/\varepsilon$. Let P be the corresponding placement that is returned by Algorithm 5.3. Then we have

$$\mathcal{S}_c(P) \leq \mathcal{S}_h(P) \leq \left(2-\frac{2}{p}\right)\cdot\left(\mathrm{OPT}+\frac{K^*}{B}B\right) \leq \left(2-\frac{2}{p}\right)\left(1+\frac{1}{\varepsilon}\right)\cdot\mathrm{OPT}.$$

Moreover, we see that

$$\frac{K^*}{B}\mathcal{S}_d(P) \leq \mathcal{S}_h(P) \leq \left(2-\frac{2}{p}\right)(1+\varepsilon)\,K^*.$$

Multiplying the last chain of inequalities by B/K^* yields

$$\mathcal{S}_d(P) \leq (2-2/p)(1+\varepsilon)B.$$

This completes the proof. □

In the above version of the heuristic for Δ-(SUM, SUM, p NODES) the test procedure in Algorithm 5.3 is called $\mathcal{O}(\log(p^2 c_{\max}/\varepsilon))$ times during the binary search, where $c_{\max} := \max\{c(e) : e \in E\}$. For the rest of this section, let $T_{\text{test}}(n) \in \mathcal{O}(n^2)$ the time required for a single call to Algorithm 5.3. Then, the total time for the algorithm would be $\mathcal{O}(\log(p^2 c_{\max}/\varepsilon)\cdot T_{\text{test}}(n))$. We will now show how to improve this running time.

Outline of a Faster Heuristic

If Algorithm 5.3 is called with some parameter K, it first computes the compound weights h_K. Then, in Step 4 it computes a $(2-2/p)$-approximation for the (unicriterion) problem of finding a set of p nodes minimizing $\mathcal{S}_{h_K}$. As mentioned above this is done with the help of our Algorithm 5.1 from Section 5.5.

This algorithm will generate n placements $P(v)$, one for each vertex and its nearest neighbors with respect to h_K. It then outputs the placement with the best objective function value $\hat{\mathcal{S}}_{h_K}$.

As before, let $K^* \in [0, p^2 \max\{c(e) : e \in E\}]$ be the minimum value such that the test procedure Algorithm 5.3 returns a placement and not a COF. Assume that we already know the ordering of the edges in the graph with respect to h_{K^*}. Then, for each vertex v we can find the $p-1$ nearest neighbors with respect to h_{K^*}. We do *not* need to know the weights; the ordering suffices. Thus, given the knowledge about the ordering with respect to h_{K^*}, we can find a set of n placements $P(v)$ containing the placement output by our slow heuristic for Δ-(SUM, SUM, p NODES) above: for each vertex $v \in V$ the placement $P(v)$ consists again of v and its $p-1$ nearest neighbors with respect to h_{K^*}.

This is what our faster algorithm will do in the first phase. It will find the ordering with respect to h_{K^*} and narrow our search to n placements. This is done without actually knowing K^*. In the second phase, we will determine the placement among these n placements which our slow heuristic would output.

Finding an Ordering with Respect to h_{K^*}

Let $m = \frac{n(n-1)}{2}$ be the number of edges in the graph $G = (V, E)$. Basically, we wish to sort the set $S := \{h_{K^*}(e_1), \dots, h_{K^*}(e_m)\}$ where K^* is not known. However, for any K we can decide whether $K^* \leq K$ or $K^* > K$ by one call to our test procedure Algorithm 5.3: If it returns a COF, then we know that $K^* > K$. Otherwise, we can conclude that $K^* \leq K$.

Recall that the h_K-weight of an edge e is given by $c(e) + \frac{K}{B}d(e)$. Thus, for each edge e, the compound weight $h_K(e)$ viewed as a function of K is a linear function. Given two edges e and e', their ordering with respect to the compound weights h_K changes at most once when K varies, namely at the point where the two linear functions intersect. Clearly, given two edges e and e' this value of K can be computed in constant time.

To simplify the presentation, we will first sketch the main idea before going into details. Imagine applying a (sequential) sorting algorithm to S. The sorting algorithm would start by comparing some values $h_{K^*}(e)$ and $h_{K^*}(e')$. Then, we could do the following:

We compute the intersection point of the two linear functions $h_K(e)$ and $h_K(e')$. If they do not intersect, the ordering of e and e' will not change if K varies. Hence, inspecting $h_K(e)$ and $h_K(e')$ for *any* value of K will tell us whether

$h_{K^*}(e) \geq h_{K^*}(e')$ or vice versa. Otherwise, let $K_{e,e'}$ be the value of K where $h_K(e)$ and $h_K(e')$ intersect. We call our procedure Algorithm 5.3 for $K = K_{e,e'}$ and find out whether $K^* \leq K_{e,e'}$ or $K^* > K_{e,e'}$. Since the ordering of the edges e and e' only changes at $K_{e,e'}$, we can then decide whether $h_{K^*}(e) \geq h_{K^*}(e')$ or vice versa. Thus, by $\mathcal{O}(1)$ calls to Algorithm 5.3, we can determine the result of a comparison.

Using the idea from above in conjunction with a standard sequential sorting algorithm (which makes $\mathcal{O}(m \log m)$ comparisons), we could find the ordering of the edges at K^* by $\mathcal{O}(m \log m)$ calls to Algorithm 5.3. However, using Megiddo's technique [Meg83] we can speed up the algorithm substantially.

The idea is to use an adaptation of a sequentialized parallel sorting algorithm such as Cole's scheme [Col88]. Recall that a comparison essentially consists of a call to Algorithm 5.3 which takes $T_{\text{test}} \in \mathcal{O}(n^2)$ time, so comparisons are expensive. Using the parallel sorting scheme, we basically accept a greater total number of comparisons, but we can use the parallelism to group the independent comparisons made in one stage of the parallel machine and then answer all of them together efficiently.

Cole's algorithm uses m processors to sort an array of m elements in parallel time $\mathcal{O}(\log m)$. Recall that in our case $m = |E|$ is the number of edges in the graph $G = (V, E)$. The algorithm is simulated serially, employing one "processor" at a time, according to some fixed permutation, letting each perform one step in each cycle. When two values $h_{K^*}(e)$ and $h_{K^*}(e')$ have to be compared, we compute the intersection point of the two linear functions, where the ordering changes (but we do *not* know the result of the comparison at this point). The crucial observation is that these critical values can be computed independently, meaning that each of the "processors" does not need any knowledge about the critical points computed by the other ones.

After the first of the $\mathcal{O}(\log m)$ stages, we are given at most m critical values of K, say $K_1 \leq K_2 \leq \cdots \leq K_r$ with $r \leq m$. For convenience set $K_0 := -\infty$ and $K_{r+1} := +\infty$. Using binary search, we find an interval $[K_i, K_{i+1}]$, where K^* must be contained.

This is done in the following way: Start with *low* $:= -\infty$ and *high* $:= +\infty$. Then compute the median $M := K_{\lfloor (r+1)/2 \rfloor}$ of the K_j in $\mathcal{O}(r)$ time. We then decide whether $K^* \leq M$ by one call to Algorithm 5.3. If it returns COF, then we know that $K^* > M$. Otherwise, $K^* \leq M$. In the first case, we set *low* $:= M$ and remove all values K_j with $K_j < M$ from our set of critical values. Similarly, in the second case we set *high* $:= M$ and remove the values greater than the median M. Clearly, this can be done in $\mathcal{O}(r)$ time. Since M was the median of the K_j the number of critical values decreases by a factor of one half.

Then, the total time effort $T_{total}(r)$ for the binary search satisfies the recurrence:

$$T_{total}(r) = T_{total}(r/2) + T_{test}(n) + \mathcal{O}(r),$$

where $T_{test}(n) \in \mathcal{O}(n^2)$ is the time needed for one call to Algorithm 5.3. The solution of the recurrence is $T_{total}(r) \in \mathcal{O}(r + T_{test}(n) \log r)$. Since $r \in \mathcal{O}(m)$, this shows that we obtain the interval $[K_i, K_{i+1}]$ containing K^* by $\mathcal{O}(\log m)$ calls to Algorithm 5.3 plus an overhead of $\mathcal{O}(m)$ elementary operations.

Notice that by construction the interval $[K_i, K_{i+1}]$ does not contain any critical points in the interior. If $K_i = K_{i+1}$, then we know that $K^* = K_i = K_{i+1}$. This way we have determined K^*. In this case we can compute the order of all edges with respect to h_{K^*} in $\mathcal{O}(m \log m)$ time and stop the modified sorting algorithm.

Otherwise, the interior of $[K_i, K_{i+1}]$ is nonempty. By one call to Algorithm 5.3 we can find out whether $K^* \leq K_i$, which implies that $K^* = K_i$ since we know that $K^* \in [K_i, K_{i+1}]$. Again, the adopted sorting procedure can stop after having computed the ordering of all edges with respect to h_{K^*}.

The remaining case is that $K_i < K^* \leq K_{i+1}$. In this case it is easy to see that answering the comparisons from the first round by inspecting the weights h_τ, where $\tau \in (K_i, K_{i+1})$ is any interior point of the interval $[K_i, K_{i+1}]$, gives the same results as answering the comparisons with respect to the h_{K^*}-weights.

Thus, at the end of the first round, our algorithm has either found K^* and thus the ordering of *all* edges in the graph with respect to their h_{K^*}-weights, or we can answer the comparisons from the first round using the ordering of the edges with respect to h_τ.

The above process is repeated $\mathcal{O}(\log m)$ times, once for each parallel step of the parallel sorting machine. Since in each of the $\mathcal{O}(\log m)$ rounds we answer all comparisons of the parallel sorting scheme, upon termination we have found the ordering of the edges with respect to the h_{K^*}-weights.

The time needed for the first stage of the algorithm above can be estimated as follows: There are $\mathcal{O}(\log m)$ cycles altogether. In each round we evaluate $\mathcal{O}(m)$ intersection points. Also, we need $\mathcal{O}(\log m)$ calls to Algorithm 5.3 plus the overhead of $\mathcal{O}(m)$. Since, $T_{test}(n) \in \mathcal{O}(n^2)$ and $m \in \mathcal{O}(n^2)$, we have the following lemma:

Lemma 5.16 *The improved heuristic computes the ordering of the edges with respect to* h_{K^*} *in time* $\mathcal{O}(n^2 \log^2 n)$. □

Finding the Right Placement

Let P_{slow} be the placement generated by our slow heuristic for Δ-(SUM, SUM, p NODES). We have already argued that, given the ordering of the edges with respect to h_{K^*}, we can find a set $\mathcal{P} = \{P(v_1), \dots, P(v_n)\}$ of placements such that $P_{slow} \in \mathcal{P}$. By the construction of our slow algorithm, it follows that P_{slow} is a placement $P(v_j)$ in $\mathcal{P}$ with minimum $\hat{\mathcal{S}}_{h_{K^*}}(P(v_j))$. We now show how to find P_{slow} in the set $\mathcal{P}$ efficiently.

For each placement $P(v_i) \in \mathcal{P}$ denote by $C_i := \hat{\mathcal{S}}_c(P(v_i))$ and $D_i := \hat{\mathcal{S}}_d(P(v_i))$ the simplified sum of the c-weights and d-weights respectively (see Equation (5.1)). Clearly, all the C_i and D_i can be found in an overall time of $\mathcal{O}(np)$.

Observe that $\hat{\mathcal{S}}_{h_K}(P(v_i)) = C_i + K\frac{D_i}{B}$. Our task now becomes to find the placement having minimum value when $K = K^*$. Again, we can view $\hat{\mathcal{S}}_{h_K}(P(v_i))$ as a linear function of K. The ordering of two placements changes, when two of the functions intersect. Using the same technique as finding the ordering of the *edges* with respect to h_{K^*}, we can find the ordering of the *placements* in $\mathcal{P}$ with respect to $\hat{\mathcal{S}}_{h_{K^*}}$ in time $\mathcal{O}(m \log m + T_{test}(n))$.

This enables us to find the same placement as the slow heuristic in an overall time of $\mathcal{O}(m \log m + T_{test}(n) \log^2 m)$. Since $T_{test}(n) \in \mathcal{O}(n^2)$ and $m \in \mathcal{O}(n^2)$, we obtain the following theorem.

Theorem 5.17 *For any fixed $\varepsilon > 0$, the improved heuristic for* Δ-(SUM, SUM, p NODES) *has a performance of* $((2 - 2/p)(1 + 1/\varepsilon), (2 - 2/p)(1 + \varepsilon))$ *and a running time of* $\mathcal{O}(n^2 \log^2 n)$. □

5.7 Extension to the Node-Weighted Case

We now briefly discuss how to extend our algorithms to apply to the case when we additionally have weights for each node in G, and the minimization objective is a function of both the edge weights and node weights. For the sake of brevity, we will illustrate our ideas for such an extension by considering one specific problem, namely Δ-(SUM, DIA, p NODES). The approximation algorithms for the other problems can be extended in a similar fashion.

The input consists again of a complete undirected graph $G = (V, E)$ with edge-weight functions c, d and weights $\omega(v)$, for each $v \in V$. The goal is to find a placement P, with $|P| = p$, such that the objective function

$$\mathcal{S}_c^{\omega}(P) = \sum_{\substack{(v,w) \in E \\ v,w \in P}} c(v,w) + \sum_{v \in P} \omega(v)$$

is minimized subject to the same constraint as for the (SUM, DIA, p NODES) problem, namely $\mathcal{D}_d(P) \le B$. We denote this extension of the (SUM, DIA, p NODES) problem by (SUM, DIA, p NODES)$^{\omega}$.

To obtain an approximate solution for Δ-(SUM, DIA, p NODES)$^{\omega}$, we transform a given instance of Δ-(SUM, DIA, p NODES)$^{\omega}$ into an instance of Δ-(SUM, DIA, p NODES) as discussed in the proof of the following lemma.

Lemma 5.18 *Given any instance* I *of* Δ-(SUM, DIA, p NODES)$^{\omega}$*, there are linear time transformations* f *and* g *such that both of the following conditions hold.*

1. *The transformation* f *constructs an instance* I' *of* Δ-(SUM, DIA, p NODES) *from* I.
2. *Given a placement* P' *of* p *nodes for* I' *with diameter* D *and objective function value* C*,* g *constructs a placement* P *of* $p' = p$ *nodes for* I *such that* P *has diameter* D *and objective function value* C.

Proof: Given the instance I, the transformation f constructs an instance I' of Δ-(SUM, DIA, p NODES) as follows.

The nodes in $G' = (V', E')$ are in one-to-one correspondence with the nodes of G. The d'-cost on each edge in G' is the same as the d-cost of the corresponding edge in G. The c'-cost for an edge (v', w') in G' is defined as follows:

$$c'(v', w') := c(v, w) + \frac{1}{2p}(\omega(v) + \omega(w)).$$

It is easy to check that the triangle inequality is satisfied for c'. Next consider a placement P' of p nodes in I'. The transformation g chooses as its placement P the p nodes in I corresponding to the p nodes in I'. By a straightforward calculation, it can be seen that for any placement P of p nodes

$$\sum_{\substack{(v',w') \in E' \\ v',w' \in P'}} c'(v', w') = \sum_{v \in P} \omega(v) + \sum_{\substack{(v,w) \in E \\ v,w \in P}} c(v, w).$$

□

Combining Lemma 5.18 with the approximation results from Theorem 5.9, we obtain a $(2, 2 - 2/p)$-approximation for Δ-(SUM, DIA, p NODES)$^{\omega}$.

5.8 Hardness Results

In this section, we prove our non-approximability results for the compact location problems under study. Several of our hardness results use recent hardness results concerning approximability of the MAX CLIQUE problem.

Definition 5.19 (CLIQUE, MAX CLIQUE)
An instance of the **CLIQUE** *problem consists of an undirected graph* $G = (V, E)$ *and an integer* $J \leq |V|$. *The question is whether there exists a clique of size* J *in* G, *that is, whether there exist* J *vertices* $C = \{v_1, \dots, v_J\}$ *such that for all* $v_i, v_k \in C$ *we have* $(v_i, v_k) \in E$.

The optimization version of the problem, denoted by **MAX CLIQUE** *, is to find a clique of maximum size in* G.

Recently, using a new characterization of NP in terms of *probabilistically checkable proof systems*, it has been shown that the MAX CLIQUE problem cannot be efficiently approximated (cf. Theorem 3.13 on page 37). We begin with a non-approximability result for Δ-(DIA, DIA, p NODES). Recall that the goal of the problem is to find a subset of p nodes of minimum c-diameter subject to a bound B on its d-diameter.

Lemma 5.20 *Unless* $P = NP$, *for all* $\varepsilon, \varepsilon' > 0$, *there can be no polynomial time algorithm* A *with the following properties: Given any instance of* Δ-(DIA, DIA, p NODES), *the algorithm* A

- *either returns a vertex subset* $S \subseteq V$ *of at least* $2p/|V|^{1/6-\varepsilon'}$ *nodes satisfying* $\mathcal{D}_d(S) \leq (2-\varepsilon)B$,
- *or provides the information that there is no placement of* p *nodes having* d*-diameter of at most* B.

Proof: Let I be an arbitrary instance of MAX CLIQUE , given by the graph $G = (V, E)$. Without loss of generality we can assume that $E \neq \varnothing$. We will give a many-one Turing reduction to the problem Δ-(DIA, DIA, p NODES).

For each $2 \leq k \leq |V|$ we construct an instance $I^{(k)}$ of Δ-(DIA, DIA, p NODES) as follows: Let $G^{(k)} = (V, E')$ with $E' = \{(u, v) : u, v \in V, u \neq v\}$ and define $c^{(k)}$, $d^{(k)}$ on E' by $c^{(k)}(e) := 1$ for all $e \in E'$ and

$$d^{(k)}(e) := \begin{cases} 1 & \text{if } e \in E, \\ 2 & \text{otherwise.} \end{cases} \tag{5.10}$$

It is trivial to see that both weight functions obey the triangle inequality. We let $B^{(k)} := 1$ and $p^{(k)} := k$. Notice that the size of an instance $I^{(k)}$ is still polynomial in the size of I, and that we have constructed only polynomially many (namely $\mathcal{O}(|V|)$) instances.

Now consider an instance $I^{(k)}$. Note that any placement P of $p^{(k)} = k$ nodes that has diameter $\mathcal{D}_{d^{(k)}}(P) \leq (2-\varepsilon)B^{(k)} = (2-\varepsilon)$ must have diameter 1. Also, any subset $S \subseteq V$ having diameter $\mathcal{D}_{d^{(k)}}(S) = 1$ must form a clique in the original graph G.

Assume that the original graph G has a clique C of size $p^{(k)} = k$. Then this clique will satisfy $\mathcal{D}_{c^{(k)}}(C) = \mathcal{D}_{d^{(k)}}(C) = 1 = B^{(k)}$. By our assumption, the algorithm A must return a set S of at least $2p^{(k)}/|V|^{1/6-\varepsilon'}$ nodes with d-diameter at most $(2-\varepsilon)B^{(k)} = (2-\varepsilon) < 2$. Thus, as noted above, the algorithm must find a placement of diameter 1, and this set will form a clique in the original graph G.

If there is no clique of size $p^{(k)} = k$ in G, any placement P of $p^{(k)} = k$ nodes in $G^{(k)}$ must include at least one edge e of length $d^{(k)}(e) = 2 > (2-\varepsilon)$. Now, according to our assumptions about A, the algorithm has the choice of either returning a set of size at least $2p^{(k)}/|V|^{1/6-\varepsilon'}$ that will form a clique in the original graph or providing the information that there is no placement P of d-diameter at most 1.

Thus, the output of algorithm A can be used to either obtain the information that G does not contain a clique of size $p^{(k)} = k$ or that G contains a clique of size at least $2k/|V|^{1/6-\varepsilon'}$.

Now, we run A for all the instances $I^{(k)}$ $(2 \le k \le |V|)$. According to our assumptions about A, this will result in an overall polynomial time algorithm, since the size of each instance $I^{(k)}$ is polynomial in the size of I and we only have $\mathcal{O}(|V|)$ instances. Let

$$s := \max\{\, k : \mathsf{A} \text{ returns a set of diameter } 1 \,\}.$$

Then, by our observations from above, we can conclude that G has a clique of size at least $2s/|V|^{1/6-\varepsilon'}$ and that there is no clique of size $s+1$ in G. Hence, we can approximate the maximum clique number of G by a factor of at most

$$\frac{s+1}{2s} \cdot |V|^{1/6-\varepsilon'} \le |V|^{1/6-\varepsilon'}.$$

By the results from Theorem 3.13 this implies $\mathsf{P}=\mathsf{NP}$. $\square$

We can interchange the roles of c and d in the proof of the last lemma to show that the optimal value of Δ-(DIA, DIA, p NODES) cannot be approximated by a factor of $2-\varepsilon$, unless $\mathsf{P} = \mathsf{NP}$, even if we allow the algorithm to violate the constraint arbitrarily. Moreover, we can replace the 2 in (5.10) by any polynomial time computable function f. This enables us to conclude similar to the proof of Theorem 4.12 on page 51 that, if the triangle inequality is not required to hold, there can be no polynomial time approximation with performance ratio $f(n)$ for neither the optimal function value nor the constraint, provided $\mathsf{P} \neq \mathsf{NP}$, where $n := |V|$ is the number of nodes in the graph. Thus we obtain:

Theorem 5.21 *Unless* $\mathsf{P} = \mathsf{NP}$*, the following statements hold:*

(i) If the c*-costs and* d*-costs do not satisfy the triangle inequality, then the problem* (DIA, DIA, p NODES) *does not have an approximation algorithm with a performance of* $(f(n), g(n))$ *for any polynomial time computable functions* f *and* g*.*

(ii) For any fixed $\varepsilon > 0$ *and* $\varepsilon' > 0$ *and any* $\alpha \geq 1$*, the problem* Δ-(DIA, DIA, p NODES) *does not have a polynomial time approximation algorithm which finds a placement of at least* $2p/|V|^{1/6-\varepsilon'}$ *facilities and which provides a performance guarantee of* $(\alpha, 2-\varepsilon)$ *or* $(2-\varepsilon, \alpha)$*.* □

The results of Lemma 5.20 also give a slightly stronger hardness result than the one stated in the last theorem. Namely, we can conclude that, unless *both* distances c and d satisfy the triangle inequality, for any fixed $\alpha, \beta \geq 1$ we can not hope to obtain an (α, β)-approximation algorithm for the (DIA, DIA, p NODES) problem. We now turn our attention to (SUM, DIA, p NODES) and Δ-(SUM, DIA, p NODES), respectively:

Theorem 5.22 *Unless* $\mathsf{P} = \mathsf{NP}$*, the following statements hold:*

(i) For any polynomial time computable functions f *and* g*, the problem* (SUM, DIA, p NODES) *does not have a* $(f(n), g(n))$*-approximation algorithm.*

(ii) For any $\alpha \geq 1$ *and any fixed* $\varepsilon > 0$*, the problem* Δ-(SUM, DIA, p NODES) *does not have a* $(\alpha, 2-\varepsilon)$*-approximation algorithm.*

Proof: We first consider the (SUM, DIA, p NODES) problem. Suppose there is a polynomial approximation algorithm A with a performance guarantee of $(f(n), g(n))$ for some polynomial time computable functions f and g. We will show that A can be used to solve an arbitrary instance of the CLIQUE problem in polynomial time, contradicting the assumption that $\mathsf{P} \neq \mathsf{NP}$.

Let $G = (V, E)$ and the integer J form an arbitrary instance of CLIQUE. Construct the following instance of (SUM, DIA, p NODES): The vertex set for the instance is V itself. For all $u, v \in V$ $(u \neq v)$, let $d(u, v) = 1$. Let $c(u, v) = 1$ if $(u, v) \in E$ and $c(u, v) = g(n) + 1$ otherwise. Finally set $p := J$ and $B := 1$ to complete the construction. In the remainder of this proof, we will refer to any edge in the instance I with d value equal to $g(n) + 1$ as a *long edge*.

If G has a clique of size J, then the nodes which form this clique constitute an optimal solution to the (SUM, DIA, p NODES) instance with sum (under c-distance) equal to $J(J-1)/2$ and diameter (under d-distance) equal to 1. Since A provides a performance guarantee of $(f(n), g(n))$, the solution returned by A cannot include any long edges. If G does not have a clique of size J, then every subset of J nodes must include at least one long edge. Therefore, by merely examining the solution produced by A, we can solve the CLIQUE problem.

We use the same construction for Δ-(SUM, DIA, p NODES) except that for every long edge, the c and d weights are chosen as 2. This ensures that both distance functions satisfy the triangle inequality. Recall that an $(\alpha, 2-\varepsilon)$-approximation algorithm must either return a placement of d-diameter at most $2-\varepsilon$ or provide the information that no placement of diameter 1 exists. Similarly to the proof of Lemma 5.20 it follows that there is such a placement violating the constraint by a factor of at most $2-\varepsilon$ if and only if G has a clique of size J. □

Again, the proof of Theorem 5.22 can be used to show that, if only one of the distance functions c and d does not obey the triangle inequality, no polynomial time (α, β)-approximation for any fixed $\alpha, \beta \geq 1$ can exist, unless $\mathsf{P} = \mathsf{NP}$.

Using ideas similar to those used in the proof of Theorems 5.21 and 5.22, the following hardness results for the sum-constrained problems and unicriterion problems can be obtained:

Theorem 5.23 *Unless* $\mathsf{P} = \mathsf{NP}$*, the problems* (SUM, SUM, p NODES) *and* (DIA, SUM, p NODES) *can not be solved in polynomial time, even when both distance functions,* c *and* d*, obey the triangle inequality. Moreover, unless* $\mathsf{P} = \mathsf{NP}$*, the following statements are true:*

(i) For all fixed $\alpha, \beta \geq 1$*, if the* c*- and* d*-costs do not satisfy the triangle inequality, the problems* (SUM, SUM, p NODES) *and* (DIA, SUM, p NODES) *do not have a polynomial time* (α, β)*-approximation algorithms.*

(ii) For all fixed $\varepsilon > 0$ *and* $\beta \geq 1$*, the problem* Δ-(DIA, SUM, p NODES) *does not have a polynomial time* $(2-\varepsilon, \beta)$*-approximation algorithm.* □

Theorem 5.24 *The problems* (DIA, p NODES)*,* (SUM, p NODES) *and* (VAR, p NODES) *are* NP*-hard, even when the distances satisfy the triangle inequality. Unless* $\mathsf{P} = \mathsf{NP}$*, if the distances are not required to satisfy the triangle inequality, then for any polynomial time computable function* f *there is no polynomial time approximation algorithm for any of the problems* (DIA, p NODES)*,* (SUM, p NODES)*,* (VAR, p NODES) *with a performance of* $f(n)$*.* □

5.9 Compact Location Problems for Tree Networks

Here we consider compact location problems where the input is restricted to be a tree based distance structure. In this case, the distances between two vertices in the graph correspond to path lengths along the given tree $T = (V, E)$. Hence, we extend the distances on T to obtain a complete graph on the vertices V.

Definition 5.25 (Tree based distance structure)
*A **tree based distance structure** is a tree $T = (V, E_T)$ and a nonnegative length $c(e)$ assigned to each edge $e \in E_T$ of the tree T. For each pair (v, w) of vertices in V, the distance $c(v, w)$ equals the sum of the edge lengths along the unique path in T connecting v and w.*

Versions of compact location problems can be defined for graphs with tree based distance structures, in the same manner as we defined them for arbitrary graphs but the distances are now specified by a tree based distance structure. We denote these problems by the prefix "TREE". For instance, for the TREE-(DIA, DIA, p NODES) problem the input is a tree T, two edge weight functions c and d defined on T, a bound B and an integer p. The requirement is to find a subset consisting of p nodes, whose d-diameter is at most B and minimizing the c-diameter among all these constrained placements.

The Unicriterion Case

We first consider the unicriterion case. Here, we are given only *one* weight function c defined on the tree T. The two problems to be identified are TREE-(SUM, p NODES) and TREE-(DIA, p NODES), where the goal is to find a placement of p nodes minimizing the objective $\mathcal{S}_c$ and $\mathcal{D}_c$ respectively.

We outline a polynomial time algorithm to solve the TREE-(SUM, p NODES) problem optimally. We first define the concept of *openness* for a placement of facilities and then prove that among the optimal solution for the problem TREE-(SUM, p NODES) there is a solution which is closed. Using this fact we can employ a dynamic programming technique to obtain an optimal solution.

Definition 5.26 (Closed set of nodes)
*Given a tree based distance structure with node set V, a set of nodes $A \subseteq V$ is **closed** if for all $v, w \in A$, every node in the (unique) tree path between v and w is in A.*

Lemma 5.27 *For any instance of TREE-(SUM, p NODES), there is at least one optimal solution which forms a closed set of nodes.*

Proof: Consider a set of $P \subseteq V$. Define the *openness* $o_P(v, w)$ of a pair of nodes v and w in P to be the number of nodes along the (unique) path between v and w that are not in P. Let

$$o(P) = \sum_{v,w \in P} o_P(v, w).$$

In the above equation we denote by E the edge set of the complete graph $G = (V, E)$ induced by the tree T such that each pair (v, w) of vertices in P

is counted only once. Among the optimal solutions to the instance, let P be one of minimal $o(P)$. Assume that $o(P)$ is nonzero. We will show that this leads to a contradiction.

Note that the deletion of any edge from the tree T partitions the tree into two disconnected subtrees. Since $o(P)$ is nonzero, there is an edge (u, v) in the tree T such that $u \in P$, and $v \notin P$, and if the edge (u, v) were to be deleted from T, the induced subtrees T_u and T_v containing u and v respectively would each contain at least one member of P. We can assume without loss of generality that the subtree containing u would contain a number $j \leq p/2$ of nodes from P.

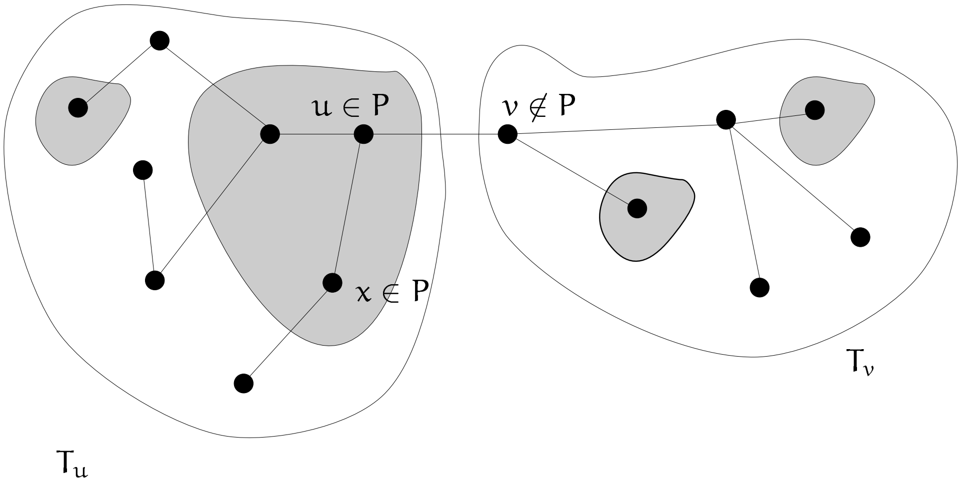


Figure 5.2: Partition of the tree T into subtrees in the proof of Lemma 5.27. The tree T_u contains $j \leq p/2$ nodes from P.

Also, without loss of generality we can assume that T_u contains a vertex $x \in P$ such that every node between x and u is in P and x does not occur as an internal node in the path between two vertices in $P \setminus \{x\}$. This vertex x is a leaf of the subtree of T_u induced by the vertices of P contained in T_u (see Figure 5.2 for an illustration). We now consider the set $P' = (P \setminus \{x\}) \cup \{v\}$. Then

$$\begin{aligned} \mathcal{S}_c(P') &= \mathcal{S}_c(P) - \sum_{w \in P \setminus \{x\}} c(x, w) + \sum_{w \in P \setminus \{x\}} c(w, v) \\ &= \mathcal{S}_c(P) + \sum_{w \in P \setminus \{x\}} \left(c(v, w) - c(x, w)\right). \end{aligned} \tag{5.11}$$

Let P_u be the members of $P \setminus \{x\}$ that are in the subtree T_u and let P_v denote the members of P that are contained in T_v. Then $|P_u| = j - 1$, and $|P_v| = p - j$. For each node $w \in P_u$, $c(w,v) \le c(w,x) + c(x,v)$. Hence

$$c(w,v) - c(w,x) \le c(x,v) \qquad \text{for all } w \in P_u. \tag{5.12}$$

For each node $w \in P_v$ we have $c(x,w) = c(x,v) + c(v,w)$, since the path from x to w must pass through v. Thus

$$c(w,v) - c(w,x) = -c(x,v) \qquad \text{for all } w \in P_v. \tag{5.13}$$

Using (5.11) to (5.13) we obtain

$$\begin{aligned}\mathcal{S}_c(P') &= \mathcal{S}_c(P) + \sum_{w \in P_u} (c(w,v) - c(w,x)) + \sum_{w \in P_v} (c(w,v) - c(w,x)) \\ &\le \mathcal{S}_c(P) + (j-1)c(x,v) - (p-j)c(x,v) \\ &= \mathcal{S}_c(P) - (p - 2j + 1)c(x,v).\end{aligned}$$

Since $j \le p/2$, it follows that $p - 2j + 1 > 0$, so $\mathcal{S}_c(P') \le \mathcal{S}_c(P)$. If $\mathcal{S}_c(P') < \mathcal{S}_c(P)$, there is an immediate contradiction to the assumption that P is an optimal solution. So assume $\mathcal{S}_c(P') = \mathcal{S}_c(P)$, in which case P' is an optimal solution.

Consider $o(P')$. Because of the manner in which x was selected, for each $y, z \in P \setminus \{x\}$, $o_{P'}(y,z)$ is no greater than $o_P(y,z)$. Also, by the choice of x, the path between x and v does not contain any vertices not in $P \setminus \{v\}$. Thus, for each vertex $w \in P_u$, we have $o_{P'}(w,v) = o_P(w,x)$.

For each node $w \in P_v$, the openness $o_{P'}(w,v)$ is strictly less than $o_P(x,v)$, since the path in P between x and w had to pass through v. Since $P_v \neq \varnothing$, it follows that $o(P') < o(P)$, contradicting the assumption that P is a solution with minimum $o(P)$. ↯ □

The proof of the following lemma can be done similarly.

Lemma 5.28 *For any instance of* TREE-(DIA, p NODES)*, there is at least one optimal solution which forms a closed set of nodes.* □

Theorem 5.29 TREE-(SUM, p NODES) *can be solved optimally in $\mathcal{O}(p^2 n)$ time.*

Proof: We use a dynamic programming based procedure to obtain our solution. Envision the tree T as being rooted at an arbitrarily selected node, say node r. In the sequel we denote by T_v the subtree of T rooted at vertex v. Thus, in particular $T = T_r$.

The algorithm processes the tree T in a bottom up manner. First the leaf nodes are processed. Then, a nonleaf node is processed after all its children have been processed.

The algorithm is based on the results of Lemma 5.27. We know that there is an optimal solution P^* which is closed. Thus, there must be a subtree of T_{z^*} rooted at some node $z^* \in P^*$ containing P^*.

Let $z \in V$ be an arbitrary node and let A be a closed set of nodes contained in the subtree rooted at z such that $z \in A$. Define

$$C^z(A) = \sum_{\substack{(x,y)\in E\\ x,y\in A\setminus\{z\}}} c(x,y) + (p - |A| + 1) \sum_{x\in A\setminus\{z\}} c(x,z).$$

In the above equation we denote again by E the edge set of the complete graph $G = (V, E)$ induced by the tree T. If $|A| = p$, then clearly $C^z(A) = \mathcal{S}_c(A)$. We now define the values C_i^z for $1 \le i \le p$ in the following way. If the subtree T_z rooted at z contains at least i nodes, then

$$C_i^z := \min_A \left\{ C^z(A) : \begin{array}{l} A \text{ is a closed set of } i \text{ nodes in the subtree} \\ T_z \text{ rooted at } z \text{ and } z \in A \end{array} \right\}.$$

If the subtree T_z headed by z contains fewer than i nodes, then define $C_i^z := \infty$. If we know C_p^z for all vertices z, then by Lemma 5.27 we can easily find the optimal function value by taking the minimum over all values C_p^z. If along with the values C_i^z we also store the corresponding sets of nodes, this will enable us to output an optimal solution. We will now show how to compute all the C_i^z bottom up starting with the leaves of the tree T.

Suppose that node z is a leaf, i.e., $T_z = \{z\}$. Then clearly

$$C_1^z = 0 \qquad \text{and} \qquad C_i^z = \infty \quad \text{for } 2 \le i \le p.$$

Now suppose that z is a nonleaf. Let $\delta(z)$ be the number of children of z, and let $v_1, \cdots, v_{\delta(z)}$ denote these children, listed in some arbitrary order. Define auxiliary values $C_i^{z,j}$ for $1 \le i \le p$, $1 \le j \le \delta(z)$ as follows:

$$C_i^{z,j} := \min_A \left\{ C^z(A) : \begin{array}{l} A \text{ is a closed set of } i \text{ nodes in } T_z,\ z \in A, \text{ and every} \\ \text{other node from } A \text{ lies in one of the subtrees} \\ T_{v_1}, \ldots, T_{v_j} \text{ rooted at } v_1, \ldots, v_j \text{ respectively} \end{array} \right\}$$

Then, for the node z we have the following equality

$$C_i^z = C_i^{z,\delta(z)} \qquad \text{for } 1 \le i \le p.$$

The values of $C_i^{z,j}$ can be computed as follows. Clearly, $C_1^{z,1} = 0$. Now consider a closed set A of $i \ge 2$ nodes containing z such that $A \setminus \{z\}$ is contained in the subtree T_{v_1} routed at v_1. Since A is closed and contains vertices from the subtree T_{v_1}, it must contain the root vertex v_1 of that subtree. For any vertex $x \in T_{v_1}$ we have $c(x,z) = c(x,v_1) + c(v_1,z)$, because the unique path between z and x (in T) passes through v_1. Thus, we can write the sum $C^z(A)$ in the following way:

$$\begin{aligned}
C^z(A) &= \sum_{\substack{(x,y)\in E\\ x,y\in A\setminus\{z\}}} c(x,y) + (p-i+1)\sum_{x\in A\setminus\{z\}} c(x,z)\\
&= \sum_{\substack{(x,y)\in E\\ x,y\in A\setminus\{z,v_1\}}} c(x,y) + \sum_{x\in A\setminus\{z,v_1\}} c(x,v_1) + (p-i+1)c(v_1,z) + (p-i+1)\sum_{x\in A\setminus\{z,v_1\}} c(x,z)\\
&= \sum_{\substack{(x,y)\in E\\ x,y\in A\setminus\{z,v_1\}}} c(x,y) + \sum_{x\in A\setminus\{z,v_1\}} c(x,v_1) + (p-i+1)c(v_1,z)\\
&\quad + (p-i+1)\sum_{x\in A\setminus\{z,v_1\}} (c(x,v_1) + c(v_1,z))\\
&= \sum_{\substack{(x,y)\in E\\ x,y\in A\setminus\{z,v_1\}}} c(x,y) + (p-(i-1)+1)\sum_{x\in A\setminus\{z,v_1\}} c(x,v_1) + (i-1)(p-i+1)c(v_1,z)\\
&\geq (i-1)(p-i+1)c(v_1,z) + C^{v_1}_{i-1},
\end{aligned}$$

where equality holds in the last line if and only if $A \setminus \{z\}$ is a set such that $C^{v_1}(A \setminus \{z\}) = C^{v_1}_{i-1}$. Thus,

$$\begin{aligned}
C^{z,1}_i &= \min \min_A \left\{ C^z(A) : \begin{array}{l} A \text{ is a closed set of } i \text{ nodes in } T_z,\\ z \in A, \text{ and every other node from } A\\ \text{lies in } T_{v_1} \end{array} \right\}\\
&= (i-1)(p-i+1)c(z,v_1) + C^{v_1}_{i-1}.
\end{aligned}$$

The above results enable us to compute the values $C^{z,1}_i$ for all $1 \leq i \leq p$:

$$\begin{aligned}
C^{z,1}_1 &= 0\\
C^{z,1}_i &= (i-1)(p-i+1)c(z,v_1) + C^{v_1}_{i-1}, \qquad \text{for } 2 \leq i \leq p.
\end{aligned}$$

Our next step consists of computing the values $C^{z,j+1}_i$ where we assume that $C^{z,j}_i$ is known for all $1 \leq i \leq p$. Consider a closed set A of i nodes containing z such that every node in $A \setminus \{z\}$ lies in one of the subtrees $T_{v_1}, \ldots, T_{v_{j+1}}$. For brevity, we will let A' denote the vertices contained in the first j subtrees and A'' denote the vertices contained in the subtree T_{v_j} rooted at v_{j+1}. Then, $A = A' \cup A'' \cup \{z\}$ and, by construction, this union is disjoint. In the sequel we assume that $|A'| = t-1$ and $|A''| = i-t \geq 1$.

$$C^z(A) = \sum_{\substack{(x,y)\in E\\ x,y\in A'\cup A''}} c(x,y) + (p-i+1)\sum_{x\in A'\cup A''} c(x,z). \tag{5.14}$$

We can write the first term in (5.14) as follows

$$\begin{aligned}\sum_{\substack{(x,y)\in E\\ x,y\in A'\cup A''}} c(x,y) &= \sum_{\substack{(x,y)\in E\\ x,y\in A'}} c(x,y) + \sum_{\substack{(x,y)\in E\\ x,y\in A''}} c(x,y) + \sum_{\substack{(x,y)\in E\\ x\in A',y\in A''}} c(x,y)\\ &= \sum_{\substack{(x,y)\in E\\ x,y\in A'}} c(x,y) + \sum_{\substack{(x,y)\in E\\ x,y\in A''}} c(x,y) + \sum_{\substack{(x,y)\in E\\ x\in A',y\in A''}} (c(x,z)+c(z,y))\\ &= \sum_{\substack{(x,y)\in E\\ x,y\in A'}} c(x,y) + \sum_{\substack{(x,y)\in E\\ x,y\in A''}} c(x,y) + (i-t)\sum_{x\in A'} c(x,z) + (t-1)\sum_{y\in A''} c(y,z).\end{aligned} \tag{5.15}$$

The second sum in (5.14) can be rewritten in the following way

$$\begin{aligned}\sum_{x\in A'\cup A''} c(x,z) &= \sum_{x\in A'} c(x,z) + \sum_{x\in A''} (c(x,v_{j+1}) + c(v_{j+1},z))\\ &= \sum_{x\in A'} c(x,z) + \sum_{x\in A''} c(v_{j+1},x) + (i-t-1)c(v_{j+1},z).\end{aligned} \tag{5.16}$$

From (5.15) and (5.16) we obtain

$$\begin{aligned}C^z(A) &= \sum_{\substack{(x,y)\in E\\ x,y\in A'}} c(x,y) + (p-t+1)\sum_{x\in A'} c(x,z)\\ &\quad + \sum_{\substack{(x,y)\in E\\ x,y\in A''}} c(x,y) + (t-1)\sum_{y\in A''} c(y,z)\\ &\quad + (p-i+1)\sum_{x\in A''} c(v_{j+1},x) + (p-i+1)(i-t-1)c(v_{j+1},z)\\ &\geq C_t^{z,j} + \sum_{\substack{(x,y)\in E\\ x,y\in A''}} c(x,y) + (p-(i-t))\sum_{y\in A''} c(v_{j+1},y)\\ &\quad + (p-i+t)(i-t-1)c(v_{j+1},z)\\ &= C_t^{z,j} + \sum_{\substack{(x,y)\in E\\ x,y\in A''\setminus\{v_{j+1}\}}} c(x,y) + (p-(i-t)+1)\sum_{y\in A''} c(v_{j+1},y)\\ &\quad + (p-i+t)(i-t-1)c(v_{j+1},z)\\ &\geq C_t^{z,j} + C_{i-t}^{v_{j+1}} + (p-i+t)(i-t-1)c(v_{j+1},z).\end{aligned}$$

In the above chain of inequalities, the equality is achieved if and only if we have $C^z(A'\cup\{z\}) = C_t^{z,j}$ and $C^{v_{j+1}}(A'') = C_{i-t}^{v_{j+1}}$. Thus, we can compute $C_i^{z,j+1}$

by the following equation:

$$C_i^{z,j+1} = \min\left\{C_i^{z,j}, \min_{t<i}\left(C_t^{z,j} + C_{i-t}^{v_{j+1}} + (p-i+t)(i-t-1)c(z,v_{j+1})\right)\right\}$$

for $1 \le i \le p$, $2 \le j \le \delta(z)$.

We now have a way to compute all the auxiliary values $C_i^{z,j}$ and hence can compute C_i^z. This way, all necessary values can be computed bottom up. As mentioned above an optimal solution to the TREE-(SUM, p NODES) problem instance has a cost given by the minimal value of the C_p^z over all nodes z. It now follows easily that we find an optimal solution by the above method in $\mathcal{O}(p^2n)$ time. □

Along the same lines one can prove the following result on the polynomial time solvability of TREE-(DIA, p NODES).

Theorem 5.30 TREE-(DIA, p NODES) *can be solved in* $\mathcal{O}(p^2n)$ *time.* □

Bicriteria Compact Location Problems on Trees

In this section we investigate the complexity of bicriteria compact location problems if restricted to trees. It turns out that the problem TREE-(SUM, SUM, p NODES) is NP-hard. In contrast, the other three constrained compact location problems for trees (namely TREE-(SUM, DIA, p NODES), TREE-(DIA, DIA, p NODES) and TREE-(DIA, SUM, p NODES)) are polynomial time solvable.

The Complexity of TREE-(SUM, SUM, p NODES)

The following result points out that obtaining an optimal solution to the TREE-(SUM, SUM, p NODES) problem is difficult.

Lemma 5.31 TREE-(SUM, SUM, p NODES) *is* NP*-hard to solve.*

Proof: We use a reduction from the PARTITION problem, which is known to be NP-complete [GJ79]. Given a multiset of (not necessarily distinct) positive integers $\{a_1, \ldots, a_n\}$, the question is whether there exists a subset $U \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in U} a_i = \sum_{i \notin U} a_i$.

Given any instance of PARTITION we construct a star-shaped graph G having $n+1$ nodes $\{x, x_1, \ldots, x_n\}$ and n edges (x, x_i), $i = 1, \ldots, n$. We then define $c(x, x_i) := a_i$ and $d(x, x_i) := D - a_i$, where $D := \sum_{i=1}^n a_i$.

We then run the hypothetical polynomial time algorithm A for TREE-(SUM, SUM, p NODES) for the instance I_j consisting of the graph defined as above and the parameters p_j, B_j, $j = 1, \ldots, n$ where $p_j := j + 1$ and $B_j := jD/2$. Observe that this will still result in an overall polynomial time.

Let $j \in \{1, \ldots, n-1\}$ be fixed and assume that P is any placement of $p_j = j+1$ nodes that includes the node x. Then

$$\sum_{x_i \in P \setminus \{x\}} c(x, x_i) = \sum_{x_i \in P \setminus \{x\}} a_i. \tag{5.17}$$

Consider a fixed node $x_i \in P$, $x_i \neq x$. Then

$$\begin{aligned} \sum_{x_k \in P \setminus \{x_i, x\}} c(x_i, x_k) &= \sum_{x_k \in P \setminus \{x_i, x\}} [c(x_i, x) + c(x, x_k)] \\ &= (j-1) \cdot c(x_i, x) + \sum_{x_k \in P \setminus \{x_i, x\}} c(x, x_k) \\ &= (j-2) \cdot c(x_i, x) + \sum_{x_k \in P \setminus \{x\}} c(x, x_k) \\ &= (j-2) a_i + \sum_{x_k \in P \setminus \{x\}} a_k. \end{aligned} \tag{5.18}$$

Using the identities in (5.17) and (5.18) we obtain

$$\mathcal{S}_c(P) = j \sum_{x_j \in P - \{x\}} a_j. \tag{5.19}$$

Similarly it follows that

$$\mathcal{S}_d(P) = jD - j \sum_{x_j \in P \setminus \{x\}} a_j. \tag{5.20}$$

Moreover, if P is any *feasible* placement for I_j (i.e., $\mathcal{S}_d(P) \leq B_j$) which includes the node x, then using Equation (5.20) and the definition of B_j, we obtain

$$\mathcal{S}_d(P) = jD - j \sum_{x_j \in P \setminus \{x\}} a_j \leq j \frac{D}{2}.$$

Thus for such a placement we get

$$\sum_{x_j \in P \setminus \{x\}} a_j \geq D/2. \tag{5.21}$$

So far we have considered only placements that include the node x. The striking point now is that any optimal feasible placement for I_j *must* indeed include x. This follows from the fact that replacing any node x_j in the placement by the node x will decrease *both* $\mathcal{S}_c$ and $\mathcal{S}_d$.

Hence using (5.19) and (5.21) we see that for any optimal placement P for I_j we have

$$\mathcal{S}_c(P) = j \sum_{x_j \in P \setminus \{x\}} a_j \geq j\frac{D}{2}. \tag{5.22}$$

Assume that there is a partition U with $|U| = j$ elements. Then, if we choose the placement $P := \{x\} \cup \{ x_i : a_i \in U \}$ for the instance I_j, we get

$$\mathcal{S}_d(P) = jD - j \sum_{x_i \in P \setminus \{x\}} a_i = jD - j\frac{D}{2} = B_j.$$

Thus the placement is feasible. Moreover,

$$\mathcal{S}_c(P) = j \sum_{x_i \in P - \{x\}} a_i = j\frac{D}{2}. \tag{5.23}$$

Hence, by (5.22) this placement is optimal and the bound from equation (5.22) is satisfied as an equality.

Assume conversely that there is an optimal placement for some I_j where the bound from (5.22) is satisfied as an equality, i.e., equation (5.23) holds. If we let $U := \{ i : x_i \in P \setminus \{x\} \}$, we then have $\sum_{i \in I} a_i = \sum_{x_i \in P \setminus \{x\}} a_i = D/2$.

Thus, by running the hypothetical algorithm A on all the instances I_j, $j = 1, \dots, n$ and inspecting the optimum function value $\mathcal{S}_c$ we can decide whether or not the given instance of PARTITION has a solution. □

Given that TREE-(SUM, SUM, p NODES) is NP-hard we investigate the existence of efficient approximation algorithms for it. By combining the parametric search technique from Algorithm 5.2 with the polynomial time algorithm from Theorem 5.29 on page 83, for solving TREE-(SUM, p NODES) (the unconstrained version) optimally, we can obtain approximation algorithm for TREE-(SUM, SUM, p NODES) with performance guarantee $(1 + \varepsilon, 1 + 1/\varepsilon)$. Thus we have the following theorem.

Theorem 5.32 *For any fixed $\varepsilon > 0$ there is a $(1 + \varepsilon, 1 + 1/\varepsilon)$-approximation algorithm for* TREE-(SUM, SUM, p NODES). □

Polynomial Time Solvable Subcases

We outline our idea for the TREE-(SUM, DIA, p NODES) problem. Polynomial time solvability for the other problems follows the same outline and is omitted.

Theorem 5.33 TREE-(SUM, DIA, p NODES) *can be solved in polynomial time.*

Proof: It is easy to see that if two vertices v and w are in a solution, then each vertex on the unique path between v and w can also be added to the solution without violating the diameter constraint.

Using the arguments from the proof of Lemma 5.28, we can conclude that there always exists an optimal solution which is connected; that is, there is an optimal solution which is a subtree of the original tree.

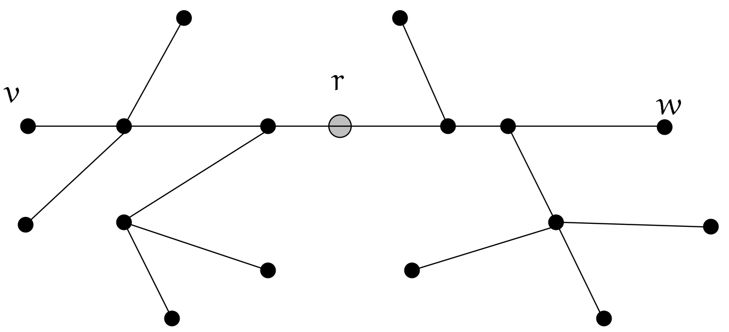


Figure 5.3: Introducing a dummy node r in the tree T.

Consider an optimal solution P^* (which forms a subtree of the original tree T with p nodes). Let $D \leq B$ be the diameter of the tree with respect to the distance function d and let v and w be vertices in P^* which are at a distance of D from each other, i.e., $d(v, w) = D$.

Let $\mathcal{P}_{v,w}$ be the unique path between v and w in the tree T. Place a dummy node r on $\mathcal{P}_{v,w}$ eventually subdividing an edge such that $d(v, r) = d(r, w) = D/2$ (see Figure 5.3). It is easy to see that for all the nodes u in the optimum solution P^* we have $d(r, u) \leq D/2$, since otherwise we could find a node of d-distance greater than D from either v or w. Thus, if we prune the tree T by removing all the vertices x from T such that $d(r, x) > D/2$, then this way we will not discard any vertex from the optimum solution.

Let the resulting tree be $T_{v,w}$ (without the dummy node r). As seen above, $T_{v,w}$ contains all the vertices from the optimum solution. Moreover, for any two nodes $x, y \in T_{v,w}$ we have $d(x, y) \leq D \leq B$. Thus, selecting any subset of the vertices of $T_{v,w}$ will have a d-diameter of at most B.

Moreover, if we solve the (unicriterion) TREE-(SUM, p NODES) problem on $T_{v,w}$ by our polynomial time algorithm from Theorem 5.29 on page 83 with distances given by c, this will give us a placement P such that $\mathcal{S}_c(P) \leq \mathcal{S}_c(P^*)$, since P^* is contained in $T_{v,w}$. Hence $\mathcal{S}_c(P) = \mathcal{S}_c(P^*)$ and we have found an optimum placement.

Algorithm 5.4 Polynomial time algorithm for TREE-(SUM, DIA, p NODES) .

Input: A tree based distance structure τ consisting of a tree T and a weight function d defined on the edges of T, an integer p, and a number B

for all $v, w \in V$ **do**
  Let $d(v, w)$ be the d-distance between v and w
  if $d(v, w) > B$ **then** {the diameter constraint}
    go to the next iteration of the **for**-loop
  end if
  $\mathcal{P}_{v,w} \leftarrow$ the unique path between v and w in the tree T
  Place a dummy node r on $\mathcal{P}_{v,w}$ eventually subdividing an edge such that $d(v, r) = d(r, w) = d(v, w)/2$
  Prune the tree T to obtain a new tree $T_{u,v} = (V', E')$ by removing all vertices x with $d(r, x) > d(v, w)/2$
  if $|V'| < p$ **then**
    go to the next iteration of the **for**-loop
  end if
  Solve the TREE-(SUM, p NODES) instance given by the tree $T_{u,v}$ and distances c *optimally* in polynomial time using the algorithm from Theorem 5.29 on page 83. Let $P(u, v)$ be the placement obtained this way
end for
return the placement $P(u, v)$ with least value $\mathcal{S}_c(P(u, v))$

The algorithm for TREE-(SUM, DIA, p NODES) resulting from the above discussion is outlined in Algorithm 5.4. We repeat the above sketched process for each pair of vertices v and w and choose the placement with the best objective function value. Correctness is immediate from the above arguments. □

Theorem 5.34 TREE-(DIA, DIA, p NODES) *and* TREE-(DIA, SUM, p NODES) *can be solved in polynomial time.* □

5.10 Geometric Versions

In [AI+91], Aggarwal et al. present efficient algorithms for geometric versions of (DIA, p NODES) and (VAR, p NODES) problems. In this case, the nodes are points in space and the distance between a pair of nodes is their Euclidean distance. Their algorithms work for arbitrary dimensions. Here, we present an efficient algorithm for the 1-dimensional (SUM, p NODES) problem. We also point out some difficulties in extending the approach of [AI+91] to obtain a polynomial time algorithm for the (SUM, p NODES) problem for higher dimensions.

In the 1-dimensional version of (SUM, p NODES), the node set V consists of n points (denoted by $x_1, x_2, \ldots, x_n$) on the real line. The distance $c(x_i, x_j)$ between x_j and c_j equals their Euclidean distance, i.e.,

$$c(x_i, x_j) = |x_i - x_j|.$$

We denote this problem by 1D-(SUM, p NODES). As mentioned earlier, minimizing the average distance is equivalent to minimizing the sum of the distances. So, we present our algorithm in terms of the latter objective.

Observe that 1D-(SUM, p NODES) is a special case of TREE-(SUM, p NODES), so basically we could use the $\mathcal{O}(p^2 n)$ time algorithm for trees from Section 5.9 to solve 1D-(SUM, p NODES) in polynomial time. However, we can exploit the geometric structure in 1D-(SUM, p NODES) to obtain a faster $\mathcal{O}(n \log n)$ algorithm.

Our algorithm is based on the following definition and lemma.

Definition 5.35 *Let* $V = \{x_1, x_2, \ldots, x_n\}$ *be a set of* n *points on the real line. A subset* S *of* V *is* ***contiguous*** *if for every pair of points* x *and* z *in* S *with* $x < z$*, there does not exist a point* $y \in V - S$ *such that* $x < y < z$*.*

Lemma 5.36 *Given a 1D-*(SUM, p NODES) *instance* I *consisting of a set* V *of* n *points and an integer* p*, every optimal solution* P^* *to* I *is a set of* p *contiguous points.*

Proof: The proof is by contradiction. Let P^* be an optimal solution to I, and suppose P^* is not contiguous. Then by the above definition, there is a pair of points x and z in P^* and a point $y \in V - P^*$ such that $x < y < z$. Without loss of generality, assume that there is no point of P^* in the open interval (x, z); i.e., x is the largest value in P^* less than y and z is smallest value in P^* greater than y. It can be verified that one of x and z can be replaced by y to obtain a placement better than P^*. □

Theorem 5.37 *The 1D-*(SUM, p NODES) *problem can be solved in* $\mathcal{O}(n \log n)$ *time.*

Proof: By Lemma 5.36, every optimal solution is a contiguous set of p points. Since there are only $n - p + 1$ such sets, it follows that an optimal solution can be found by considering each contiguous set with p points and choosing a set with the minimum value with respect to the $\mathcal{S}_c$ function.

To implement this algorithm in $\mathcal{O}(n \log n)$ time, the points are first sorted into increasing order. Let $\{x_1, x_2, \ldots, x_n\}$ denote the points in increasing order. The

$n - p + 1$ contiguous sets are then considered in order. We now indicate how the value of the objective $\mathcal{S}_c$ for a given set can be obtained in constant time, using the value of the preceding set.

Let $S_i = \{x_{i-p+1}, x_{i-p+2}, \ldots, x_i\}$ denote the contiguous set of p points ending at x_i ($p \le i \le n$). By the definition of the $\mathcal{S}_c$ function,

$$\mathcal{S}_c(S_i) = \sum_{i-p+1 \le j < k \le i} c(x_j, x_k) \quad (p \le i \le n).$$

The goal is to find a set S_r for which $\mathcal{S}_c(S_r)$ is a minimum among the sets S_p, $S_{p+1}, \ldots, S_n$. To compute the values of $\mathcal{S}_c(S_i)$ ($p \le i \le n$) efficiently, we define some auxiliary functions.

Let $\mathcal{SL}_c(S_i)$ and $\mathcal{SR}_c(S_i)$ denote the sum of the distances from each point in S_i to x_{i-p+1} and x_i respectively, i.e., to the leftmost (rightmost) point in S_i. In other words,

$$\mathcal{SL}_c(S_i) = \sum_{j=i-p+2}^{i} c(x_{i-p+1}, x_j) \qquad (p \le i \le n) \tag{5.24}$$

$$\mathcal{SR}_c(S_i) = \sum_{j=i-p+1}^{i-1} c(x_j, x_i) \qquad (p \le i \le n). \tag{5.25}$$

Using these equations, the following recurrences can be derived.

$$\mathcal{SL}_c(S_{i+1}) = \mathcal{SL}_c(S_i) - (p-1)c(x_{i-p+1}, x_{i-p+2}) + c(x_{i-p+2}, x_{i+1}) \tag{5.26}$$

$$\mathcal{SR}_c(S_{i+1}) = \mathcal{SR}_c(S_i) - c(x_{i-p+1}, x_i) + (p-1)c(x_i, x_{i+1}) \tag{5.27}$$

$$\mathcal{S}_c(S_{i+1}) = \mathcal{S}_c(S_i) - \mathcal{SL}_c(S_i) + \mathcal{SR}_c(S_{i+1}) \tag{5.28}$$

The value of $\mathcal{S}_c(S_p)$ can be computed in $\mathcal{O}(p)$ time using the following equation.

$$\mathcal{S}_c(S_p) = \sum_{j=1}^{p-1} j(p-j)c(x_j, x_{j+1}) \tag{5.29}$$

The values of $\mathcal{SL}_c(S_p)$ and $\mathcal{SR}_c(S_p)$ can be computed in $\mathcal{O}(p)$ time using Equations (5.24) and (5.25). Given the values of $\mathcal{S}_c(S_i)$, $\mathcal{SL}_c(S_i)$ and $\mathcal{SR}_c(S_i)$, Equations (5.26), (5.27) and (5.28) show how the values of $\mathcal{S}_c(S_{i+1})$, $\mathcal{SL}_c(S_{i+1})$ and $\mathcal{SR}_c(S_{i+1})$ can be computed in $\mathcal{O}(1)$ time ($p \le i < n$). Thus, a set S_r for which $\mathcal{S}_c(S_r)$ is a minimum among $S_p, S_{p+1}, \ldots, S_n$ can be found in $\mathcal{O}(n)$ time.

Therefore, the running time of the algorithm is dominated by the sorting step which takes $\mathcal{O}(n \log n)$ time. □

In the case of the 2-dimensional version of (SUM, p NODES), denoted by 2D-(SUM, p NODES), it can be shown that every optimal solution P^* is such that every point inside the convex hull of P^* belongs to P^*, thereby extending the notion of contiguity to two dimensional space. Unfortunately, the number of such convex hulls can be exponential.

In [AI+91], an optimal placement with respect to variance $\mathcal{Q}_c$ was found by computing the pth-order Voronoi diagram, and finding a set with the smallest variance from among the sets corresponding to the Voronoi regions. This approach works because it was shown that there is a set P^* with minimum variance and a position z in the plane, such that P^* is the set of the p points nearest to z. However, the following example shows that there exists an instance of 2D-(SUM, p NODES) which has the property that the set with the minimum sum of distances is not the set of p points nearest to *any* position in the plane.

Consider the sets $A = \{a_1, a_2, \ldots, a_{p/2}\}$ with each a_i placed at $(-1, 0)$, $B = \{b_1, b_2, \ldots, b_{p/2}\}$ with each b_i placed at $(1, 0)$, a point x placed at $(0, -1+\varepsilon)$, a point y placed at $(0, 1-\varepsilon)$.

Construct the instance of 2D-(SUM, p NODES) with $V = A \cup B \cup \{x, y\}$, ε a small positive quantity, and the number of facilities to be placed being p. For $p \geq 4$, the optimal placement is $P^* = A \cup B$.

However for each point z on the plane, the p points closest to z contain either x or y. If z is in the first quadrant, then the point y is closer to z than any point in A. Symmetric arguments holds for the other quadrants.

This example is valid, even when A and B are made sets of distinct geometric points clustered around $(-1, 0)$ and $(1, 0)$ respectively. Thus we observe that the sets corresponding to the Voronoi regions of a pth-order Voronoi diagram do not correspond to an optimal solution. Thus, it does not seem possible to directly extend the approach of [AI+91] for finding a placement with minimum variance to the problem of finding a placement that minimizes the sum of the distances.

6

Node Weighted Network Upgrading

6.1 Introduction

Several problems arising in areas such as communication networks and VLSI design can be expressed in the following general form: Enhance the performance of an underlying network by carrying out upgrades at certain nodes or edges of the network [Ber92, Phi93, PS95, KN+96d].

Consider the following scenario which best illustrates the type of problems we investigate. A large communication company is approached by a client with the requirement to interconnect a set of cities housing the client's offices (e.g. banks with high transaction rates between branches). The company has a list of feasible links that it can use to construct a network to connect these cities. Each link has a construction cost associated with it. One of the main concerns of the client is to build a communication network of minimum cost. This is the ubiquitous minimum spanning tree problem (see also Example 3.2). With the advent of optical communication technology, the client would like to upgrade the communication network and has allocated a fixed budget to do so. In communication networks, *upgrading a node* corresponds to installing faster communication equipment at that node. Such an upgrade reduces the communication delay along each edge emanating from the node. Similarly, *upgrading an edge* can be achieved by replacing the old line with a new optical cable. In general, there is a cost for improving each node or edge in the existing network by a unit amount. The goal is to design a strategy to upgrade the network so that the total cost of upgrading is no more than the fixed budget, and the cost of a minimum spanning tree for the upgraded network is the least over all the possible improvements of the network satisfying the budget constraint.

Although substantial work has been done in finding optimal networks (e.g. spanning trees) in graphs, there has been little work on how to *modify* a graph so as to optimize the cost of the network in the resulting graph. In this and the following two chapters, we formulate and study such network upgrade or network improvement problems.

Section 6.2 presents the node based upgrading model which will be used to formulate the problems in this chapter. We discuss the corresponding symmetric problems in Section 6.3. In Section 6.4 we survey related work. Section 6.5 contains our approximation algorithm for the bottleneck upgrading problem. We improve this algorithm in the unit-cost case in Section 6.7 in terms of both performance and running time. In Section 6.8 we give linear time exact algorithms for the upgrading problems in this chapter if restricted to the class of treewidth bounded graphs. Our hardness results are established in Section 6.9.

6.2 Preliminaries and Problem Definition

In the *node based upgrading model* discussed in this chapter we are given the following situation: Let $G = (V, E)$ be a connected undirected graph. For each edge $e = (u, v) \in E$ from the graph G we are given three nonnegative numbers: $d(e)$ represents the *length* or *delay* of the link e. If exactly one of the endpoints u and v is upgraded, the delay of e decreases to $d_m(e)$, the "medium" delay. If both endpoints are upgraded, then the delay falls to $d_l(e)$, the "low" delay. It is assumed that $d_l(e) \leq d_m(e) \leq d(e)$.

Thus, the upgrade of a node v reduces the delay of each edge incident with v. For each node $v \in V$ the value $c(v)$ specifies how expensive it is to upgrade the node. For a subset W of V, the cost of upgrading all the nodes in W, denoted by $c(W)$, is equal to $\sum_{v \in W} c(v)$.

Given a bound D on the bottleneck delay of a spanning tree, we partition the set of edges into four sets according to how many of the endpoints must be upgraded in order to make the delay of an edge fall below the threshold D. An edge of delay at most D is called an *uncritical edge*. An edge e is said to be a *medium critical edge*, if $d(e) > D$ and $d_m(e) \leq D$. Finally, we say that e is a *highly critical edge*, if $d_m(e) > D$ and $d_l(e) \leq D$. Finally, the set of *useless edges*, contains those edges with $d_l(e) > D$. Since node upgrading can not reduce the delay of a useless edge to be below the threshold D, we will assume without loss of generality that the set of useless edges is empty.

The node based upgrading model can be motivated as follows. In communication networks, upgrading a node corresponds to installing faster communication equipment at that node. Such an upgrade reduces the communication delay along each edge emanating from the node. In signal flow networks used in VLSI design, upgrading a node corresponds to replacing a circuit module at the node by a functionally equivalent module containing suitable drivers. Such an upgrade decreases the signal transmission delay along the wires connected to the module.

We are now ready to formulate the problems studied in this chapter.

Definition 6.1 (Node Upgrading Bottleneck Spanning Tree Problems)
Given an edge and node weighted graph $G = (V, E)$ *as above and a bound* D*, the **upgrading bottleneck spanning tree problem**,* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE)*, is to upgrade a set* $S \subseteq V$ *of nodes such that the resulting graph has a spanning tree of bottleneck delay at most* D *and* $c(S)$ *is minimized.*

By (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *we denote the subset of instances where the cost of upgrading each node equals* 1*. Thus, the goal is to find an upgrading set of minimum cardinality.*

We can formulate the above problems more rigorously in our framework of k-criteria problems as follows. For a set $S \subseteq V$ of vertices, denote by d_S the edge weight function resulting from the upgrade of the vertices in S, i.e., for an edge $e = (u, v) \in E$

$$d_S(u, v) := \begin{cases} d(u, v) & \text{if } S \cap \{u, v\} = \varnothing, \\ d_m(u, v) & \text{if } |S \cap \{u, v\}| = 1, \\ d_l(u, v) & \text{if } |S \cap \{u, v\}| = 2. \end{cases} \tag{6.1}$$

Then, (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) can be restated as finding a subgraph from the set

$$\left\{ (T, S) : \begin{array}{l} T \text{ is a spanning tree of } G \text{ and} \\ S \subseteq V \text{ is a subset of the vertices such that } \max_{e \in T} d_S(e) \leq D \end{array} \right\}$$

such that $c(S)$ is minimized.

6.3 Symmetric Problems

The above problems are formulated by specifying a bound on the bottleneck delay after the upgrade, while the upgrading cost is to be minimized. It is also meaningful to consider the corresponding symmetric problems, (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) and (BOTTLENECK, UPGRADING SIZE, SPANNING TREE), where we are given a bound on the upgrading cost and want to obtain the best possible bottleneck delay while staying within our budget restrictions.

The following lemma shows that if we have a good approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE), we can convert it into a good approximation algorithm for the symmetric problem (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) with only a minor additional computational effort. The same result can be established for the unit cost variants (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) and (BOTTLENECK, UPGRADING SIZE, SPANNING TREE).

We will use this result and formulate our approximation algorithms only for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) and (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) respectively, which will be more convenient.

Lemma 6.2 *Suppose that A is a bicriteria approximation algorithm for* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *with a performance of* (α, β)*. Then, one can construct an approximation algorithm for* (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) *with performance of* (β, α) *by using* $\mathcal{O}(\log |E|) \subseteq \mathcal{O}(\log |V|)$ *calls to A, plus an overhead of* $\mathcal{O}(|E| \log |E|)$ *elementary operations, where* $G = (V, E)$ *is the graph given in the input.*

Proof: Suppose we are given an instance of (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) specified by a graph $G = (V, E)$, the cost function c, the delay function d on the edges and the bound B on the node upgrading cost. Let OPT denote the optimum bottleneck delay after upgrading a vertex set of cost at most B. Observe that OPT $\in \mathcal{M}$, where

$$\mathcal{M} = \{\, d(e), d_m(e), d_l(e) : e \in E \,\}. \tag{6.2}$$

Observe that $|\mathcal{M}| \in \mathcal{O}(|E|)$. For a value $D \in \mathcal{M}$ let I_D be the instance of (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) given by the weighted graph G and the bound D on the bottleneck delay of the tree: the goal for I_D is to find a minimum cost upgrading set such that the modified network contains a bottleneck tree of bottleneck delay at most D.

We sort the elements of $\mathcal{M}$ in time $\mathcal{O}(|E| \log |E|)$ and then use a binary search to find the minimum value $D \in \mathcal{M}$ such that algorithm A applied to the instance I_D of (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) outputs an upgrading set of cost at most αB. To show that this binary search works and terminates with a value $D \leq$ OPT, it suffices to show (cf. Lemma 3.15) that A can be viewed as a β-relaxed procedure with test parameter D.

To this end, imagine that we call the outcome of A applied to I_D a COF if the cost of the upgrading set found by A is greater than αB or A returns the information that node upgrading can not reduce the bottleneck delay to a value of at most D. Since by the performance guarantee of A the bound of D on the bottleneck delay of the tree is violated by a factor of at most β, the only thing that remains to show is that, if the outcome is a COF, then in fact OPT $> D$. To see this, consider the optimal upgrading set S^* for the instance of (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE). This forms a feasible solution for I_D for $D \geq$ OPT of cost at most B. Thus, the minimum upgrading cost for the instance I_D is also at most B and A will have no choice but returning an upgrading set of cost at most αB resulting in a bottleneck tree of delay no more than βD. Consequently, by our convention the outcome of A as applied to I_D will not be a COF.

The above arguments shows that after $\mathcal{O}(\log |E|) \subseteq \mathcal{O}(\log |V|)$ calls to A we have a value $D \in \mathcal{M}$ at hand satisfying $D \leq \text{OPT}$. Moreover, we are also given a vertex set W of cost $c(W) \leq \alpha B$ such that upgrading the vertices in W makes the graph G contain a bottleneck spanning tree of bottleneck delay at most αD. This proves the claim of the lemma. $\square$

We have seen above that an (α, β)-approximation algorithm for the (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) problem can be used to design a (β, α)-approximation algorithm for the symmetric problem (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE). The converse is also true. In this case we use the bicriteria algorithm to search for the optimal upgrading cost. This is stated and proved in the following lemma.

Lemma 6.3 *Suppose that A is a bicriteria approximation algorithm for* (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) *with a performance of* (α, β). *Then, there is a* (β, α)*-approximation algorithm for* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE).

Proof: Suppose that we are given an instance of (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE). Assume without loss of generality that all the costs $c(v)$, $v \in V$ are integral.[1] Then, the optimal upgrading cost OPT to obtain a bottleneck spanning tree of delay at most D is an integer satisfying $0 \leq \text{OPT} \leq K := \sum_{v \in V} c(v)$. We perform a binary search on the integers in the interval $[0, K]$ and find the minimum value B such that the (α, β)-approximation algorithm for (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) finds an upgrading set of cost at most βB resulting in a bottleneck spanning tree of bottleneck delay at most αD. Observe that for each value in $[0, K]$ the empty set is a feasible (but possible horribly bad) solution to the corresponding instance of (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE), where we have the bound K bound on the upgrading cost. Thus, for each B, the approximation algorithm must find an upgrading set of cost at most βB resulting in a bottleneck tree of delay at most α times the optimum delay. It follows readily that the value B found during the search satisfies $B \leq \text{OPT}$. By construction, the set returned for B will give us a bottleneck spanning tree of delay at most αB. $\square$

The methods sketched in the last two lemmas allow us to convert approximation algorithms for one of the problems to a algorithm for the symmetric

[1] We can enforce this property by scaling all the node costs by a common denominator.

problem without loss in the performance. The conversion of an algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) has the nice property that the running time increases only by a factor of $\mathcal{O}(\log n)$, while the other way round we get a factor of $\mathcal{O}(\log K)$, where $K = \sum_{v \in V} c(v)$.

6.4 Related Work

The node upgrading model used in this chapter generalizes a model which was introduced in a recent paper by Paik and Sahni [PS95]. In the model from [PS95], we are given a graph $G = (V, E)$ and a nonnegative delay function d on the edges of G. When a node v is upgraded, the delay of each edge incident on v decreases by a fixed factor ρ, where $0 < \rho < 1$. Thus, if $e = (v, u)$ is an edge, its delay after upgrading exactly one of v and u is $\rho d(e)$; the delay of e falls to $\rho^2 d(e)$, if both v and u are upgraded. It is easy to see that this model is a special case of the model introduced in this chapter, namely when $d_m(e) = \rho d(e)$ and $d_l(e) = \rho^2 d(e)$ for each edge e. Paik and Sahni also considered different problems. In Section 6.10 we give a simple approximation algorithm for one of their problems, called LINKDELAY, with a performance of $(2, 1)$.

In Chapter 7 and Chapter 8 we will consider network improvement problems under a different model where there are cost functions associated with improving *edge* weights.

Frederickson and Solis-Oba [FSO96] considered the problem of increasing the weight of the minimum spanning tree in a graph subject to a budget constraint where the cost functions are assumed to be linear in the weight increase. In contrast to the work presented here and in the following two chapters they showed that their problem is solvable in strongly polynomial time. Berman [Ber92] considers the problem of upgrading edges in a given tree to minimize its shortest path tree weight and shows that the problem can be solved in polynomial time by a greedy algorithm. Phillips [Phi93] studies the problem of finding an optimal strategy for reducing the capacity of a network. The goal in this problem is to reduce the edge lengths in the network under a budget constraint such that the minimum cut in the resulting network is minimized. This problem is NP-hard and Phillips gives an FPTAS on planar graphs.

6.5 An Algorithm for Bottleneck Upgrading

In this section, we present our approximation algorithm for (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE). This algorithm provides a performance guarantee of $(2 \ln n, 1)$ on a graph $G = (V, E)$ with $n := |V|$ nodes. In Section 6.9 we will counterbalance this approximation result with a hardness result which shows that, unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log \log n)})$, this performance is essentially the best possible.

Overview

We first give a brief overview of our algorithm. The algorithm maintains a set S of nodes, a set F of edges and a set $\mathcal{C}$ of clusters which partition the vertex set V of the given graph G. The set $\mathcal{C}$ of clusters is initialized to be the set of connected components of the bottleneck graph Bottleneck(G, d, D), containing only those edges e which have a delay $d(e)$ of at most D. The set S of upgrading nodes is initially empty.

The algorithm iteratively merges clusters until only one cluster remains. To this end, in each iteration it determines a node v of minimum *quotient cost*. The quotient cost of a node v is the ratio whose numerator is the cost of v plus the costs of some nodes adjacent to v in different clusters via highly critical edges, and whose denominator is the number of clusters which have nodes adjacent to v. A precise definition of the quotient cost appears in Equation (6.3) in Algorithm 6.1. This quotient cost measures the "average upgrading cost" of v and the vertices that are adjacent to v through highly critical edges. The algorithm then adds v and the nodes mentioned above to the solution set S and merges the corresponding clusters.

The algorithm is shown in Algorithm 6.1. Step 7 can be implemented in polynomial time as an argument of Klein and Ravi in [KR93] shows: For each node v, define the *quotient cost* of v to be the minimum value of the ratio in Equation (6.3) in Step 7 achieved by this node. To find the quotient cost of v, we can order the components in $\mathcal{C}$ as $C_1, C_2, C_3, \dots$ in nondecreasing order of $c(v, C_j)$ (where without loss of generality $v \in C_1$). In computing the quotient cost of v, it is sufficient to consider subsets of $\mathcal{C}$ of the form $\{C_1, C_2, \dots, C_j\}$, where $2 \leq j \leq q$. Thus, the quotient cost for a given vertex can be computed in polynomial time; by computing the quotient cost for each vertex, we can determine the minimum quotient cost, and thus carry out Step 7 in polynomial time. We will elaborate on the total running time of our algorithm and on an efficient implementation starting on page 109.

It is easy to see that the set S output by Algorithm 6.1 is indeed a valid upgrading set, since all the edges added to F in Step 10 will be of delay at most D after upgrading the nodes in S.

In the sequel, we use S^* to denote an optimal upgrading set,i.e., an upgrading set of minimal cost $\text{OPT} := c(S^*)$. We now proceed to prove the following theorem which indicates the performance guarantee provided by the algorithm.

Algorithm 6.1 Approximation algorithm for node upgrading.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, a node weight function c, and a number D

1 $G' \leftarrow \text{Bottleneck}(G, d, D)$
2 $C_1, \dots, C_q \leftarrow$ connected components of G'
3 $S \leftarrow \varnothing$
4 $F \leftarrow$ set of edges of G'
5 **while** $G' = (V, F)$ has more than one connected component **do**
6 $\quad$ {Assume that $\mathcal{C} = \{C_1 \dots, C_q\}$ is the set of components}
7 $\quad$ Find a node $v \in V$ in the graph G minimizing the ratio

$$\min_{2 \leq r \leq q} \min_{\{C_1, \dots, C_r\} \subseteq \mathcal{C}} \frac{c(v) + \sum_{j=1}^{r} c(v, C_j)}{r}. \tag{6.3}$$

$\quad$ Here, the cost $c(v, C_j)$ is defined in the following way: If $v \in C_j$ or v is adjacent to a node in C_j via a medium critical edge, then $c(v, C_j) := 0$. If all the edges from v to C_j are highly critical, then $c(v, C_j)$ is defined to be the minimum cost of a vertex in C_j adjacent to v. If there is no edge between v and any node in C_j, then $c(v, C_j) := +\infty$.
8 $\quad$ Let v be the node and $C_1, \dots, C_r$ be the components in $\mathcal{C}$ chosen in Step 7 above, where w.l.o.g. $v \in C_1$.
9 $\quad$ Let $e_2, \dots, e_r$ be a set of edges in G connecting v to $C_2, \dots, C_r$, respectively.
10 $\quad$ $F \leftarrow F \cup \{e_2, \dots, e_r\}$ {Merge $C_1, C_2, \dots, C_r$ into one component}
11 $\quad$ $S \leftarrow S \cup \{v\}$ {Add center to upgrading set}
12 $\quad$ **for** $i \leftarrow 2, \dots, r$ **do**
13 $\quad\quad$ **if** $e_i = (v, v_i)$ is highly critical **then**
14 $\quad\quad\quad$ $S \leftarrow S \cup \{v_i\}$ {Add finger to upgrading set}
15 $\quad\quad$ **end if**
16 $\quad$ **end for**
$\quad$ {Note that the total cost of the nodes added to the solution S is exactly $c(v) + \sum_{j=1}^{r} c(v, C_j)$.}
17 $\quad$ $C_1, \dots, C_{q'} \leftarrow$ connected components of $G' = (V, F)$
18 **end while**
19 **return** S

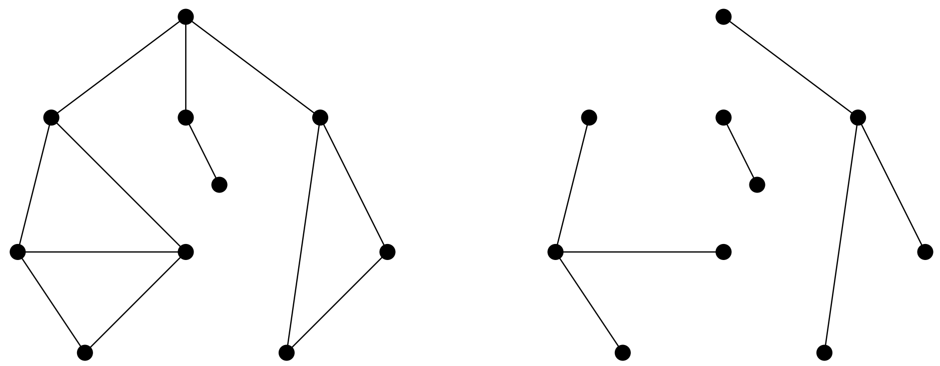
Theorem 6.4 *Algorithm 6.1 as applied to* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *has a performance of* $(2 \ln n, 1)$*, where* n *denotes the number of nodes in the graph.*

Our proof of Theorem 6.4 relies on several lemmas, which are presented below. We estimate the cost of the nodes added by the heuristic in each iteration by first establishing an averaging lemma and then using a potential function argument. The notion of a *claw decomposition* which is introduced in the next subsection will be a crucial tool in the analysis. We will also need the following lemma, which can be proved by elementary techniques from mathematical analysis.

Lemma 6.5 *For* $\tau < 1$ *the inequality* $\ln(1 - \tau) \leq -\tau$ *holds true.* □

Claw Decompositions

We employ the notion of *claw decompositions* in showing that the quotient cost of each node chosen in Step 7 is small compared to the optimal solution.



(a) Graph $G = (V, E)$ (b) Claw decomposition of V in G

Figure 6.1: Example of a claw decomposition.

Definition 6.6 (Claw, claw decomposition)
A graph $G = (V, E)$ *is called a* ***claw****, if the edge set* E *is of the form* $E = \{ (v, w) : w \in V \setminus \{v\} \}$ *for some node* $v \in V$*. If there are at most two nodes in the claw then we can choose any of the nodes as its* ***center****. Otherwise, the node with degree greater than one is the unique* ***center****. The vertices in the claw different from the center are said to be the* ***fingers*** *of the claw. A claw with at least two nodes is called a* ***nontrivial claw****.*

Let G *be a graph with node set* V*. A* ***claw decomposition*** *of* V *in* G *is a collection of node-disjoint nontrivial claws, which are all subgraphs of* G *and whose vertices form a partition of* V*.*

Figure 6.1 shows an example of a graph and a corresponding claw decomposition. The following theorem establishes the existence of a claw-decomposition for any connected graph.

Theorem 6.7 *Let* G *be a connected graph with node set* V*, where* $|V| \geq 2$*. Then there is a claw decomposition of* V *in* G*.*

Proof: If $|V| = 2$, then the claim is trivial. Thus, assume that $|V| > 2$ and that the claim has already been established for all connected graphs of at most $|V| - 1$ nodes.

Choose an arbitrary node $v \in V$. Consider the graph $G' = G[V \setminus \{v\}]$ induced by the vertices in $V \setminus \{v\}$. Let $C_1, \ldots, C_k$ be the connected components of G', enumerated in such a way that $|C_i| = 1$ for $1 \leq i \leq i'$ and $|C_i| \geq 2$ for $i' + 1 \leq i \leq k$. Then the single node components $C_1, \ldots, C_{i'}$ consist of neighbors of v in G.

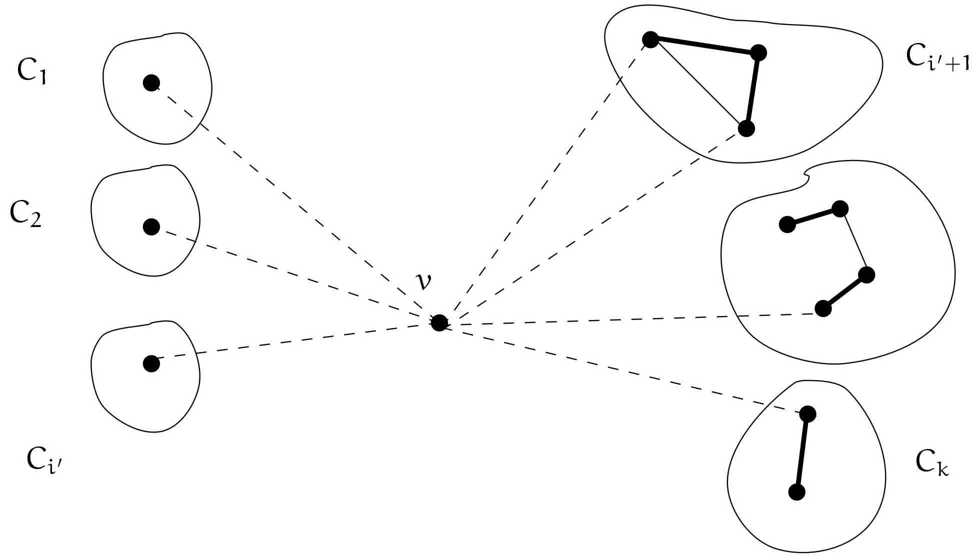


Figure 6.2: Components resulting from the removal of v.

By induction hypothesis, there are claw decompositions for the components $C_{i'+1}, \ldots, C_k$ which consist of at least two vertices each. Figure 6.2 illustrates the components in $G[V \setminus \{v\}]$. The claw decompositions in $C_{i'+1}, \ldots, C_k$ are indicated by the thick edges.

We now distinguish between two cases, $i' > 0$ and $i' = 0$. In the first case, there is at least one component in $G[V \setminus \{v\}]$ that consists of a single node. Then, the claws from the decompositions of $C_{i'+1}, \dots, C_k$ together with the claw with center v and fingers consisting of the nodes from the single node components $C_1, \dots, C_{i'}$ form a valid claw decomposition of V in G.

If $i' = 0$, meaning that *all* components of G' consist of at least two nodes we choose one component, say C_1, which contains a neighbor u of v in G. The vertex u occurs in one claw of the claw decomposition of C_1. If u is the center of that claw, we can enlarge the claw by adding v to it as a finger (see Figure 6.3). This will give us a valid claw decomposition of V in G.

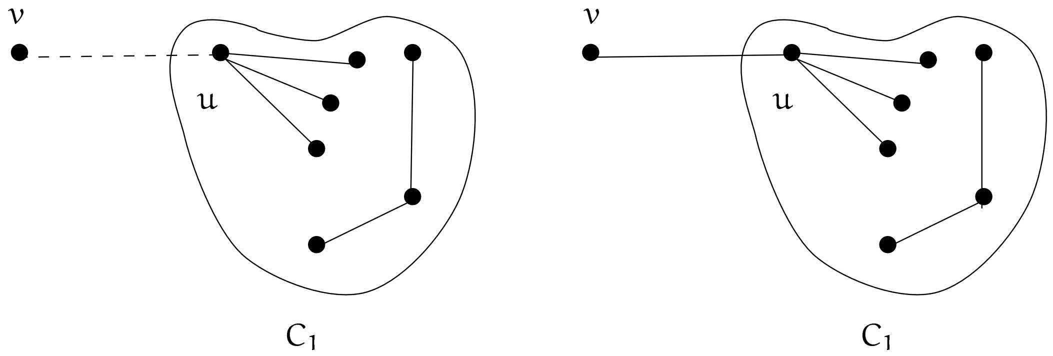


Figure 6.3: The neighbor u of v is the center of a claw in C_1.

If u is not the center of its claw, then we remove u from that claw and add the two vertex claw consisting of u and v to our set of claws (see Figure 6.4). Again, we obtain a valid claw decomposition of V in G. □

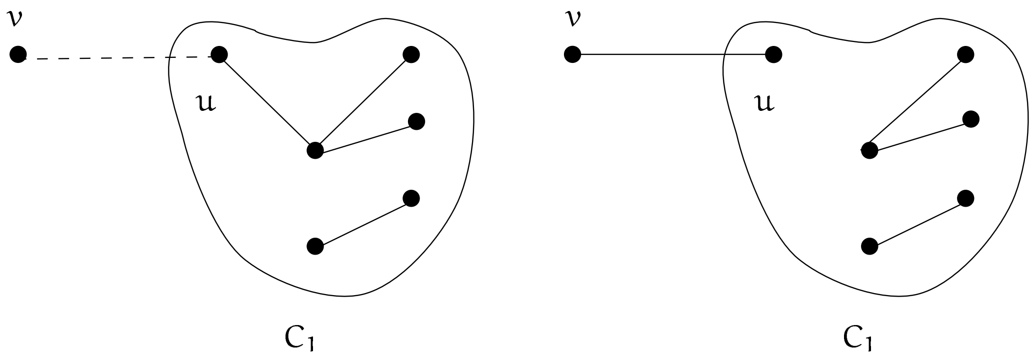


Figure 6.4: The neighbor u of v is a finger in a claw.

An Averaging Lemma

Lemma 6.8 *Let v be a node chosen in Step 7 of Algorithm 6.1 and let C denote the total cost of the nodes added to the solution set S in this iteration. Let there be q clusters before v is chosen and assume that in this iteration r clusters are merged. Then*

$$\frac{C}{r} \leq \frac{\mathrm{OPT}}{q}.$$

Proof: Let T^* be an optimal tree with the nodes S^* be the upgraded nodes. Let $\mathrm{OPT} := c(S^*)$ be the cost of the optimal solution. Let $\mathcal{C} = C_1, \ldots, C_q$ be the clusters when the node v was chosen and let $T^*(v)$ be the graph obtained from T^* by contracting each C_j to a supernode. $T^*(v)$ is connected and contains all supernodes. We then remove edges (if necessary) from $T^*(v)$ so as to make it a spanning tree. Note that all the edges in this tree are either medium or highly critical.

Let $A \subseteq S^*$ be the set of nodes in the optimal solution that are adjacent to another cluster in $T^*(v)$. Clearly, the cost of these nodes is no more than OPT. Take a claw decomposition of $T^*(v)$. We now obtain a set of claws in the graph G itself in the following way: Initialize E' to be the empty set. For each claw in the decomposition with center C_1' and fingers $C_2', \ldots, C_l'$ we do the following: For each edge (C_1', C_j') the optimal tree T^* must have contained an edge (u, w) with $u \in C_1'$ and $w \in C_i'$. Notice that since this edge was either medium or highly critical, at least one of the vertices u and w must belong to $A \subseteq S^*$. We add (u, w) to E'.

It is easy to see that the subgraph of G induced by the edges in E' consists of disjoint nontrivial claws. Also, all edges in the claws were medium or highly critical and the total number of nodes in the claws is at least q. We need one more useful observation: If a claw center is not contained in A, then *all* the fingers of the claw must be contained in A, since the edges in the claw were medium or highly critical.

Let A_c be the set of nodes from A acting as centers in the just generated claws. Let A^{med} denote the fingers of the claws contained in A which are connected to their claw center via a medium critical edge, whereas A^{high} stands for the set of fingers adjacent to the center via a highly critical edge and also contained in A. For each claw with exactly two nodes we designate an arbitrary one of the nodes to be the center. Then by construction, A_c, A^{med} and A^{high} are disjoint. Therefore,

$$\mathrm{OPT} \geq \sum_{u \in A_c \cup A^{\mathrm{high}}} c(u) + \sum_{u \in A^{\mathrm{med}}} c(u). \tag{6.4}$$

For a node $u \in A_c$, let N_u denote the number of vertices in the claw centered at u. We have seen that if a center is not in A, then *all* the fingers belong to the optimal solution. Clearly, this can only happen, if the claw centered at u does not contain a highly critical edge. Thus, we can estimate the total number of nodes in the claws from above by summing up the cardinalities of the claws with centers in A and for all other claws adding twice the number of fingers. Hence

$$\sum_{u \in A_c} N_u + 2|A^{\mathrm{med}}| \geq |\{ w : w \text{ belongs to some claw} \}| \geq q, \tag{6.5}$$

since the total number of nodes in the claws is at least q.

We now estimate the first sum in (6.4). If $u \in A_c$, then the quotient cost of u is at most the cost of u plus the cost of the fingers in the claw that are in A^{high} divided by the total number of nodes in the claw. This in turn is at least C/r by the choice of the algorithm in Step 7. By summing up over all those centers, this leads to

$$\sum_{u \in A_c \cup A^{\mathrm{high}}} c(u) \geq \frac{C}{r} \sum_{u \in A_c} N_u. \tag{6.6}$$

Now, for a node u in A^{med}, its quotient cost is at most $c(u)/2$, which again is at least C/r. Thus

$$\sum_{u \in A^{\mathrm{med}}} c(u) \geq \sum_{u \in A^{\mathrm{med}}} 2\frac{C}{r} = 2|A^{\mathrm{med}}| \cdot \frac{C}{r}. \tag{6.7}$$

Using (6.6) and (6.7) in (6.4) yields

$$\begin{aligned} \mathrm{OPT} &\geq \sum_{u \in A_c \cup A^{\mathrm{high}}} c(u) + \sum_{u \in A^{\mathrm{med}}} c(u) \\ &\geq \frac{C}{r} \left(\sum_{u \in A_c} N_u + 2|A^{\mathrm{med}}| \right) \\ &\overset{(6.5)}{\geq} \frac{C}{r} \cdot q. \end{aligned}$$

This proves the claim. □

A Potential Function Argument

We are now ready to complete the proof of the performance stated in Theorem 6.4. Assume that the algorithm uses f iterations of the loop and denote by $v_1, \dots v_f$ the vertices chosen in Step 7 of the algorithm.

Let ϕ_j denote the number of clusters *after* choosing vertex v_j in this iteration. Thus, for instance, $\phi_0 = t$, the number of components at the beginning of the whole algorithm and $\phi_f = 1$, since we end up with one cluster. Let the number of clusters merged using vertex v_j be r_j and the total cost of the vertices added in that iteration be c_j. Then we have

$$\phi_j = \phi_{j-1} - (r_j - 1). \tag{6.8}$$

Notice that, since $r_j \geq 2$, we have $r_j - 1 \geq \frac{1}{2} r_j$. Using this inequality in (6.8) we obtain

$$\phi_j \leq \phi_{j-1} - \frac{1}{2} r_j. \tag{6.9}$$

Observe that $\phi_j \geq 2$ for $j = 0, \ldots, f-1$, since the algorithm does not stop before the f-th iteration. Notice also that $\phi_f = 1$. Then by Lemma 6.8, we have

$$r_j \geq \frac{c_j \phi_{j-1}}{\mathrm{OPT}} \tag{6.10}$$

for all $0 \leq j \leq f$. We now use an analysis technique due to Leighton and Rao [LR88] to complete the proof. Substituting equation (6.10) into (6.9) yields

$$\phi_j \leq \phi_{j-1} - \frac{1}{2} \cdot \frac{c_j \phi_{j-1}}{\mathrm{OPT}} = \phi_{j-1} \cdot \left(1 - \frac{c_j}{2 \cdot \mathrm{OPT}}\right). \tag{6.11}$$

Using the recurrence (6.11), we obtain

$$\phi_f \leq \phi_0 \prod_{j=1}^{f} \left(1 - \frac{c_j}{2 \cdot \mathrm{OPT}}\right). \tag{6.12}$$

Taking natural logarithms on both sides and simplifying using the estimate $\ln(1-\tau) \leq -\tau$ (Lemma 6.5), we obtain

$$2 \cdot \mathrm{OPT} \cdot \ln\left(\frac{\phi_0}{\phi_f}\right) \geq \sum_{j=1}^{f} c_j. \tag{6.13}$$

Notice that by Lemma 6.8 we have

$$c_j \leq \mathrm{OPT} \cdot \frac{r_j}{\phi_{j-1}} \leq \mathrm{OPT} < 2 \cdot \mathrm{OPT},$$

and so the logarithms of all the terms in the product of (6.12) are well defined. Note also that $\phi_0 \leq n := |V|$ and $\phi_f = 1$ and hence from (6.13) we get

$$\sum_{j=1}^{f} c_j \leq 2 \cdot \mathrm{OPT} \cdot \ln n. \tag{6.14}$$

Notice that the total cost of the nodes chosen by the algorithm is exactly the sum $\sum_{j=1}^{f} c_j$. This completes the proof of Theorem 6.4. □

Running Time

We now sketch how to implement our Algorithm 6.1 efficiently. Let $G = (V, E)$ have $n := |V|$ nodes and $m := |E|$ edges. Computing the bottleneck graph in Step 1 can be done in $\mathcal{O}(m)$ time. The computation of the connected components in Step 2 can be accomplished by classical graph theoretic algorithms in time $\mathcal{O}(n + m)$ (cf. [Nol75, CLR90]).

The main computational effort lies in the **while**-loop. The key ingredient for a good running time is to make a fast computation of the quotient cost of each node possible. We are going to realize this task with the help of a fast *disjoint-set data structure* [CLR90]. Such a data structure maintains a collection of disjoint dynamic sets. Each set is identified by a representative which is some member of the set. The following operations are available.

MAKE-SET(x) creates a new set whose only element (and thus representative) is x.

UNION(x, y) unites the dynamic sets that contain x and y.

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

For details we refer the reader to a standard book on algorithms and data structures such as [CLR90]. There, one can also find the proof of the following theorem.

Theorem 6.9 *A sequence of* t MAKE-SET, UNION *and* FIND-SET *operations,* s *of which are* MAKE-SET *operations, can be performed in worst-case time* $\mathcal{O}(t\alpha(t, s))$*, where* α *is the inverse of Ackerman's function.* □

We now outline an efficient implementation of our Algorithm 6.1 using Theorem 6.9. The idea is to maintain for each node $v \in V$ a sorted list $L(v)$ of those neighbors of v which are in different clusters than v. The information, which node is contained in which cluster is updated with the help of the fast data structure for disjoint sets. Let FIND-SET(v) denote the cluster containing vertex v. Then, we will sort each list $L(v) = (v_1, \ldots, v_p)$ in such a way that $c(v, \text{FIND-SET}(v_1)) \leq \cdots \leq c(v, \text{FIND-SET}(v_p))$, where $c(v, \text{FIND-SET}(u))$ is the cost defined in Step 7 of Algorithm 6.1.

Algorithm 6.2 shows how to initialize the necessary information. For reasons of conformity with the notations used in our Algorithm 6.1 we add v to the head of the list $L(v)$. After initializing the list $L(v)$ with the neighbors of the respective vertex $v \in V$ and sorting this list with respect to weight $\omega(v, .)$, the

Algorithm 6.2 Initialization of the algorithm for node upgrading.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, a node weight function c, and a number D

```
for all v ∈ V do
  MAKE-SET(v)
end for
for all e = (v, u) ∈ E do
  if d(e) ≤ D then
    UNION(v, u)
  end if
end for
for all v ∈ V do
  ω(v, v) ← 0
  L(v) ← List of neighbors of v in G
  for all u ∈ L(v) do
    if (v, u) is highly critical then
      ω(v, u) ← c(u)
    else
      ω(v, u) ← 0
    end if
  end for
  Sort L(v) according to the weight function ω(v, .).
end for
Add v to the head of L(v).
Call Algorithm 6.3 to remove useless vertices from the lists L.
return L, ω
```

initialization Algorithm 6.2 calls Algorithm 6.3 to remove "useless vertices" from the lists $L(v)$. Here, a vertex $u \in L(v)$ is called "useless", if it satisfies at least one of the following conditions:

1. The vertex u is contained in the same cluster as v. In this case we can discard u, since we only want to keep track of the neighbors of v that are in *different* clusters as v.

2. The edge (v, u) is highly critical and there is another vertex $u' \in L(v)$ such that u is in the same cluster as u' and either (v, u') is only medium critical or it is also highly critical but $c(u') < c(u)$. Again, we can discard u, since it will be of no use with regard to the quotient cost of v.

Algorithm 6.3 processes each list $L(v)$ and removes the useless vertices. Since

Algorithm 6.3 Algorithm to remove useless vertices from the lists L.

Input: A graph $G = (V, E)$, for each vertex $v \in V$ a list $L(v)$ of neighbors of v in G

```
for all v ∈ V do
  for all u ∈ L(v) do
    cluster(FIND-SET(u)) ← No
  end for
  for all u ∈ L(v) do
    if cluster(FIND-SET(u)) = Yes then
      Delete u from L(v)
    else
      cluster(FIND-SET(u)) ← Yes
    end if
  end for
end for
```

the list $L(v)$ is already sorted, a single pass over $L(v)$ suffices to accomplish this task. At the end of Algorithm 6.2 the lists $L(v)$ have the following desirable properties:

1. For each cluster C adjacent to v there is a vertex $w \in L(v)$ such that $w \in C$.
2. For each vertex u in the list $L(v)$ we have $\omega(v, u) = c(v, \text{FIND-SET}(u))$ where the cost c is defined in Step 7 of Algorithm 6.1.
3. The list $L(v)$ is sorted with respect to $\omega(v, .)$.

With the information in $L(v)$ and $\omega(v, .)$, it is now easy to compute the quotient cost of the node v. We only need to process the list $L(v) = (v = v_1, \dots, v_p)$ from the beginning to the end and choose r such that

$$\frac{c(v) + \sum_{j=1}^{r} \omega(v, v_j)}{r}$$

is minimized (cf. the comments about the quotient cost on page 101). This is depicted in Algorithm 6.4 which finds and returns a vertex v of minimum quotient cost.

Finally, we need to update our information about the clusters and in the lists $L(v)$ $(v \in V)$. Let v' be a node with minimum quotient cost and let $v_2, \dots, v_r$ be the fingers in the corresponding claw. We must merge the clusters of v' and $v_2, \dots, v_r$ into one cluster. This can be done with the UNION operation

Algorithm 6.4 Algorithm for computing a node with minimum quotient cost.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, a node weight function c, a number D, for each vertex v a list of neighbors $L(v)$, and a weighing function $\omega(v, .)$

for all $v \in V$ **do**
  {Assume that $L(v) = (v = v_1, v_2, \ldots, v_p)$}
  Choose $r \in \{1, \ldots, p\}$ satisfying

$$\frac{c(v) + \sum_{j=1}^{r} \omega(v, v_i)}{r} = \min_{1 \leq r' \leq q} \frac{c(v) + \sum_{j=1}^{r} \omega(v, v_j)}{r}.$$

  quotient$(v) \leftarrow \dfrac{c(v) + \sum_{j=1}^{r} \omega(v, v_j)}{r}$
end for
return a node $v \in V$ with minimum quotient(v) and the vertices $v_2, \ldots, v_r$ in the fingers of the corresponding claw

provided by the disjoint set data structure. Finally, we need to clean up all the lists $L(v)$ $(v \in V)$ by removing all unnecessary vertices. This can again be accomplished by a call to Algorithm 6.3. Algorithm 6.5 shows the total merging process. After the run of Algorithm 6.5 the lists $L(v)$ will again possess the desirable properties enabling us to compute the quotient cost of each vertex fast.

Algorithm 6.5 Algorithm to merge clusters after a node upgrade.

Input: A vertex v' of minimum quotient cost, the fingers $v_2, \ldots, v_r$ of the corresponding claw, and for each vertex $v \in V$ a list $L(v)$ of neighbors

for $i \leftarrow 2, \ldots, r$ **do**
  UNION(v', v_i)
end for
Call Algorithm 6.3 to remove useless vertices from the lists L.
return L, ω

We are ready to estimate the total time of our algorithm. In the initialization (Algorithm 6.2) we need n MAKE-SET and $\mathcal{O}(m)$ UNION operations. Steps 9 to 20 need $\mathcal{O}(m \log n)$ time. For the removal of useless vertices in Algorithm 6.3 we need $\mathcal{O}(m)$ FIND-SET plus $\mathcal{O}(m)$ elementary operations. This results in a total time of $\mathcal{O}(m \log n + m\alpha(m, n)) \subseteq \mathcal{O}(m \log n)$ for the initialization.

Algorithms 6.4 and 6.5 are used during the run of the **while**-loop of Algorithm 6.1, which is executed $\mathcal{O}(n)$ times.

As far as computing the quotient costs in Algorithm 6.4 is concerned, observe that one iteration of the **for**-loop needs $\mathcal{O}(\deg_G(v))$ time, where $\deg_G(v)$ is the degree of v in G. Since $\sum_{v\in V} \deg_G(v) = 2m$, this implies that Algorithm 6.4 runs in time $\mathcal{O}(m)$.

The merging of clusters and update of information in Algorithm 6.5 needs $\mathcal{O}(n)$ UNION operations plus the call to Algorithm 6.3 to remove the useless vertices, which takes $\mathcal{O}(m\alpha(m,n))$ time.

The above arguments show that one iteration of the **while**-loop needs time $\mathcal{O}(m\alpha(m,n))$. Since the number of iterations is $\mathcal{O}(n)$, this results in an overall time of $\mathcal{O}(nm\alpha(m,n))$, which dominates the time of $\mathcal{O}(m \log n)$ needed in the initialization phase. We summarize our results in the following theorem.

Theorem 6.10 *Algorithm 6.1 as applied to* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *has a performance of* $(2\ln n, 1)$. *It can be implemented to run in time* $\mathcal{O}(nm\alpha(m,n))$, *where* n *denotes the number of nodes,* m *the number of edges in the graph, and* α *is the inverse of Ackerman's function.* □

Using Lemma 6.2 we obtain the following approximation result for the symmetric problem (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE).

Theorem 6.11 *There is an approximation algorithm for* (BOTTLENECK, NODE UPGRADING COST, SPANNING TREE) *with a performance of* $(1, 2\ln n)$. *It can be implemented to run in time* $\mathcal{O}(nm\alpha(m,n)\log n)$. □

6.6 Using the Proof Techniques for the Analysis of Algorithms

The proof of Theorem 6.4 used the averaging Lemma 6.8 and a potential function argument. In this section we will briefly demonstrate the usefulness of this technique for the analysis of approximation algorithms, mainly those of "greedy-type".

To this end we recall the MIN SET COVER problem (cf. page 36). An instance $(Q, \mathcal{F})$ of MIN SET COVER consists of a finite set Q of ground elements and a family $\mathcal{F}$ of subsets of Q such that each element of Q belongs to at least one subset in $\mathcal{F}$. The problem consists of finding a minimum size subset $\mathcal{C} \subseteq \mathcal{F}$

such that the union of the sets in $\mathcal{C}$ cover all elements in Q, i.e.,

$$\bigcup_{S \in \mathcal{C}} S = Q. \tag{6.15}$$

Any subset $\mathcal{C}$ of $\mathcal{F}$ satisfying (6.15) is said to be a *set cover*. In the decision version of MIN SET COVER, denoted by SET COVER, one is additionally given an integer k, and the question is whether one can find a set cover of size at most k. It is well known that SET COVER is NP-complete [GJ79, Problem SP5]. Consequently, MIN SET COVER is NP-hard. Moreover, by the results of Feige [Fei96] MIN SET COVER is NP-hard to approximate within a factor of $\beta < \ln |Q|$ (see Theorem 3.12 on page 36).

A classical approximation algorithm for MIN SET COVER is the greedy method shown in Algorithm 6.6 (cf. [CLR90]). The approach used is a natural one: The method works by picking, at each stage, a set S that covers the most remaining uncovered elements.

Algorithm 6.6 Greedy approximation algorithm for MIN SET COVER.

Input: A set Q of ground elements, a family $\mathcal{F}$ of subsets of Q

1 $U \leftarrow Q$ {uncovered elements}
2 $\mathcal{C} \leftarrow \emptyset$ {the cover to be constructed}
3 **while** $U \neq \emptyset$ **do**
4   Select a set $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5    $U \leftarrow U \setminus S$
6    $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7 **end while**
8 **return** $\mathcal{C}$

We will now use our techniques from the proof of Theorem 6.4 to show that the performance of Algorithm 6.6 is $1 + \ln |Q|$. To this end, we first establish the following analogon to Lemma 6.8.

Lemma 6.12 *Let S be a set chosen in Step 4. Let the size of U be the set of uncovered elements before S is chosen. Then*

$$\frac{1}{|S \cap U|} \leq \frac{\mathrm{OPT}}{|U|}.$$

Proof: The optimal cover $\mathcal{C}^*$ consists of OPT sets from $\mathcal{F}$. Since their union covers Q and thus in particular U, there must be at least one set $Q' \in \mathcal{C}^*$ such that $|Q' \cap U| \geq |U|/\mathrm{OPT}$. Thus, $1/|Q' \cap U| \leq \mathrm{OPT}/|U|$ and the claim now follows by the choice of S in Step 4 of Algorithm 6.6. □

The rest of the analysis parallels the proof of Theorem 6.4. Suppose that the algorithm uses f iterations and denote by $S_1, \dots, S_f$ the sets chosen by Algorithm 6.6 in Step 4 in this order. Denote by ϕ_j the number of uncovered elements after S_j was added to $\mathcal{C}$ by the algorithm and by r_j the number of nodes from U covered by S_j. Then, $\phi_0 = |Q|$, $\phi_f = 0$ and

$$\phi_j = \phi_{j-1} - r_j, \qquad \text{for } j = 1, \dots, f. \tag{6.16}$$

Notice that by Lemma 6.12 we have that

$$r_j \geq \frac{\phi_{j-1}}{\mathrm{OPT}}, \qquad \text{for } j = 1, \dots, f. \tag{6.17}$$

If we substitute (6.17) into (6.16) we get that

$$\phi_{f-1} \leq \phi_0 \left(1 - \frac{1}{\mathrm{OPT}}\right)^{f-1}.$$

If we take natural logarithms and use again the estimate $\ln(1-\tau) \leq -\tau$ from Lemma 6.5 this yields

$$f - 1 \leq \mathrm{OPT} \cdot \ln\left(\frac{\phi_0}{\phi_{f-1}}\right).$$

Since $\phi_0 = |Q|$ and $\phi_{f-1} \geq 1$ (due to the fact that the algorithm has not stopped after the $(f-1)$th iteration), this shows that

$$f \leq \mathrm{OPT} \cdot \ln|Q| + 1 \leq (1 + \ln|Q|) \cdot \mathrm{OPT}.$$

We summarize the above results in the following theorem.

Theorem 6.13 *Algorithm 6.6 is an approximation algorithm for the* MIN SET COVER *problem with a performance of* $1 + \ln|Q|$. □

The above analysis of the greedy algorithm for the MIN SET COVER problem illustrates that our proof technique is indeed a powerful tool for the analysis of algorithms. It is particularly well applicable to algorithms of greedy type, since for these algorithms usually an analogon to the averaging Lemma 6.8 is easy to establish.

It should be noted that one can show the following stronger result on the performance of Algorithm 6.6 (see e.g. [CLR90, Mot92]):

Theorem 6.14 *Algorithm 6.6 is an approximation algorithm for the* MIN SET COVER *problem with a performance of* $\mathcal{H}(\max\{|S'| : S' \in \mathcal{F}\})$*, where* $\mathcal{H}(n) = \sum_{i=1}^{n} \frac{1}{i}$ *is the* n*th harmonic number. Since* $\mathcal{H}(n) \leq 1 + \ln n$*, the performance provided is* $1 + \ln \max\{|S'| : S' \in \mathcal{F}\}$. □

6.7 Improved Approximation Algorithms for Unit Costs

In this section, we will show how to obtain an approximation algorithm for the case of unit node costs with a performance guarantee of $(5 + 4 \ln \Delta, 1)$, where Δ denotes the maximum degree of the input graph. For $\Delta \in o(\sqrt{n})$, this improves on the performance of our algorithm for the general case. In particular, for the class of bounded-degree graphs we obtain a *constant factor approximation*.

Performance for the Case of No Highly Critical Edges

Recall that in the case of (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) the objective is to find an upgrading set of *minimum cardinality*. We first consider the case, when all edges are either uncritical or medium critical.

Algorithm 6.7 on the next page shows the improved heuristic. We now establish the performance of the algorithm, which is summarized in the following theorem.

Theorem 6.15 *Algorithm 6.7 is a polynomial time approximation algorithm for* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *when all edges are either uncritical or medium critical that, given any such instance of the problem, finds an upgrading set* S *of size*

$$|S| \leq 2(1 + \ln \Delta) \cdot \mathrm{OPT} - 1,$$

where Δ *is the maximum degree in the input graph* G *and* OPT *is the size of an optimal upgrading set. In particular the algorithm has a performance guarantee of* $(2 + 2 \ln \Delta, 1)$.

The proof relies again on several lemmas. Algorithm 6.7 clearly detects the case, when no node needs to be upgraded (Step 3). Thus, for the rest of the analysis we will restrict ourselves to the case, when the optimal solution contains at least one vertex. We use S^* to denote an optimal upgrading set and define $\mathrm{OPT} := |S^*|$. Also, let T^* be a corresponding bottleneck spanning tree, i.e., a tree which has bottleneck delay at most D after upgrading the vertices in S^*.

Lemma 6.16 *Let* $\mathcal{C}^*$ *be a minimum size set cover for the instance* $(Q, \mathcal{F})$ *of the* MIN SET COVER *problem constructed in Step 6 of Algorithm 6.7. Then* $|\mathcal{C}^*| \leq \mathrm{OPT}$.

Algorithm 6.7 Improved alg. for unit costs without highly critical edges.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, and a number D

$G' \leftarrow \text{Bottleneck}(G, d, D)$
$C_1, \ldots, C_q \leftarrow$ connected components of G'
if $q = 1$ **then**
  return $\varnothing$
end if
Construct an instance $(Q, \mathcal{F})$ of MIN SET COVER as follows:
Let the ground elements be $Q := \{C_1, \ldots, C_q\}$. For each $v \in V$ define the set

$$S_v := \{\, C_j : v \in C_j \text{ or } v \text{ is adjacent to } C_j \text{ via a medium critical edge} \,\}$$

and let $\mathcal{F} := \{\, S_v : v \in V \,\}$.
Find a set cover $\mathcal{C} = \{S_{v_1}, \ldots, S_{v_k}\}$ of size at most $1 + \ln \max\{\, |S_v| : S_v \in \mathcal{F} \,\}$ times the minimum size set cover for the instance $(Q, \mathcal{F})$.
$S \leftarrow \{v_1, \ldots, v_k\}$
$S' \leftarrow \varnothing$
for all $v \in S$ **do**
  for all $e \in E$ incident with v **do**
    $d(e) \leftarrow d_m(e)$ {thus the delay of such an edge e will be at most D}
  end for
end for
$G' = (V, E') \leftarrow \text{Bottleneck}(G, d, D)$
$C'_1, \ldots, C'_s \leftarrow$ connected components of G'
Construct an auxiliary graph $\hat{G} = (\{C'_1, \ldots, C'_s\}, \hat{E})$ with $(C'_i, C'_j) \in \hat{G}$ if and only if there are vertices $v \in C'_i$ and $w \in C'_j$ such that $(v, w) \in E$.
$\hat{T} \leftarrow$ spanning tree of $\hat{G}$
for all edges $(C'_i, C'_j) \in \hat{T}$ **do**
  Choose $v \in C'_i$ and $w \in C'_j$ such that $(v, w) \in E$.
  $S' \leftarrow S' \cup \{v\}$
end for
return $S \cup S'$

Proof: We show that the sets S_v with $v \in S^*$ form a set cover for the instance $(Q, \mathcal{F})$ of MIN SET COVER. This will prove the claim of the lemma.

Consider an arbitrary component $C_j \in Q$. We have to show that C_j is contained in the union of the sets $\bigcup_{v \in S^*} S_v$.

Since $G' = \text{Bottleneck}(G, d, D)$ contains more than one connected component, the tree T^* must contain a medium critical edge (u, w) with $u \in C_j$ and $w \notin C_j$. Since after the upgrade of the vertices in S^* the tree T^* has bottleneck delay at most D it follows that either u or w must be contained in S^*.

We have $C_j \in S_w$ (because w is connected to C_j via a medium critical edge) and $C_j \in S_u$ (since u is contained in C_j). Thus, we can conclude that C_j is covered by the sets S_v, $v \in S^*$. □

Lemma 6.17 *Let* S *be the set constructed in Step 8. Then the bottleneck graph* G' *computed in Step 15 has at most* $|S|$ *connected components.*

Proof: We show that for each node v there is a path in G' to at least one node from S. Assume for the sake of a contradiction that this is not the case. Then there is a vertex v in a connected component C'_v of G' where the component G' does not contain any vertex of S. By construction, C'_v must contain one of the connected components, say C, computed in Step 2 *before* the upgrade of the vertices in S. Since the sets S_v corresponding to the vertices $v \in S$ were a feasible set cover, it follows that C must have either contained a vertex from S or must have been adjacent to a vertex from S via a medium critical edge. In both cases after the upgrade this vertex from S is contained in the same connected component as the vertices in C. But this means that C'_v must contain at least one vertex from S yielding a contradiction. ↯ □

The proof of the following lemma is just as easy.

Lemma 6.18 *The set* $S \cup S'$ *output by Algorithm 6.7 is a valid upgrading set in* G. □

Using the results of Lemma 6.16, Lemma 6.17, and Lemma 6.18, we can now complete the proof of Theorem 6.15 in the following way. By Lemma 6.16, we know that

$$|S^*| = \text{OPT} \geq |\mathcal{C}^*|, \tag{6.18}$$

where $\mathcal{C}^*$ is an optimal set cover for the instance $(Q, \mathcal{F})$ of MIN SET COVER constructed in Step 6. Step 7 of the algorithm, i.e., finding an approximation to the optimal set cover, can be done using the greedy Algorithm 6.6 from page 114. As stated in Theorem 6.14, this algorithm will produce a set cover $\mathcal{C}$

of size at most $(1 + \ln \max\{|S_v| : v \in V\}) \cdot |\mathcal{C}^*|$. Since $|S_v| \le \Delta$, where Δ is the maximum degree in the graph G and $|S| = |\mathcal{C}|$, this yields

$$|S| = |\mathcal{C}| \le (1 + \ln \Delta) \cdot |\mathcal{C}^*| \overset{(6.18)}{\le} (1 + \ln \Delta) \cdot \mathrm{OPT}. \tag{6.19}$$

We now address the cardinality of the set S'. Since by Lemma 6.17 the bottleneck graph G' computed in Step 15 has at most $|S|$ connected components, it follows that the tree $\hat{T}$ computed in Step 18 contains no more than $|S|-1$ edges. Since in the loop in Steps 19 to 22 for each edge in T_H we add one vertex to S', we have that $|S'| \le |S| - 1$. Combining this result with (6.19) we obtain:

$$|S \cup S'| \le |S| + |S'| \le 2|S| - 1 \le 2(1 + \ln \Delta) \cdot \mathrm{OPT} - 1.$$

This completes the proof of Theorem 6.15. □

Running Time

The running time of Algorithm 6.7 can be analyzed as follows. As usually, let $G = (V, E)$ with $n := |V|$ and $m := |E|$. Computing the connected components of G' needs $\mathcal{O}(n + m)$ time. The instance of MIN SET COVER can also be constructed in linear time. As shown in [CLR90] the greedy Algorithm 6.6 for MIN SET COVER can be implemented to run in time $\mathcal{O}(\sum_{v \in V} |S_v|)$, which is in $\mathcal{O}(m)$ in our case. Computing the auxiliary graph in Step 17 can be accomplished in $\mathcal{O}(n + m)$ time. It is straightforward to verify that all of the remaining steps can also be carried out in linear time.

Theorem 6.19 *Algorithm 6.7 is a polynomial time approximation algorithm for* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *with performance guarantee of* $(2 + 2\ln \Delta, 1)$*, when all edges are either uncritical or medium critical, where* Δ *is the maximum degree in the input graph* G*. It can be implemented to run in time* $\mathcal{O}(n + m)$. □

Extension to the General Unit Cost Case

We now extend the result of Theorem 6.19 to the case where there are also highly critical edges, i.e., edges where *both* endpoints need to be upgraded in order to make the delay of the edge fall below the threshold D.

The basic idea for the extended algorithm is the following. We first compute the edge subgraph G' of G consisting only of the medium and uncritical edges. Notice that all the edges between the different connected components of G' are highly critical. For each connected component C_i of G' we run Algorithm 6.7

to obtain an upgrading set which makes C_i contain a spanning tree of delay at most D after the upgrade. In the final step, we find a good upgrading set which reduces the delay of some (highly critical) edges between the different components so that the whole graph becomes connected by edges of delay at most D. Algorithm 6.8 displays the extended heuristic.

Algorithm 6.8 Improved algorithm for node upgrading with unit costs.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, and a number D

1 $S \leftarrow \varnothing$
2 $E' \leftarrow \{ e \in E : e$ is either uncritical or medium critical$\}$
3 $G' \leftarrow (V, E')$
4 $C_1, \dots, C_r \leftarrow$ connected components of G'
5 **for** $i \leftarrow 1, \dots, r$ **do**
6 Run Algorithm 6.7 on the connected component C_i and add the upgrading set produced by the algorithm to S
7 **end for**
8 Construct an auxiliary graph $H = (\{C_1, \dots, C_r\}, E_H)$ which contains an edge $(C_i, C_j) \in E_H$ if and only if there exist vertices $v \in C_i$ and $u \in C_j$ such that $(v, u) \in E$.
{Notice that such an edge (v, u) is highly critical}
9 $T \leftarrow$ spanning tree of H
10 $U \leftarrow \varnothing$
11 **for all** edges $(C_i, C_j) \in T$ **do**
12 Choose $v \in C_i$ and $u \in C_j$ with $(v, u) \in E$
13 $U \leftarrow U \cup \{v, u\}$
14 **end for**
15 **return** $S \cup U$

In the sequel we use again OPT to denote the cardinality of an optimal solution S^*.

Lemma 6.20 *For the size of an optimal solution and the set U of vertices constructed in Steps 11 to 14 the following inequalities hold:*

$$\mathrm{OPT} \geq r \qquad \text{and} \qquad |U| \leq 2(r-1), \tag{6.20}$$

where r is the number of components of the graph G' computed in Step 3.

Proof: Let T^* be a bottleneck spanning tree in the graph resulting from the upgrade of the vertices in the optimum set S^*. Since there are r connected components $C_1, \dots, C_r$ in the graph G', T^* contains at least $r-1$ highly critical edges having endpoints in different components. For each of these edges, both endpoints must belong to OPT, which implies that $|\mathrm{OPT}| \geq r$. This proves the first inequality in (6.20).

On the other hand, Algorithm 6.8 computes a spanning tree T in Step 9 which has $r-1$ edges. For each of the $r-1$ edges of T the algorithm adds two vertices to U. Thus, $|U| \leq 2(r-1)$. □

Performance Guarantee

We are now ready to establish the performance of the extended approximation algorithm.

Lemma 6.21 *The performance of Algorithm 6.8 as applied to* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *is* $(5 + 4 \ln \Delta, 1)$, *where* Δ *denotes the maximum degree in the graph given in the input.*

Proof: Again, let T^* be a bottleneck spanning tree in the graph resulting from the upgrade of the vertices in the optimal upgrading set S^*. Let OPT_i be the minimum number of nodes which must be upgraded in component C_i to make it contain a spanning tree of delay at most D, i.e., OPT_i is the optimal solution value of the instance of (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) given by the graph $G[C_i]$, the restriction of d, d_m, d_l to the edges of $G[C_i]$ and the bound D. Consider the intersection of S^* with the cluster C_i. We are going to show that the inequality

$$|S^* \cap C_i| \geq \frac{1}{2}\mathrm{OPT}_i \tag{6.21}$$

holds true. In fact, let $S_i := S^* \cap C_i$. We first prove that, after upgrading the nodes in S_i, each node in $C_i \setminus S_i$ is connected to at least one node from S_i via paths in C_i containing edges of delay no more than D.

Assume that this is not the case. Then there is a node $v \in C_i \setminus S_i$, whose unique path in T^* to any node in S_i contains at least one highly critical edge. Fix $w \in S_i$ and consider the path $(v = u_0, u_1, \dots, u_k = w)$ from v to w in T^*. Without loss of generality, we can assume that this path does not contain any node from S_i different from w. Let ℓ be the smallest number such that $(u_\ell, u_{\ell+1})$ is highly critical. As $v \notin S_i$, we have $\ell \geq 1$. Since after the upgrade T^* contains only edges of delay at most D, we see that both nodes u_ℓ and $u_{\ell+1}$ belong to S_i, contradicting the fact that the path did not contain any other node from S_i other than w.

We have seen that after upgrading the nodes in S_i the cluster C_i restricted to the edges of delay at most D contains at most $|S_i| = |S^* \cap C_i|$ connected components. It is now easy to see that upgrading at most one more node from each of these components will make C_i connected by edges of delay at most D. Thus, there exists an upgrading set in C_i of size at most twice the size of S_i. Consequently, the minimum cardinality upgrading set in C_i has also size at most $2|S_i|$, which proves (6.21).

Since the clusters are disjoint, we can conclude that the size of the optimal upgrading set S^* is bounded from below as follows:

$$\mathrm{OPT} = |S^*| = \sum_{i=1}^{r} |S^* \cap C_i| \overset{(6.21)}{\geq} \frac{1}{2} \sum_{i=1}^{r} \mathrm{OPT}_i. \tag{6.22}$$

We know by Theorem 6.15 that Algorithm 6.7 finds a solution for each cluster whose value is bounded from above by $2(1 + \ln \Delta_i) \cdot \mathrm{OPT}_i - 1$, where Δ_i is the maximum degree in the subgraph of G induced by C_i. Thus, using additionally Lemma 6.20, we can estimate the cardinality of the solution set S generated by our algorithm:

$$\begin{aligned}
|S| &\leq 2(r-1) + \sum_{i=1}^{r} \left(2(1 + \ln \Delta_i) \cdot \mathrm{OPT}_i - 1\right) \\
&\leq r - 2 + 2(1 + \ln \Delta) \sum_{i=1}^{r} \mathrm{OPT}_i \\
&\overset{(6.22)}{\leq} r - 2 + 4(1 + \ln \Delta) \cdot \mathrm{OPT} \\
&\overset{\text{Lemma 6.20}}{\leq} (5 + 4 \ln \Delta) \cdot \mathrm{OPT} - 2.
\end{aligned}$$

□

Running Time

The running time of Algorithm 6.8 can be estimated in the following way. Computing the graph G' and its connected components needs time $\mathcal{O}(n+m)$. Let component C_i have n_i nodes and m_i edges. Then, since Algorithm 6.7 as applied to component C_i needs $\mathcal{O}(n_i + m_i)$ time and we have $\sum_{i=1}^{r} n_i = n$ and $\sum_{i=1}^{r} m_i \leq m$, we see that the **for**-loop in Steps 5 to 7 can be accomplished in time $\mathcal{O}(n+m)$. The auxiliary graph H in Step 8 can be computed in linear time as well as a spanning tree of H. Also, the final **for**-loop can be implemented to use only $\mathcal{O}(n + m)$ time. Hence, the whole Algorithm 6.8 runs in linear time. We summarize our results of this section in the following theorem.

Theorem 6.22 *Algorithm 6.8 is an approximation algorithm for* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *with a performance of* $(5 + 4 \ln \Delta, 1)$. *It can be implemented to run in time* $\mathcal{O}(n + m)$. □

Using a similar argumentation as in Lemma 6.2 we obtain the following approximation result for the symmetric problem (BOTTLENECK, UPGRADING SIZE, SPANNING TREE).

Theorem 6.23 *There is an approximation algorithm for* (BOTTLENECK, UPGRADING SIZE, SPANNING TREE) *with a performance of* $(1, 5 + 4 \ln \Delta)$. *It can be implemented to run in time* $\mathcal{O}((n + m) \log n)$. □

6.8 Treewidth-Bounded Graphs

In this section we will show that (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) can be solved in polynomial time if restricted to the class of treewidth-bounded graphs. For the sake of a better presentation we will first show how to solve the problem in polynomial time on series-parallel graphs. Then, we will describe how the ideas carry over to treewidth-bounded graphs.

Restriction to Series-Parallel Graphs

Let G be a series-parallel graph with the two terminals s and t. We call an edge subgraph G' of G consisting of two disjoint spanning trees containing s and t respectively a *terminal forest*.

For a set $M \subseteq \{s, t\}$, define $C(M)$ to be the least cost of an upgrading set S in G with $S \cap \{s, t\} = M$ such that after upgrading this set G contains a bottleneck spanning tree of delay at most D. If there is no upgrading set S such that the bottleneck delay can be reduced to be at most D and $S \cap \{s, t\} = M$, then $C(M) := +\infty$. In the same way as we defined C, we define C' for the minimum upgrading cost to obtain a terminal forest of bottleneck delay at most D.

Clearly, if we know the four values $C(M)$, we can tell the optimum objective function value. We will now show that for a series parallel graph G we can compute C and C' by using the information of the *decomposition tree* of G in a total of $\mathcal{O}(n)$ time. The basic idea is to keep track of which terminals belong to an optimal upgrading set.

First, we will take care of the case that G is the series composition of G_1 and G_2 (see Figure 2.2 on page 18). Assume that we have already computed the values C and C' for G_1 and G_2. Denote them by C_1, C_1' and C_2, C_2' respectively.

It is easy to see that the restriction of any tree T to G_1 and G_2, respectively, is again a tree. Thus, we can compute C with the help of C_1 and C_2 in the following way.

$$C(M) = \min\{C_1(M \setminus \{t\}) + C_2(M \setminus \{s\}), C_1(M \cup \{t\}) + C_2(M \cup \{s\}) - c(t_1)\}$$

The first term above considers the case when the terminal $t_1(= s_2)$ is not upgraded. The second term takes care of t_1 being upgraded.

Similarly, we now compute C' for G. A terminal forest in G must either be a terminal forest in G_1 and a tree in G_2 or vice versa. No other possibilities exist. It now follows that C can be computed by

$$\begin{aligned} C'(M) = \min\{&C_1'(M \setminus \{t\}) + C_2(M \setminus \{s\}), C_1'(M \cup \{t\}) + C_2(M \cup \{s\}) - c(t_1), \\ &C_1(M \setminus \{t\}) + C_2'(M \setminus \{s\}), C_1(M \cup \{t\}) + C_2'(M \cup \{s\}) - c(t_1)\}. \end{aligned}$$

We now consider the case that G is parallely composed from G_1 and G_2 (see Figure 2.2 on page 18). Again, we assume that the two arrays C and C' are already available for G_1 and G_2.

We start with the computation of C. A tree T in G must be a tree in exactly one of the graphs G_1 and G_2 and a terminal forest in the second one. We just need to distinguish between the cases covering the upgrade of the terminals of G_1 and G_2. We must make sure that s_1 is upgraded if and only if s_2 is. We thus obtain C by the following formula:

$$\begin{aligned} C(\{s,t\}) = \min\{&C_1'(\{s,t\}) + C_2(\{s,t\}) - c(s_1) - c(t_1), \\ &C_1(\{s,t\}) + C_2'(\{s \cup t\}) - c(s_1) - c(t_1)\}, \end{aligned}$$

$$C(\{t\}) = \min\{C_1'(\{t\}) + C_2(\{t\}) - c(t_1), C_1(\{t\}) + C_2'(\{t\})] - c(t_1)\},$$

$$C(\{s\}) = \min\{C_1'(\{s\}) + C_2(\{s\}) - c(s_1), C_1(\{s\}) + C_2'(\{s\}) - c(s_1)\},$$

$$C(\varnothing) = \min\{C_1'(\varnothing) + C_2(\varnothing), C_1(\varnothing) + C_2'(\varnothing)\}.$$

We proceed with C'. If G' is a terminal forest of G, it is straightforward to see that the restriction to *both* graphs G_1 and G_2 is a terminal forest of that particular graph. Thus, C' can be computed by using the information from C_1' and C_2' by the following formula:

$$\begin{aligned} C(\{s,t\}) &= C_1'(\{s,t\}) + C_2'\{s,t\} - c(s_1) - c(t_1), \\ C(\{t\}) &= C_1'(\{t\}) + C_2'(\{t\}) - c(t_1), \\ C(\{s\}) &= C_1'(\{s\}) + C_2'(\{s\}) - c(s_1), \\ C(\varnothing) &= C_1'(\varnothing) + C_2'(\varnothing). \end{aligned}$$

Finally, observe that for a series parallel graph consisting of the two terminals s and t and the edge (s, t) we can trivially compute the arrays C and C'.

Using the above recurrences, the array C can be computed in linear time for a series parallel graph G, provided a decomposition tree for G is given. Since such a decomposition tree with $\mathcal{O}(n)$ nodes can be computed in $\mathcal{O}(n)$ time [VTL82], we can conclude that the dynamic programming algorithm presented above runs in total time $\mathcal{O}(n)$. It should be noted that by also keeping track of the respective upgrading sets we can not only find the optimal function value but also the optimal upgrading set.

Theorem 6.24 *If restricted to the class of series-parallel graphs, the problem* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *can be solved optimally in $\mathcal{O}(n)$-time.* □

Extension to Treewidth-Bounded Graphs

Let Γ be any class of decomposable graphs. Let the maximum number of terminals associated with any graph G in Γ be k. Following [BLW87], it is assumed that a given graph G is accompanied by a parse tree of size $\mathcal{O}(n)$ specifying how G is constructed using the rules and that the size of the parse tree is linear in the number of nodes. Moreover, we may assume without loss of generality that the parse tree is a binary tree.

Theorem 6.25 *If restricted to any class of treewidth bounded graphs with no more than k terminals, where k is fixed, the problem* (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) *can be solved optimally in $\mathcal{O}(2^{2k}k^{2k}n)$-time.*

Proof: Let $t_1, \ldots, t_k$ be the terminals of G and let π be a partition of these terminals. Define a π-terminal forest F to be a spanning forest of G with the following properties:

1. For each block of π the forest F contains a tree spanning all the vertices in that block.

2. No pair of trees is connected.

The notion of a π-terminal forest generalizes the concept of spanning trees and terminal forests introduced on page 123. In the case of series parallel graphs, the set of terminals is $\{s, t\}$. The possible partitions of $\{s, t\}$ are $\pi_1 = (\{s, t\}, \emptyset)$ and $\pi_2 = (\{s\}, \{t\})$. Partition π_1 corresponds to a spanning tree of G, while π_2 gives us a terminal tree.

We keep the following information along with each partition π of terminals of G and each subset M of the terminals $\{t_1, \ldots, t_k\}$.

$C^\pi(M)$:= Minimum cost of a subset $S \subseteq V$ with $S \cap \{t_1, \ldots, t_k\} = M$ such that after upgrading the vertices in S the graph G contains a π-terminal tree of bottleneck cost at most D.

For the above defined cost, if there is no forest satisfying the required conditions the value of $C^\pi(M)$ is defined to be $+\infty$. Note that the number of cost values associated with any graph in Γ is $\mathcal{O}(k^k \cdot 2^k)$. We now show how the cost values can be computed in a bottom-up manner given the parse tree for G. Since the method is very similar to the case of series-parallel graphs treated above we only sketch the main ideas.

To begin with, since Γ is fixed, the number of primitive graphs is finite. For a primitive graph, each cost value can be computed in constant time, since the number of forests to be examined is fixed. Now consider computing the cost values for a graph G constructed from subgraphs G_1 and G_2, where the cost values for G_1 and G_2 have already been computed.

Let a partition π and a subset M of the terminals of G be given. Any upgrading set S in G with $S \cap V = M$ resulting in a π-terminal tree of bottleneck delay at most D induces two upgrading sets, one in G_1 and one in G_2. Since we have maintained the best cost values for all possibilities for G_1 and G_2, we can reconstruct for the partition π and the set M the cost value $C^\pi(M)$. We can do this in time independent of the sizes of G_1 and G_2 because they interact only at the terminals to form G, and we have maintained all relevant information.

Hence we can generate all possible cost values for G by considering combinations of all relevant pairs of cost values for G_1 and G_2. This takes time $\mathcal{O}(1)$ per combination for a total time of $\mathcal{O}(2^{2k} \cdot k^{2k})$. As in [BLW87], we assume that the size of the given parse tree for G is $\mathcal{O}(n)$. Thus the dynamic programming algorithm takes time $\mathcal{O}(2^{2k}k^{2k}n)$. This completes the proof. □

The algorithm presented in the proof of the last theorem, although being linear for fixed k, is only practical for small values of k, since the constant factor $2^{2k}k^{2k}$ in front of the n grows extremely fast with k. Thus, the above results might be considered to be more of theoretical interest than application oriented.

6.9 Hardness Results

In this section we establish our hardness results for the node upgrading problems under study. We show that already the problem (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) is hard to approximate within a logarithmic fac-

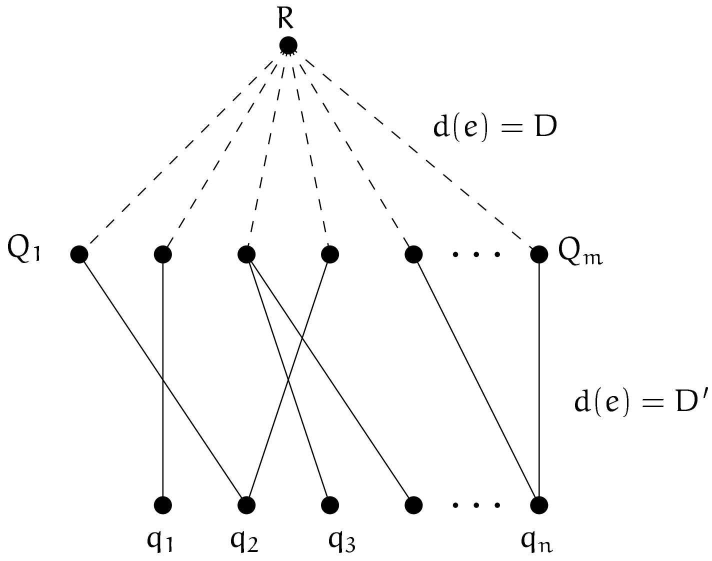


Figure 6.5: Reduction from SET COVER for the hardness of (UPGRADING SIZE, BOTTLENECK, SPANNING TREE).

tor. Naturally, the hardness results carry over to the weighted version, (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE).

Theorem 6.26 *The* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *problem is* NP*-hard for any fixed bound* $D > 0$ *on the bottleneck delay of the tree even for bipartite graphs.*

Proof: We show that SET COVER [GJ79, Problem SP5] reduces to (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) in polynomial time. An instance of SET COVER consists of a finite set Q of ground elements $\{q_1, \ldots, q_n\}$, a collection $\mathcal{F} = \{Q_1, \ldots, Q_m\}$ of subsets of Q and an integer K. The question is whether one can pick at most K sets form $\mathcal{F}$ such that their union equals Q.

Given an instance of SET COVER, we first construct the natural bipartite graph, one side of the partition for set nodes Q_j, $j = 1, \ldots, m$, and the other for element nodes q_i, $i = 1, \ldots, n$. We insert an edge (Q_j, q_i) if $q_i \in Q_j$. We now add one more node R (the "root") and connect R to all the set nodes.

In the remainder of this proof, we do not distinguish between a node and the set or element that it represents. Note that the resulting graph is bipartite (with R and the element nodes on one side and the set nodes on the other side). An example of a graph constructed this way is shown in Figure 6.5.

For each edge of the form $e = (R, Q_i)$, the delay values are $d(e) = d_m(e) = d_l(e) = D$, while each edge of the form $e = (Q_i, q_j)$ has delays $d(e) = D'$ and

$d_m(e) = d_l(e) = D$, where $D' > D$ is arbitrarily chosen (in Figure 6.5 each of the dashed lines has delay D and the solid lines have each delay D'). We set the bound on the bottleneck delay parameter to D.

We now claim that there is an upgrading set of size K resulting in a bottleneck spanning tree of bottleneck delay at most D for the instance of (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) just constructed if and only if the instance of SET COVER has a cover of size at most K.

First assume that we can cover the elements in Q by at most K sets. Without loss of generality assume that the set cover consists of exactly K sets, which are $Q_1 \ldots, Q_K$. We then upgrade the corresponding set nodes. Consider the resulting graph. For each element node q_j there is now an edge of delay D connecting q_j to some set node $S(q_j)$ from $Q_1, \ldots, Q_K$ (If q_j appears in two or more sets whose corresponding nodes are upgraded, then choose one such set node arbitrarily). Then the set

$$\{(R, Q_i) : i = 1, \ldots, m\} \cup \{(q_j, S(q_j)) : j = 1, \ldots, n\}$$

is the edge set of a spanning tree of G, where none of the edges has delay more than D.

Now assume conversely that there is an upgrading set of size at most K for the instance of (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) constructed above. Let $S \subseteq V$, $|S| \leq K$ be a set of nodes that are upgraded and let $T = (V, E_T)$ be a spanning tree in the resulting graph of bottleneck cost at most D.

We now transform the tree T into a tree T' of at most the same bottleneck delay such that each set node Q_j is adjacent to R in T'. To this end, do the following for each set node Q_j, $j = 1, \ldots, m$: If $(R, Q_j) \in T$, then continue with the next set node. Otherwise, since T is connected, there must be a path P from Q_j to R in T. By the bipartiteness of G, the first two edges in this path are of the form (Q_j, q_i) and $(q_i, Q_{j'})$ for some element node q_i and some other set node $Q_{j'}$. Adding the edge (R, Q_j) to T will induce a simple cycle in T containing the edges in P and (R, Q_j). Thus if we remove (Q_j, q_i) from T the resulting graph will again be a spanning tree of G.

Observe that the bottleneck delay of the tree does not increase during the procedure from above, since each edge (R, Q_j) inserted has already delay D in the original graph.

Now, consider the set S of upgraded nodes. If $R \in S$, we can safely remove R from S without affecting the bottleneck cost of our spanning tree T'. For each element node $q_i \in S$, we have at least one set node $s(Q_i)$ adjacent to q_i in T'. We replace q_i by $s(q_i)$ in S and, continuing this, obtain a set S' of at most K set nodes, which are upgraded.

We will now argue that the set nodes from S' form a set cover. To this end, consider an arbitrary element node q_i. If $q_i \in S$, i.e., q_i was one of the upgraded element nodes, then we have added some set node $s(q_i)$ to S that is adjacent to q_i in G. Thus, $s(q_i) \in S'$ contains q_i and, consequently, q_i is covered by the sets in S'. In the other case, $q_i \notin S'$. The tree T' contains at least one edge (Q_j, q_i) of delay at most D. But, since q_i was not upgraded, the only possibility of (Q_j, q_i) having delay D is that Q_j had been upgraded, i.e., $Q_j \in S$. Since we have not removed any set node from S in the transition to S', it follows that $Q_j \in S'$ and thus, again, q_i is covered by the sets in S'. □

Note that the reduction in the proof of Theorem 6.26 preserves approximations. Any set cover of size K becomes an upgrading set of size K and any upgrading set of size K becomes a cover of size at most K. In fact, is is readily verified that the reduction is indeed an L-reduction (see page 14). Combining this approximation preserving transformation with the hardness results of Feige [Fei96] (Theorem 3.12 on page 36) about MIN SET COVER and replacing D' by $f(n) \cdot D + 1$, where f is a polynomial time computable function, we get the following theorem.

Theorem 6.27 *Unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$*, there can be no polynomial time approximation algorithm for* (UPGRADING SIZE, BOTTLENECK, SPANNING TREE) *with a performance of* $(\alpha, f(n))$ *for any fixed* $\alpha < \ln n$ *and any polynomial time computable function* f. □

Theorem 6.27 shows that the performance $(2 \ln n, 1)$ of our approximation Algorithm 6.1 is essentially the best unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$. Finally, an application of Lemma 6.3 gives us an analogous hardness result for the symmetric problem.

Corollary 6.28 *Unless* $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{\mathcal{O}(\log\log n)})$*, there can be no polynomial time approximation algorithm for* (BOTTLENECK, UPGRADING SIZE, SPANNING TREE) *with a performance of* $(f(n), \alpha)$ *for any fixed* $\alpha < \ln n$ *and any polynomial time computable function* f. □

6.10 The Link Delay Problem

In the (NODE UPGRADING COST, BOTTLENECK, SPANNING TREE) problem we were searching for a minimum cost upgrading set such that in the upgraded network we could find a bottleneck spanning tree of bottleneck delay at most a given bound D. It is also meaningful to ask for a minimum cost upgrading set such that after the upgrade *all* the edges in the network are of delay at most D and not only those in a bottleneck spanning tree. This problem has been called the *link delay problem* or LINKDELAY for short in literature [PS95].

Paik and Sahni showed in [PS95] that LINKDELAY is NP-hard by providing a reduction from VERTEX COVER , which is the decision version of MIN VERTEX COVER given in Example 3.4 on page 23. The reduction was done in their node upgrading model mentioned in Section 6.4. Since this model is a special case of our model, we immediately obtain hardness of LINKDELAY in our setting.

We present a simple polynomial time approximation algorithm for LINKDELAY with a performance of $(2, 1)$. Its basic outline is as follows. The upgrading set S is initialized to be the empty set. If there are useless edges, i.e., edges where upgrading can not reduce the delay to be at most D, we inform about the infeasibility of the problem. Then, for each highly critical edge both endpoints are added to the upgrading set S. It follows that up to this point S is a subset of the optimal solution S^*. We update the delays of the edges incident with the nodes in S. Observe that for each of the edges which still has delay strictly greater than the bound D it suffices to upgrade *at most one* of its endpoints. Our algorithm finds an approximation C to the problem of finding a minimum cost vertex cover in the edge subgraph of G consisting of those edges whose delay still exceeds D. We output $S \cup C$ as an upgrading set.

The heuristic is shown in Algorithm 6.9. Step 12, i.e., computing an approximate minimum cost vertex cover can be done using known approximation algorithms, see e.g. [Mot92]. The following theorem states our approximation result:

Theorem 6.29 *Algorithm 6.9 is an approximation algorithm for* LINKDELAY *with a performance of* $(2, 1)$.

Proof: Let S^* be an upgrading set of minimum cost. Clearly, the set S in Step 7 of Algorithm 6.9 must be contained in S^*. It is also easy to see that $S^* \setminus V'$ must be a vertex cover in the graph G' as defined in Step 12 of the algorithm. Thus, the cost of the nodes in $S^* \setminus V'$ is at least that of the optimum vertex cover in G'. The proof can now be completed in a straightforward manner. □

The proof of Theorem 6.29 shows also that, if we can compute a minimum cost vertex cover in G' efficiently, i.e., in polynomial time, then this result immediately carries over to LINKDELAY. Bern et al. have shown [BLW87] that the minimum cost vertex cover problem can be solved in polynomial time on graphs of bounded treewidth. Thus, we can conclude:

Corollary 6.30 *The restriction of* LINKDELAY *to graphs of bounded treewidth can be solved in polynomial time.* □

We now address the hardness of LINKDELAY. As already mentioned, NP-hardness has been established in [PS95]. We briefly argue that in fact one can find an L-reduction (see page 14) from MIN VERTEX COVER to LINKDELAY.

Algorithm 6.9 Approximation algorithm for the Link Delay Problem.

Input: A graph $G = (V, E)$, three edge weight functions d, d_m, d_l, a node weight function c, and a number D

for all $e \in E$ **do**
  if $d_l(e) > D$ **then** {e is a "useless edge"}
    return "The link delay cannot be reduced to be at most D"
  end if
end for
$E_{high} \leftarrow \{ e \in E : e \text{ is highly critical} \}$
$S \leftarrow \{ u, v : (u, v) \in E_{high} \}$
for all $e \in E - E_{high}$ such that e is incident with some node in S **do**
  $d(e) \leftarrow d_m(e)$ {one endpoint of e is already upgraded}
end for
$E' := \{ e \in E - E_{high} : d(e) > D \}$
Compute a 2-approximation C to the problem of finding a minimum total cost vertex cover in the graph $G' = (V, E')$
return $S \cup C$

Given an instance I of MIN VERTEX COVER given by a graph $G = (V, E)$ with $|V| =: n$ we construct an instance I' of LINKDELAY as follows: We use the same graph $G = (V, E)$, set $c(v) = 1$ for all $v \in V$, $d(e) = d_m(e) = f(n) + 1$ and $d_l(e) = 1$ for all $e \in E$, where f is any polynomial time computable function. Finally, let the bound D be 1.

Any vertex cover in G will give us a feasible upgrading set for the instance I' of LINKDELAY. Thus, the optimum size upgrading set has cardinality at most that of the minimum vertex cover: $\mathrm{OPT}(I') \leq \mathrm{OPT}(I)$. It is also straightforward to see that any feasible upgrading set for I' will form a vertex cover in G. This way, we have found a L-reduction from MIN VERTEX COVER to LINKDELAY. Moreover, this L-reduction shows us that the existence of an $(\alpha, f(n))$-approximation algorithm for LINKDELAY implies that there is an α-approximation algorithm for MIN VERTEX COVER.

As shown in [PY91] the MIN VERTEX COVER problem is MaxSNP-hard.[2] Thus, our above arguments show that LINKDELAY is also MaxSNP-hard and by the results from [AL+92] (see Theorem 3.11) there can be no PTAS for LINKDELAY, unless P = NP.

In [BGS95] a stronger hardness result than MaxSNP-hardness is established for MIN VERTEX COVER. It is shown that, unless P = NP, MIN VERTEX COVER can not be approximated with a performance of $\alpha < 1.038$. Thus, we obtain

[2] The problem is MaxSNP-complete in graphs of bounded degree.

the following result on the hardness of LINKDELAY.

Theorem 6.31 *Unless* $\mathsf{P} = \mathsf{NP}$*, for any* $\alpha < 1.038$ *and any polynomial time computable function* f*, there is no* $(\alpha, f(n))$*-approximation algorithm for the* LINKDELAY *problem.* □

7

Network Edge Improvement Problems

7.1 Introduction

The problem of computing a minimum spanning tree for a network is a well studied problem in computer science. In this chapter, we consider a variant of the problem where the goal is to shorten the edge lengths of a given network so that the length of a minimum spanning tree in resulting network is as small as possible. The problem is considered in a context where there is a cost associated with shortening a link and there is a budget constraint on the total cost of shortening the edges. Thus, the network improvement problems considered in this chapter are based on an *edge based upgrading model* in contrast to the node based upgrading model discussed in Chapter 6.

The network edge improvement problem outlined above models situations wherein an organization desires to take advantage of technological advances to upgrade the communications network interconnecting its branches and has allocated a fixed budget to do so. The goal is to devise a strategy that upgrades the links of the network so that the total upgrading cost is within the chosen budget, and the length of a minimum spanning tree in the upgraded network is the smallest among all strategies that obey the budget constraint. A precise definition of the problem is given in the next section. As already mentioned in Chapter 6 such problems arise in diverse areas such as design of high speed communication networks [KJ83], video on demand [KPP93], teleconferencing [KPP92a, KPP92b], VLSI design [CK+92, ZPD94] and database retrieval [Rav94].

We prove that in general the problem of computing an optimal reduction is NP-hard even for very simple classes of graphs and linear cost functions on the edges. We also present efficient approximation algorithms covering a broad class of cost functions. Our techniques introduced here are not restricted to the case of spanning trees. We show that they extend to more general network design problems specified by submodular functions such as those in [GW92, GG+94]. In particular, this enables us to find good solutions to edge improvement problems where the goal is to minimize the weight of a Steiner tree or a generalized Steiner network.

The remainder of the chapter is organized as follows. Our edge based improvement model is presented in the next Section. In Section 7.3 we briefly discuss the structure of an optimal solution for linear reduction costs. Section 7.4 presents our approximation algorithm for general graphs and general reduction cost functions. Section 7.5 shows how our ideas extend to more general subgraph classes than just spanning trees. We improve our approximation results in Section 7.6 for linear reduction costs on the edges. Section 7.7 contains a further improved version for the class of treewidth-bounded graphs. In Section 7.8 we establish hardness results for the problems tackled in this chapter.

7.2 Preliminaries and Problem Definition

Let $G = (V, E)$ be an undirected graph. Associated with each edge $e \in E$, there are nonnegative values as follows: $\ell(e)$ denotes the *length* or the *weight* of the edge e and $\ell_{\min}(e)$ denotes the *minimum length* to which the edge e can be reduced. Consequently, we assume throughout the presentation that $\ell_{\min}(e) \leq \ell(e)$. The nonnegative *cost function* c_e indicates how expensive it is to reduce the length of e by a certain amount. We assume without loss of generality that $c_e(0) = 0$ for all edges $e \in E$.[1] We also make the natural assumption that each cost function c_e is nondecreasing.

A *reduction strategy* (or simply *reduction*) r on the edges of G specifies how to reduce the ℓ-length of each edge e to a value in the range $[\ell_{\min}(e), \ell(e)]$. Formally, we require that

$$\ell(e) - r(e) \geq \ell_{\min}(e) \qquad \text{for all } e \in E.$$

The *cost* of the reduction r is $\sum_{e \in E} c_e(r(e))$. If r is a reduction in G, then we can consider the graph G with edge weights given by the "reduced lengths", namely $(\ell - r)(e) := \ell(e) - r(e)$ $(e \in E)$.

Let T be a spanning tree of G. The *total length* of T under the weight function ℓ, denoted by $\ell(T)$, is defined to be the sum of the lengths of the edges that are in T. We denote the total length of a minimum spanning tree with respect to the weight function ℓ by $\mathrm{MST}_G(\ell)$. Similarly, if r is a (feasible) reduction in G then $\mathrm{MST}_G(\ell - r)$ denotes the weight of a MST with respect to the reduced lengths $\ell(e) - r(e)$ $(e \in E)$. We omit the graph G in the subscript whenever such an omission does not cause any ambiguity.

[1] Any reduction will incur a minimum cost of $\sum_{e \in E} c_e(0)$ and we can subtract this sum from the budget in advance.

We are now ready to state the basic problem studied in this chapter. The goal is to shorten the edges in the graph such that the resulting graph has a spanning tree of small weight. A formal statement of the problem is given in the next definition:

Definition 7.1 (Budget Minimum Total Cost Spanning Tree Problem)
Given an edge-weighted graph G *as above and a nonnegative budget value* B*, the* ***budget minimum total cost spanning tree problem****,* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE)*, is to find a subgraph from the set*

$$\left\{ (T, r) : T \text{ is a spanning tree of } G, r \text{ a reduction on } G \text{ and } \sum_{e \in T} c_e(r(e)) \leq B \right\},$$

such that $(\ell - r)(T)$ *is as small as possible.*

Since given a reduction r finding a tree with minimum total length under $\ell - r$ can be accomplished by one minimum spanning tree computation (and thus in particular in polynomial time), we could have also formulated the above problem as that of finding a feasible reduction r such that $\mathrm{MST}_G(\ell - r)$ has the least possible value and to output a corresponding tree. The reasons why we chose to give the formulation in Definition 7.1 are the following: On the one hand we wanted to show how the problem fits smoothly into our framework given in Definition 3.1. On the other hand the above version can be generalized more easily to the case when we do not wish to minimize the weight of a spanning tree but rather the weight of subgraphs that are "harder" to find, e.g. to Steiner trees.

Let Γ be a subgraph class and denote by $\Gamma^*_G(\ell)$ the minimum cost subgraph with respect to ℓ. Then, the network edge improvement problem in its most general form can be stated as finding a subgraph from

$$\left\{ (S, r) : S \in \Gamma, r \text{ is a reduction on } G \text{ and } \sum_{e \in S} c_e(r(e)) \leq B \right\},$$

such that $(\ell - r)(S)$ is minimized.

In the sequel we will deal with the approximation of (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE). In Section 7.5 we will show how our ideas extend nicely to more general subgraph classes Γ. Chapter 8 addresses the case of Steiner trees with additional constraints such as bounds on the maximum degree in the tree.

An (α, β)-approximation algorithm for the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) problem, is required to find a reduction r of cost at most βB and a spanning tree T of the graph G such that such that

$$\frac{(\ell - r)(T)}{\mathrm{MST}_G(\ell - r^*)} \leq \alpha, \tag{7.1}$$

where r^* denotes an optimal edge-reduction on G of cost at most B.

Notice again that in the case of spanning trees (which are our main concern in this chapter) instead of (7.1) we can equivalently require that

$$\frac{\mathrm{MST}_G(\ell - r)}{\mathrm{MST}_G(\ell - r^*)} \leq \alpha.$$

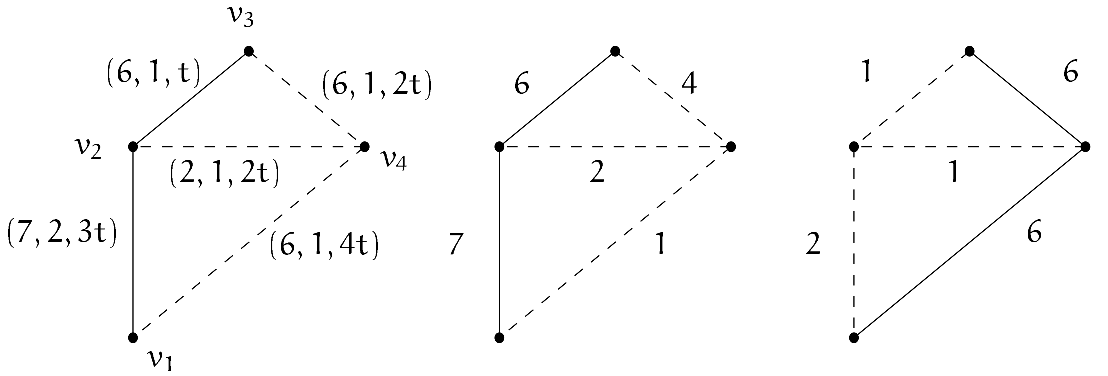
Example 7.2 Consider the graphs given in Figure 7.1 on the facing page. Figure 7.1 on the next page(a) shows a graph G where each edge e is associated with the three values $(\ell(e), \ell_{\min}(e), c_e(t))$. The cost functions c_e on the edges in this simple example are linear. The dashed lines indicate the edges of the minimum spanning tree in G.

The result of a modification of G is shown in Figure 7.1(b). Again, the edges belonging to the minimum spanning tree are drawn as dashed lines. The modification corresponding to Figure 7.1(b) involves a cost of 24 and the weight of the resulting MST is 7.

Figure 7.1(c) shows the graph with edge lengths resulting from a reduction that is optimal among all reductions of cost no more than 22. There, the weight of the spanning tree resulting from the reduction is 4. Thus, the reduction of Figure 7.1(b) is a $(7/4, 24/22)$-approximation to an optimal solution with budget 22. ◇

As in Chapter 6 we can consider a symmetric problem to (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) in which we have a bound D on the weight of an MST in the upgraded network and we want to find a reduction of minimum cost. Denote this symmetric problem by (EDGE IMPROVEMENT COST, TOTAL WEIGHT, SPANNING TREE). Let A be an (α, β)-approximation algorithm for (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE). Let $\varepsilon > 0$ be fixed. We can use this algorithm to obtain a $((1+\varepsilon)\beta, \alpha)$-approximation for (EDGE IMPROVEMENT COST, TOTAL WEIGHT, SPANNING TREE) in that we basically search for the optimal upgrading cost. We search the set

$$\mathcal{M} := \{0, 1+\varepsilon, (1+\varepsilon)^2, \ldots, (1+\varepsilon)^k\}, \text{ where } k = \left\lceil \log \sum_{e \in E} c_e(\ell(e) - \ell_{\min}(e)) \right\rceil$$



(a) The original graph G. (b) Modification of G with cost 24. (c) Optimal Modification for a budget of 22.

Figure 7.1: An example of a graph modification via edge reductions.

and find the minimum value $t \in \mathcal{M}$ such that A finds a reduction r of cost at most βt such that $\mathrm{MST}_G(\ell - r) \leq \alpha D$. Using basically the arguments from Lemma 6.3 it is then straightforward to establish the following result:

Lemma 7.3 *Suppose that there is a bicriteria approximation algorithm for the* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *problem with a performance of* (α, β). *Then, for any* $\varepsilon > 0$ *there exists a* $((1+\varepsilon)\beta, \alpha)$-*approximation algorithm for* (EDGE IMPROVEMENT COST, TOTAL WEIGHT, SPANNING TREE). □

7.3 Structure of an Optimal Solution

In this section we comment on the structure of optimal solutions to the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE)-problem for *linear reduction costs* on the edges, i.e., for each edge $e \in E$ we have $c_e(t) = c_e \cdot t$ for a constant c_e. We also look at special cases of the problem that can be solved in polynomial time.

First, suppose that the given budget B is zero. Then (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) reduces to the well known minimum spanning tree problem (with length function $\ell(e)$), and is known to be optimally solvable by classical algorithms (e.g. Kruskal's algorithm). Similarly, if $B = +\infty$ (i.e., there is no bound on the cost of upgrading the network), the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) problem again reduces to the minimum spanning tree problem but this time with edge-lengths given by $\ell_{\min}$.

Optimal solutions to (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) also exhibit some structure in the general case (i.e., $B \notin \{0, +\infty\}$). Any (feasible) reduction r induces a tree in a natural way, namely a minimum spanning tree T_r in the modified graph. Observe that the quality of the solution produced via the reduction r depends solely on the weight of T_r, so all the cost incurred in upgrading edges not in T_r is wasted. Moreover, for any *fixed* tree T in G, the Greedy-strategy that successively reduces a cheapest available edge is an optimal reduction strategy. Thus, if we already knew a minimum spanning tree T_{r^*} corresponding to an optimal reduction r^*, we could solve (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) quite easily.

This observation also suggests a very simple exponential time algorithm for solving (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE): Enumerate all spanning trees in G, apply the Greedy strategy to each of them and then select the best solution. Unfortunately, a graph G with n nodes can have n^{n-2} different spanning trees.

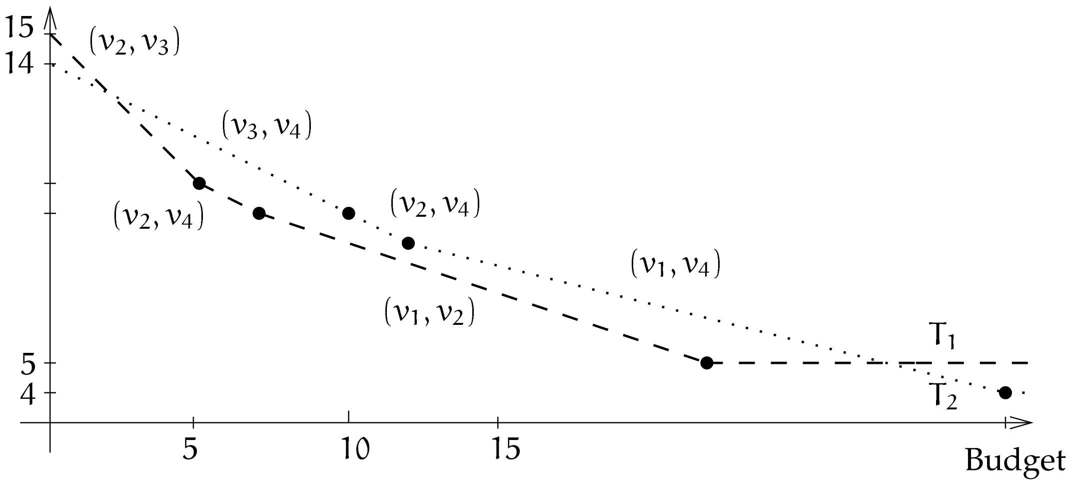


Figure 7.2: Remaining weight of the trees T_1 and T_2 as a function of the budget.

We now discuss the *sensitivity* of optimal reduction strategies to changes in the given budget B. If we fix a spanning tree and plot the weight of that tree as a function of the money spent on it in a Greedy manner, we see that each piece corresponds to a budget range where one particular edge e is shortened. Thus it is easy to see that the piece has slope $-1/c_e$.

Figure 7.2 shows the plots corresponding to the tree T_1 consisting of the edges (v_2, v_3), (v_2, v_4), (v_1, v_2) and the tree T_2 consisting of the edges (v_3, v_4), (v_2, v_4)

and (v_1, v_4) taken from the example graph of Figure 7.1 on page 137. As can be seen from Figure 7.2, the plots for different trees can cross each other multiple times. If we plot the weights of all spanning trees on the same set of axes, the lower envelope gives the optimal remaining weight per budget. It is easy to see that the lower envelope can have an exponential number of linear pieces.

7.4 Approximation Algorithms on General Graphs

In this section, we present our approximation algorithm. As mentioned earlier, the approximation algorithm extends easily to a broad class of network improvement problems where the objective to be minimized is the total cost of a connected subnetwork (e.g. budget constrained minimum Steiner tree problem). This extension is discussed in Section 7.5.

We first give an informal description of how the algorithm works. The main procedure uses a parametric search. In this search, the algorithm tries to find a good compromise between weighing the total length and the corresponding reduction cost of a tree in general. To this end, the algorithm performs a binary search with parameter K on the interval

$$\mathcal{I} := \left[\frac{1}{\varepsilon} (n-1) \min_{e \in E} \ell_{\min}(e), \frac{1}{\varepsilon} (n-1) \max_{e \in E} \ell(e) \right]. \tag{7.2}$$

Note that if $\mathrm{MST}_G(\ell - r^*)$ denotes the total weight of a minimum spanning tree after an optimal reduction r^*, then

$$\frac{1}{\varepsilon} \mathrm{MST}_G(\ell - r^*) \in \mathcal{I}.$$

For each $K \in \mathcal{I}$, which is probed with the help of a test procedure during the search, the algorithm first calculates a coarse heuristic measure that indicates how important it is to shorten an edge. Then, for each edge e in the graph the blend of its length and the reduction cost is refined by using the information in the cost function c_e. After calculating such *compound costs* for the edges, we compute a minimum spanning tree with respect to these costs. The algorithm stops when a good blend has been found, meaning in this context that there exists a tree of total compound cost that is small compared to the current parameter K.

For large values of K the reduction costs on the edges are weighted more than their lengths and the algorithm will tend to reduce the edge lengths only by a small amount, resulting in low overall reduction costs and more or less heavy trees. Also, since K is large, the test on the compound cost of the minimum spanning tree computed will succeed. The algorithm now tries to reduce K as much as possible and find a minimum $K \in \mathcal{I}$ such that it can successfully compute a light compound cost spanning tree.

Algorithm 7.1 displays our approximation algorithm. This algorithm uses the test procedure given in Algorithm 7.2.

Algorithm 7.1 Approximation algorithm for edge improvement.

Input: A graph $G = (V, E)$, two edge weight functions ℓ, $\ell_{\min}$, cost functions c_e $(e \in E)$, a number B; two *fixed* accuracy parameters $\varepsilon, \varepsilon' > 0$

1 Perform a binary search on the interval

$$\mathcal{I} = \left[\frac{1}{\varepsilon}(n-1)\min_{e\in E}\ell_{\min}(e), \frac{1}{\varepsilon}(n-1)\max_{e\in E}\ell(e)\right] \tag{7.3}$$

with a spacing[2] of ε' to find the minimum value $K' \in \mathcal{I}$ such that Algorithm 7.2 called with the test parameter K' does not return COF.

2 Let T' be the tree generated by Algorithm 7.2 for the test value K' and let t'_e $(e \in T')$ be the corresponding "fine tuned" blend parameters.

3 Define the reduction r by $r(e) := 0$ if e is not included in T and by $r(e) := t'_e$ otherwise.

4 **return** T and r

Correctness and Performance Guarantee

The performance guarantee provided by Algorithm 7.1 is summarized in the following theorem.

Theorem 7.4 *For any fixed $\varepsilon, \varepsilon' > 0$, Algorithm 7.1 is an approximation algorithm for* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *that finds a solution whose length is at most $1 + 1/\varepsilon$ times that of a minimum length spanning tree plus an additive constant of at most ε', and the total cost of the improvement is at most $1 + \varepsilon$ times the budget B.*

The proof of Theorem 7.4 relies mainly on the following Lemma 7.5, which ensures that the binary search in the main procedure works correctly. In stating this lemma, we use the notation introduced in the two Algorithms 7.1 and 7.2 described above.

[2]Searching an interval $[a, b]$ with a *spacing* of ε' means testing all the values $a + k \cdot \varepsilon'$ for $k = 0, \ldots, \lceil \frac{b-a}{\varepsilon'} \rceil$.

Algorithm 7.2 Test procedure for edge improvement.

Input: A graph $G = (V, E)$, two edge weight functions ℓ, $\ell_{\min}$, cost functions c_e $(e \in E)$, a number B, and a test parameter K; a *fixed* accuracy parameter $\varepsilon > 0$

1 {This procedure tries to estimate whether in the current blend of lengths and reduction costs, the costs are weighted strongly enough (i.e., whether K is large enough) resulting in a low cost reduction. For this purpose, it uses the heuristic measure computed in Step 3.}
2 **for all** $e \in E$ **do**
3 $\quad h_K(e) \leftarrow \min_{t \in [0, \ell(e) - \ell_{\min}(e)]} \left(\ell(e) - t + \frac{K}{B} c_e(t)\right)$
$\quad$ Also, let t_e be the value of t which achieves the value $h_K(e)$.
4 **end for**
5 $T \leftarrow \mathrm{MST}_G(h_K)$
6 **if** $h_K(T) \leq (1 + \varepsilon)K$ **then**
7 $\quad$ **return** T and t_e $(e \in T)$
8 **else**
9 $\quad$ **return** COF
10 **end if**

Lemma 7.5 *Define* F *on* $\mathbb{R}_+ \setminus \{0\}$ *by* $F(K) := \frac{\mathrm{MST}_G(h_K)}{K}$. *Then* F *is monotonically nonincreasing on* $\mathbb{R}_+ \setminus \{0\}$.

Proof: Let $K^{(1)}$ and $K^{(2)}$ be two positive numbers such that $K^{(1)} < K^{(2)}$. For $i = 1, 2$ let $T^{(i)}$ be a minimum spanning tree in G under the cost function $h_{K^{(i)}}$. Then

$$h_{K^{(i)}}(T^{(i)}) = \sum_{e \in T^{(i)}} h_{K^{(i)}}(e) = \underbrace{\sum_{e \in T^{(i)}} (\ell(e) - t_e^{(i)})}_{=: L^{(i)}} + \frac{K^{(i)}}{B} \underbrace{\sum_{e \in T^{(i)}} c_e(t_e^{(i)})}_{=: C^{(i)}}$$
$$=: L^{(i)} + \frac{K^{(i)}}{B} C^{(i)}.$$

Here, $t_e^{(i)}$ are the values chosen in Step 3 of Algorithm 7.2 which minimize $\ell(e) - t + \frac{K^{(i)}}{B} c_e(t)$ on the interval $[0, \ell(e) - \ell_{\min}(e)]$. By dividing the last equation by $K^{(i)}$ we obtain that

$$F(K^{(i)}) = \frac{L^{(i)}}{K^{(i)}} + \frac{C^{(i)}}{B} \quad \text{for } i \in \{1, 2\}. \tag{7.4}$$

In the next step we find an upper bound for $F(K^{(2)})$. To this end, we estimate the weight of each edge in $T^{(1)}$ under the cost function $h_{K^{(2)}}$. Let $e \in T^{(1)}$ be an arbitrary edge. Then by the choice of $t_e^{(2)}$ in Step 3 of Algorithm 7.2 it follows that

$$\begin{aligned} h_{K^{(2)}}(e) &= \ell(e) - t_e^{(2)} + \frac{K^{(2)}}{B} c_e(t_e^{(2)}) \\ &= \min_{t \in [0, \ell(e) - \ell_{\min}(e)]} \left(\ell(e) - t + \frac{K^{(2)}}{B} c_e(t) \right) \\ &\leq \ell(e) - t_e^{(1)} + \frac{K^{(2)}}{B} c_e(t_e^{(1)}). \end{aligned} \tag{7.5}$$

Summing up the inequalities in (7.5) over all $e \in T^{(1)}$, we obtain:

$$h_{K^{(2)}}(T^{(1)}) \leq L^{(1)} + \frac{K^{(2)}}{B} C^{(1)}. \tag{7.6}$$

Dividing (7.6) by $K^{(2)}$ and using the fact that $h_{K^{(2)}}(T^{(2)}) \leq h_{K^{(2)}}(T^{(1)})$ this results in

$$F(K^{(2)}) \leq \frac{L^{(1)}}{K^{(2)}} + \frac{C^{(1)}}{B} < \frac{L^{(1)}}{K^{(1)}} + \frac{C^{(1)}}{B} = F(K^{(1)}). \tag{7.7}$$

The strict inequality in the chain above stems from the fact that $K^{(1)} < K^{(2)}$. This completes the proof of the lemma. □

Corollary 7.6 *If Algorithm 7.2 does not return COF for some $K' > 0$ then it does not return COF for all $K > K'$ either. Thus, the binary search in Algorithm 7.1 works correctly.*

Proof: Let T' be a minimum spanning tree with respect to $h_{K'}$. Then, since the test procedure in Algorithm 7.2 does not return COF we have that $h_{K'}(T') \leq (1+\varepsilon)K'$, i.e., $F(K') \leq (1+\varepsilon)$. Thus, it follows by Lemma 7.5 on the page before that $F(K) \leq (1+\varepsilon)$ for all $K > K'$. Since $F(K) = \mathrm{MST}_G(h_K)/K$, this is equivalent to saying that $\mathrm{MST}_G(h_K) \leq (1+\varepsilon)K$ for all $K > K'$. □

We now have all the ingredients to complete the proof of Theorem 7.4.

Proof of Theorem 7.4: Let r^* be an optimal feasible reduction and let T^* be a minimum spanning tree in G with respect to the weight function $\ell - r^*$. For the sake of shorter notation let OPT := $(\ell - r^*)(T^*)$ be its total weight in the graph with the edge lengths resulting from the optimal reduction r^*.

We first show that Algorithm 7.2 would not return COF if called with the value $\tilde{K}$ which is the smallest value in the ε'-spacing of the interval $\mathcal{I}$ satisfying the condition $\tilde{K} \geq \mathrm{OPT}/\varepsilon$. Thus, $\tilde{K}$ is some rational number satisfying

$$\tilde{K} = \frac{\mathrm{OPT}}{\varepsilon} + \varepsilon'', \quad \text{where } 0 \leq \varepsilon'' < \varepsilon'. \tag{7.8}$$

For each edge $e \in T^*$ we can estimate the weight $h_{\tilde{K}}(e)$ similar to inequality (7.5) in the proof of Lemma 7.5:

$$\begin{aligned} h_{\tilde{K}}(e) &= \min_{t \in [0, \ell(e) - \ell_{\min}(e)]} \left(\ell(e) - t + \frac{\tilde{K}}{B} c_e(t) \right) \\ &\leq \ell(e) - r^*(e) + \frac{\tilde{K}}{B} c_e(r^*(e)). \end{aligned}$$

This way, we see that the weight of T^* under $h_{\tilde{K}}$ is no more than OPT $+ \frac{\tilde{K}}{B} B$. Consequently, the minimum spanning tree with respect to $h_{\tilde{K}}$ that would be found by the procedure during the call has $h_{\tilde{K}}$-weight at most

$$\text{OPT} + \tilde{K} \overset{(7.8)}{=} \varepsilon(\tilde{K} - \varepsilon'') + \tilde{K} \leq (1 + \varepsilon)\tilde{K}.$$

Hence, the test in Step 7 of Algorithm 7.2 would be successful and the procedure would not return COF. Since we know by Corollary 7.6 that the binary search correctly locates a minimum value K', this now implies that the minimum value K' must satisfy

$$K' \leq \tilde{K} = \frac{\text{OPT}}{\varepsilon} + \varepsilon'', \quad \text{where } 0 \leq \varepsilon'' < \varepsilon'. \tag{7.9}$$

Let T' be the minimum spanning tree found by Algorithm 7.2 when called with the test parameter K'. Since $K', B \geq 0$ and $c_e(t) \geq 0$ for all t, we have

$$h_{K'}(T') = \sum_{e \in T'} (\ell(e) - t'_e) + \frac{K'}{B} \sum_{e \in T'} c_e(t'_e) \geq \sum_{e \in T'} (\ell(e) - t'_e). \tag{7.10}$$

Here, again the numbers t'_e are the values of t chosen in Step 3 of the test procedure in Algorithm 7.2. For the reduction r which is calculated in Step 3 of Algorithm 7.1 it now follows from (7.10) that

$$\text{MST}_G(\ell - r) \leq (\ell - r)(T') \leq h_{K'}(T'). \tag{7.11}$$

Moreover,

$$\begin{aligned} h_{K'}(T') \leq h_{K'}(T^*) &\leq \text{OPT} + \frac{K'}{B} B \\ &\overset{(7.9)}{\leq} \text{OPT} + \frac{\text{OPT}}{\varepsilon} + \varepsilon'' \\ &\leq \left(1 + \frac{1}{\varepsilon}\right) \text{OPT} + \varepsilon' \\ &= \left(1 + \frac{1}{\varepsilon}\right) \cdot \text{MST}_G(\ell - r^*) + \varepsilon'. \end{aligned}$$

Using this result in (7.11), we get $\mathrm{MST}_G(\ell - r) \leq (1 + 1/\varepsilon)\,\mathrm{MST}_G(\ell - r^*) + \varepsilon'$, which proves the claimed performance of the algorithm with respect to the weight of an MST in the graph after applying the reduction r.

We now estimate the cost of the reduction r found by our heuristic. Note that the cost of r is exactly $\sum_{e \in T'} c_e(t'_e)$. We have

$$\frac{K'}{B} \sum_{e \in T'} c_e(t'_e) \leq \sum_{e \in T'} \left(\ell(e) - t'_e + \frac{K'}{B} c_e(t'_e) \right) = h_{K'}(T') \leq (1 + \varepsilon) K'.$$

Dividing the last chain of inequalities by $\frac{K'}{B}$ yields that the budget B is violated by a factor of at most $(1 + \varepsilon)$ as claimed in the theorem. □

Running Time

We now show that the algorithm can be implemented to run in polynomial time for a broad class of reduction cost functions c_e on the edges of the graph. Let

$$L_{\max} = \max\{\, \ell(e) : e \in E \,\}. \tag{7.12}$$

Then the total number of calls to the test Algorithm 7.2 is in $\mathcal{O}(\log(\frac{nL_{\max}}{\varepsilon\varepsilon'}))$. Since ε and ε' are fixed, the test procedure is called only a polynomial number of times. Thus, to prove that the overall running time of the algorithm is polynomial, it suffices to show that each execution of Algorithm 7.2 can be completed in polynomial time. Here, the only fact to show is that we can minimize each of the functions

$$f_e(t) := \ell(e) - t + \frac{K}{B} c_e(t)$$

on the respective compact interval $\mathcal{J} := [0, \ell(e) - \ell_{\min}(e)]$ in Step 3 of the procedure in polynomial time. The rest of the procedure consists of computing a minimum spanning tree which can be done in $\mathcal{O}(n + m\beta(m, n))$ time using the algorithm of Gabow et al. [GG+86], where

$$\beta(m, n) = \min\{\, i : \log^{(i)} n \leq m/n \,\}. \tag{7.13}$$

Consider the execution of Algorithm 7.2 for a given value of K. Observe that in Step 3 the number $\ell(e)$ is an additive constant and $\frac{K}{B}$ is a constant factor. Thus, the constrained minimization of f_e can be done easily for the following sample classes of functions c_e:

1. Linear functions, i.e., $c_e(t) = c_e \cdot t$ for a constant c_e: Then f_e is a linear function in t and the minimum is attained at one of the endpoints of $\mathcal{J}$. Minimizing f_e can be done in constant time. Thus, the total running time of the heuristic is $\mathcal{O}(\log(\frac{nL_{max}}{\varepsilon\varepsilon'})(n + m\beta(m, n)))$. We will show in Section 7.6 how to improve the algorithm for this particular class of functions.

2. Concave functions: Let $\Delta(e) := \ell(e) - \ell_{\min}(e)$. Then, for any $0 < \lambda < 1$ we have by the concavity of c_e (which implies the concavity of f_e):

$$f_e(\lambda \cdot 0 + (1-\lambda)\Delta(e)) \geq \lambda f_e(0) + (1-\lambda) f_e(\Delta(e)) \geq \min\{f_e(0), f_e(\Delta(e))\}.$$

   Thus, the minimum of f_e is again either at 0 or at $\ell(e) - \ell_{\min}(e)$.

3. Differentiable convex functions where we can find a root of the equation $c_e'(t) = \frac{B}{K}$ explicitly.

4. Functions that are piecewise of one of the types described above. Notice that the number of pieces is polynomial in the input size.

For the first three classes of functions that are mentioned above the total effort for our algorithm consists essentially of $\mathcal{O}(\log(\frac{nL_{max}}{\varepsilon\varepsilon'}))$ minimum spanning tree computations, which does not only result in an overall polynomial time but also in a complexity that is feasible in practice.

Theorem 7.7 *For any fixed $\varepsilon, \varepsilon' > 0$, Algorithm 7.1 is an approximation algorithm for* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *that finds a solution whose length is at most $1 + 1/\varepsilon$ times that of a minimum length spanning tree plus an additive constant of at most ε', and the total cost of the improvement is at most $1 + \varepsilon$ times the budget B. For the first three classes of cost functions discussed above it can be implemented to run in time $\mathcal{O}\left(T_{MST} \cdot \log \frac{nL_{max}}{\varepsilon\varepsilon'}\right)$, where $T_{MST} \in \mathcal{O}(n + m\beta(m, n))$ is the time required for an MST computation.* □

Notes on the Algorithm

In the sequel we discuss several modifications of our Algorithm 7.1. First, we show how to avoid the additive constant of ε' in the performance guarantee at the price of a slightly worse multiplicative factor. Then it is shown how our algorithm can be modified to accommodate additional restrictions on the reductions such as integrality constraints. Finally, we discuss the case when the minimization for computing the compound costs of the edges is carried out only approximately and not exactly.

Avoiding an Additive Error

Recall that for any fixed $\varepsilon, \varepsilon' > 0$, Algorithm 7.1 found a tree T whose length is at most $1 + 1/\varepsilon$ times that of a minimum length spanning tree plus an additive constant of at most ε', and the total cost of the improvement was at most $1 + \varepsilon$ times the budget B.

Imagine that we change the binary search in Step 1 of Algorithm 7.1 in the following way. Let $a := \frac{1}{\varepsilon}(n-1)\min_{e\in E} \ell_{\min}(e)$ and $b := \frac{1}{\varepsilon}(n-1)\max_{e\in E} \ell(e)$. Instead of searching the interval $\mathcal{I} = [a, b]$ defined in Equation (7.3) with a spacing of ε' we perform a binary search on the set

$$\mathcal{M} := \left\{a, 2a, 2^2a, \dots, 2^k a\right\}, \qquad \text{where } k = \left\lceil \log \frac{b}{a} \right\rceil. \tag{7.14}$$

We find the minimum value $K \in \mathcal{M}$ such that our test procedure in Algorithm 7.2 called with the test parameter K does not return a COF. Along the lines of the proof of Theorem 7.4 it is straightforward to show that the value K' found this way satisfies

$$K' \leq 2 \cdot \frac{\mathrm{OPT}}{\varepsilon}, \tag{7.15}$$

where = $\mathrm{OPT} := \mathrm{MST}_G(\ell - r^*)$ is again the length of the minimum spanning tree in G resulting from an optimal reduction r^*. Let $T_{K'}$ be the tree found by Algorithm 7.2 when called with the test parameter K'. Using the inequality (7.15) instead of (7.9) in the proof of Theorem 7.4 it then follows that

$$h_{K'}(T'_K) \leq h_{K'}(T^*) \leq \mathrm{OPT} + K' \leq \mathrm{OPT} + 2\frac{\mathrm{OPT}}{\varepsilon} = \left(1 + \frac{2}{\varepsilon}\right)\mathrm{OPT}.$$

Thus, for the reduction r found by our modified algorithm we have

$$\mathrm{MST}_G(\ell - r) \leq \left(1 + \frac{2}{\varepsilon}\right) \cdot \mathrm{OPT}.$$

It also follows by the arguments of the proof of Theorem 7.4 that the cost of the reduction r is at most $1 + \varepsilon$ times the budget B. Algorithm 7.3 shows the modified algorithm resulting from the above discussion.

The running time of Algorithm 7.3 can be estimated similar to page 144: Let $L_{\max}$ be as in (7.12). The size of the set $\mathcal{M}$ defined in (7.14) is in $\mathcal{O}\left(\log \frac{nL_{\max}}{\varepsilon}\right)$, which is polynomial in the input if $\varepsilon > 0$ is fixed. The total number of calls to the test Algorithm 7.2 is in $\mathcal{O}(\log|\mathcal{M}|) \subseteq \mathcal{O}(\log\log \frac{nL_{\max}}{\varepsilon})$. Thus, for the classes of functions presented on page 144 we obtain a polynomial time approximation algorithm with a performance of $(1 + 2/\varepsilon, 1 + \varepsilon)$.

Algorithm 7.3 Modified algorithm avoiding the additive error.

Input: A graph $G = (V, E)$, two edge weight functions ℓ, $\ell_{\min}$, cost functions c_e $(e \in E)$, a number B; a *fixed* accuracy parameter $\varepsilon > 0$

1 $a \leftarrow \frac{1}{\varepsilon}(n-1)\min_{e\in E} \ell_{\min}(e)$
2 $b \leftarrow \frac{1}{\varepsilon}(n-1)\max_{e\in E} \ell(e)$
3 Perform a binary search on the set

$$\mathcal{M} := \left\{a, 2a, 2^2 a, \dots, 2^k a\right\}, \qquad \text{where } k = \left\lceil \log \frac{b}{a} \right\rceil$$

to find the minimum value $K' \in \mathcal{M}$ such that Algorithm 7.2 called with the test parameter K does not return COF.
4 Let T' be the tree generated by Algorithm 7.2 for the test value K' and let t'_e $(e \in T')$ be the corresponding "fine tuned" blend parameters.
5 Define the reduction r by $r(e) := 0$ if e is not included in T and by $r(e) := t'_e$ otherwise.
6 **return** T and r

Theorem 7.8 *For any fixed $\varepsilon > 0$, there is a $(1 + 2/\varepsilon, 1 + \varepsilon)$-approximation algorithm for the problem* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE). *For the first three classes of cost functions discussed on page 7.4 it can be implemented to run in time $\mathcal{O}(\log \frac{nL_{\max}}{\varepsilon} + T_{\text{MST}} \cdot \log\log \frac{nL_{\max}}{\varepsilon})$, where $T_{\text{MST}} \in \mathcal{O}(n + m\beta(m, n))$ is the time required for an MST computation.* $\square$

Additional Restrictions on the Edge Reductions

There are two natural and meaningful additional requirements for a reduction r. The first is *integrality* of r and the second one is the constraint that for all $e \in E$ we have $r(e) \in \{0, \ell(e) - \ell_{\min}(e)\}$. Integer valued reductions are helpful to model discrete steps of improvement, e.g. the addition of a number of communication links parallel to already existing ones in the network. Reductions that take values only from $\{0, \ell(e) - \ell_{\min}(e)\}$ can be used to model the insertion of alternative edges to the graph G, with the reduction of the edge e corresponding to the construction of a new edge e' parallel to e with length $\ell_{\min}(e)$.

We would like to note that our Algorithms 7.1 and 7.3 can be modified easily to handle the case when one of the above restrictions is posed on the reduction. In this case, Step 3 of Algorithm 7.2 is modified in such a way that the minimization is carried out only over the integers in $[0, \ell(e) - \ell_{\min}(e)]$ or on the two element set $\{0, \ell(e) - \ell_{\min}(e)\}$ respectively. In the latter case it is trivial that this minimization can be accomplished in polynomial time. We now discuss the integer case.

If the cost functions are linear or concave, the reduction delivered by our algorithm will already satisfy the additional constraint (under the natural assumptions that the functions ℓ and $\ell_{\min}$ are integral). If the cost functions c_e are convex, for each edge $e \in E$ we first find the (rational) minimizer τ_e of f_e in the interval $[0, \ell(e) - \ell_{\min}(e)]$. By convexity of f_e the integral minimum is attained either at $\lfloor \tau_e \rfloor$ or at $\lceil \tau_e \rceil$. Thus, by inspecting f_e at these two values, we can carry out the minimization in polynomial time. Similar arguments hold for the case of piecewise linear, piecewise concave, and piecewise convex functions.

Approximate Minimization for the Compound Costs

So far, we have assumed that each of the functions $f_e(t) = \ell(e) - t + \frac{K}{B} c_e(t)$ can be minimized *exactly* on the respective compact interval $[0, \ell(e) - \ell_{\min}(e)]$. This indeed is not necessary to obtain an approximation algorithm with a constant factor approximation for (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE). We discuss this in the sequel. Suppose that in Step 3 of Algorithm 7.2 for each edge $e \in E$ we find a value $\tilde{t}_e$ satisfying

$$f_e(\hat{t}_e) \leq \rho \cdot \min_{t \in [0, \ell(e) - \ell_{\min}(e)]} f_e(t)$$

for some $\rho \geq 1$. The compound weight of edge e is then given by $h_K(e) = \ell(e) - \hat{t}_e + \frac{K}{B} c_e(\hat{t}_e)$. We also modify Step 7 to check whether the compound weight of the tree is at most $\rho(1 + \varepsilon)K$.

It turns out that in the case of this modification Corollary 7.6 does not carry over, i.e., we are not able to perform a *binary search* on the interval $\mathcal{I}$. Nevertheless, we can still perform a *linear* search on the the set $\mathcal{M}$ defined in (7.14) in polynomial time, since the size of $\mathcal{M}$ is polynomial. We find the minimum value $K' \in \mathcal{M}$ such that the modified test is successful.

Let $\tilde{K} = \mathrm{OPT}/\varepsilon$. We show that the test via the modified Algorithm 7.2 is successful for $\tilde{K}$. To this end, let r^* be an optimal reduction and $T^* = \mathrm{MST}_G(\ell - r^*)$

a corresponding MST of length OPT. We have

$$\begin{aligned} h_{\tilde{K}}(T^*) &\leq \sum_{e \in T^*} \rho \min_{t \in [0, \ell(e) - \ell_{\min}(e)]} \left(\ell(e) - t + \frac{\tilde{K}}{B} c_e(t) \right) \\ &\leq \rho \sum_{e \in T^*} \left(\ell(e) - r^*(e) + \frac{\tilde{K}}{B} c_e(r^*(e)) \right) \\ &\leq \rho(\mathrm{OPT} + \tilde{K}) \\ &\leq \rho(1 + \varepsilon)\tilde{K}. \end{aligned}$$

We can now conclude that the linear search terminates with a value K' satisfying $K' \leq 2 \cdot \mathrm{OPT}/\varepsilon$. Using the arguments in the proof of Theorem 7.8 it now follows that with the approximate minimization in the compound costs will find a reduction r and a tree T such that

$$(\ell - r)(T) \leq \rho \left(1 + \frac{2}{\varepsilon} \right) \cdot \mathrm{MST}_G(\ell - r^*)$$

and $\sum_{e \in E} c_e(r(e)) \leq \rho(1 + \varepsilon)B$.

The above arguments show that exact minimization of the functions f_e is not necessary. An approximate minimization with multiplicative error $\rho \geq 1$ will simply worsen our performance guarantees by just that factor of ρ with respect to both optimality and constraint violation.

7.5 Extension to Steiner Trees and Other Networks

The techniques used to obtain results for the problem (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) in Section 7.4 are quite general. They extend to more general network improvement problems. In fact, given *any* ρ-approximation algorithm for finding a subgraph $S \in \Gamma$ minimizing $\ell(S)$ (where ℓ is an edge-weight function on the edges of G), the method allows us to obtain a $((1 + 2/\varepsilon)\rho, (1 + \varepsilon)\rho)$-approximation for the corresponding edge improvement problem. For example, using this technique in conjunction with the results of Goemans et al. [GW92, GG+94], we get the first approximation results for edge based improvement problems such as finding minimum cost generalized Steiner trees, minimum k-edge connected subgraphs, or other network design problems specified by weakly supermodular functions. In this section we briefly illustrate the modifications of our algorithm for this case. The notations and definitions used are taken from [GW92].

Many network design problems can be formulated as cut-covering problems [GW92, GG+94]: For a certain family of cuts in a graph the problem is to find a minimum cost subgraph intersecting all the cuts in the family. Formally, let $G = (V, E)$ be a graph with nonnegative edge weights $\ell(e)$, $e \in E$. Let $f : 2^V \to \{0, 1\}$ be a function and consider the following integer program:

$$\begin{aligned} \text{(IP)} \quad \min \sum_{e \in E} \ell(e) \cdot x_e & \\ x(\delta(U)) \geq f(U), \qquad & \varnothing \neq U \subset V \\ x_e \in \{0, 1\}, \qquad & e \in E, \end{aligned}$$

where $\delta(U)$ denotes the set of edges having exactly one endpoint in U and $x(F) = \sum_{e \in F} x_e$. The problem (IP) is termed a *constrained forest problem*. A function f is called *proper*, if it obeys the following properties:

(i) **Symmetry**: $f(U) = f(V - U)$ for all $U \subseteq V$; and

(ii) **Disjointness**: If A and B are disjoint, then $f(A) = f(B) = 0$ implies that $f(A \cup B) = 0$.

It is also assumed that $f(V) = 0$. Many interesting families of forests can be modeled with the help of (IP). Among those are spanning trees, Steiner trees, generalized Steiner trees and T-joins. We refer to [GW92] for a comprehensive list of problems.

Theorem 7.9 ([GW92]) *For any proper function f there exists an approximation algorithm for the constrained forest problem (IP) with a performance of $2 - 2/m$, where m denotes the number of edges in the graph. This algorithm can be implemented to run in time $\mathcal{O}(\min\{n^2 \log n, mn\alpha(m, n)\})$, where α is the inverse of Ackerman's function.* □

Let Π be any proper constrained forest problem. Let Γ be the subgraph class specified by (IP) in conjunction with the proper function f. Also, let A be the approximation algorithm mentioned in Theorem 7.9 with performance $2 - 2/m$.

We modify Step 5 of our test procedure in Algorithm 7.2 on page 141 as follows: We call A to compute an approximate solution to the problem of finding a minimum cost subgraph from Γ using the weight $h_K(e)$ for each $e \in E$. We also modify Step 7 to check whether the compound weight of the subgraph is bounded from above by $(2 - 2/m) \cdot (1 + \varepsilon)K$.

Then, the proof of Theorem 7.8 carries over with the additional factor $2 - 2/m$ in the performance for the budget as well as the cost of the subgraph. The arguments are similar to the ones given on page 148 for the approximate minimization of functions f_e. Thus, using Algorithm 7.3 for any fixed $\varepsilon > 0$ we obtain a $((2-2/m)(1+2/\varepsilon),(2-2/m)(1+\varepsilon))$-approximation algorithm for the problem (TOTAL WEIGHT, EDGE IMPROVEMENT COST, Γ). Particularly, in the special case of Steiner trees this gives us the following theorem.

Theorem 7.10 *For any fixed $\varepsilon > 0$, there exists a $((2-2/m)(1+2/\varepsilon),(2-2/m)(1+\varepsilon))$-approximation algorithm for the problem* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE). *For the first three classes of cost functions discussed on page 7.4 this algorithm can be implemented to have a running time of $\mathcal{O}(n^2 \log n \cdot \log \frac{nL_{\max}}{\varepsilon})$.* □

Following the same lines as above we obtain the following general result.

Theorem 7.11 *Suppose there is a polynomial time ρ-approximation algorithm for a unicriterion minimization problem Π, consisting of finding a minimum total edge cost subgraph $S \in \Gamma$ on an edge weighted graph $G = (V, E)$. Then for all $\varepsilon > 0$, there is a polynomial time $(\rho \cdot (1+2/\varepsilon), \rho \cdot (1+\varepsilon))$-approximation algorithm for the corresponding edge improvement problem* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, Γ). □

7.6 An Improved Algorithm for Linear Reduction Costs

In this section we will show how to improve the performance and the running time of the approximation algorithm from Section 7.4 in the case that the reduction costs on the edges are *linear*, i.e. that $c_e(t) = c_e t$ for $e \in E$.

The first observation for the improved algorithm is the following: In Step 3 of Algorithm 7.2 the linear function $f_e(t) := \ell(e) + t(\frac{K}{B}c_e - 1)$ is minimized on the interval $[0, \Delta(e)]$, where $\Delta(e) := \ell(e) - \ell_{\min}(e)$. At which of the two endpoints of the interval the minimum is attained depends solely on the factor $\frac{K}{B}c_e - 1$. If $\frac{K}{B}c_e - 1 \leq 0$, i.e., if $K \leq B/c_e$, then f_e attains its minimum at $\Delta(e)$. Otherwise, f_e is minimized at 0.

The Structure of the Compound Weights

Observe that the h_K-weight of an edge e is given by

$$h_K(e) := \begin{cases} \ell_{\min}(e) + K\frac{(\ell(e)-\ell_{\min}(e))c_e}{B} & \text{if } K < B/c_e, \\ \ell(e) & \text{if } K \geq B/c_e. \end{cases} \tag{7.16}$$

Thus, for each edge e, the compound weight $h_K(e)$ viewed as a function of K is a piecewise linear function with exactly one breakpoint at B/c_e. For $K \geq B/c_e$, the function has the constant value $\ell(e)$, while for $K < B/c_e$ it has slope $(\ell(e) - \ell_{\min}(e))c_e/B$.

If we plot the compound weight $h_K(e)$ for each edge $e \in E$, for increasing K we get a linear function with exactly one breakpoint. This breakpoint is at B/c_e. Figure 7.3 shows an example of graphs of these compound weights.

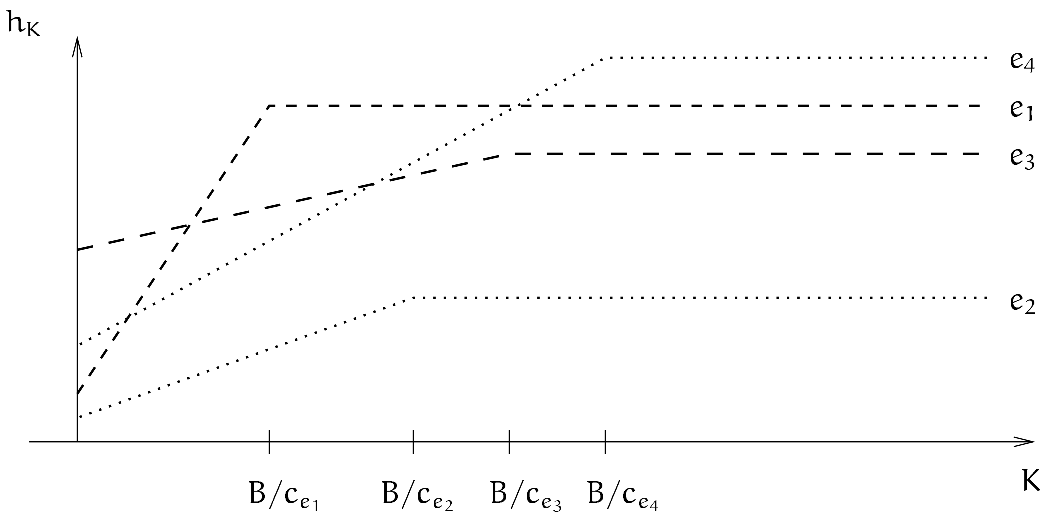


Figure 7.3: Compound weight h_K of edges for increasing K.

It is easy to see that, given two edges e and e', their ordering with respect to the compound weights h_K changes at most twice when K varies. Also, these at most two values of K can be computed in constant time.

Outline of the Improved Algorithm

Let $K^* \in \mathcal{I}$ be the overall minimum value[3] such that Algorithm 7.2 would return a tree T (and not a COF) if called with $K = K^*$. We can use the analysis from Section 7.4 to show that $K^* \leq \mathrm{OPT}/\varepsilon$, where OPT again denotes the length of an optimal reduced tree for a budget of B.

We now have the following important lemma:

[3]The compound weights $h_K(e)$ $(e \in E)$ as functions of K are continuous. Since the graph G contains only a finite number of different spanning trees, $\mathrm{MST}_G(h_K)$ (as a function of K) is continuous and hence $\min\{\, K \in \mathcal{I} : \mathrm{MST}_G(h_K) \leq (1+\varepsilon)K \,\}$ is well defined.

Lemma 7.12 *If the ordering of the edges with respect to their* h_{K^*}*-weights is known, we can construct a tree* T *and a reduction* r *in time* $\mathcal{O}(n + m\beta(m,n))$ *with the following properties:*

(i) The cost $\sum_{e \in E} c_e r(e)$ *of the reduction* r *is at most* $(1+\varepsilon)B$.

(ii) The weight $(\ell - r)(T)$ *in the modified graph is no more than* $(1+1/\varepsilon)\mathrm{OPT}$.

Here, $\beta(m,n)$ *is defined in* (7.13) *(see also page 12).*

Proof: Given the ordering of the edges according to their weights, we can use the minimum spanning tree algorithm of Gabow et al. [GG+86] to compute a minimum spanning tree with respect to the h_{K^*}-weights, without actually knowing these weights. The ordering suffices for this purpose.

Observe that, if we knew the h_{K^*}-weights of the edges in the tree T, we could now construct a reduction r' just as in Step 3 of Algorithm 7.1. Using exactly the same arguments as in the proof of Theorem 7.4 but now using that $K^* \leq \mathrm{OPT}/\varepsilon$ instead of $K' \leq \mathrm{OPT}/\varepsilon + \varepsilon'$, it then follows that

$$(\ell - r')(T) \leq (1 + 1/\varepsilon)\mathrm{OPT},$$

and that the cost of the reduction r' is at most $(1+\varepsilon)B$.

But by the assumption of the lemma, we only have knowledge about the *ordering* of the edges and *not* about K^* or h_{K^*}. Nevertheless, this is not grave. Recall from Section 7.3 that, given a tree and a budget, we can construct an optimal reduction on the tree for that budget in $\mathcal{O}(n)$ time by a Greedy-type algorithm that repeatedly reduces the length of the cheapest edge until the budget is exhausted. Thus, if we compute such a reduction r on our tree T with the budget set to $(1+\varepsilon)B$, the length of T under $\ell - r$ will be at most $(\ell - r')(T)$, which in turn is bounded from above by $(1 + 1/\varepsilon)\mathrm{OPT}$. □

Lemma 7.12 suggests finding an ordering of the edges in the graph according to their compound weight at K^*. As in Section 5.6 this can be accomplished efficiently using the elegant trick of Megiddo [Meg83].

Basically we wish to sort the set $S := \{h_{K^*}(e_1), \ldots, h_{K^*}(e_m)\}$ where K^* is not known. However, for any K we can decide whether $K^* \leq K$ or $K^* > K$ by one MST computation: We compute an MST with respect to edge weights given by h_K and compare its weight to $(1+\varepsilon)K$. If the weight is bounded from above by $(1+\varepsilon)K$, then we know that $K \leq K^*$. Otherwise, we can conclude that $K^* > K$.

The sorting of the set S now works the same way as in Section 5.6 on page 71 with the difference that the ordering of two edges can change at most twice instead of just once. By $\mathcal{O}(\log^2 n)$ minimum spanning tree computations plus an overhead of $\mathcal{O}(m \log n)$ elementary operations we can find the ordering of the edges with respect to h_{K^*}. Lemma 7.12 then enables us to find a $(1+1/\varepsilon, 1+\varepsilon)$-approximation.

Theorem 7.13 *For any fixed* $\varepsilon > 0$ *the modified algorithm is a* $(1+1/\varepsilon, 1+\varepsilon)$*-approximation algorithm for* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *with a running time of* $\mathcal{O}(n\log^2 n + m\log^2 n\beta(n,m))$*, where* $\beta(m,n) = \min\{\, i : \log^{(i)} n \le m/n\,\}$. □

7.7 Algorithms for Treewidth Bounded Graphs and Linear Costs

In this section we obtain an improved algorithm for the class of treewidth bounded graphs when the reduction costs on the edges are linear. The basic idea behind the algorithm in this section is to reduce the problem of improving the tree to some appropriately chosen bicriteria problem. The bicriteria problem of our choice will be the problem (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE) from Example 3.3. To this end we recall the following result from [MR+95a].

Theorem 7.14

1. *For any fixed* $\varepsilon > 0$*, there exists a* $(1+1/\varepsilon, 1+\varepsilon)$*-approximation algorithm for the* (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE) *problem.*
2. *For the class of treewidth-bounded graphs, for any fixed* $\varepsilon > 0$*, there is a* $(1+\varepsilon, 1+\varepsilon)$*-approximation algorithm for* (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE). □

We will use the second part of the theorem to obtain an improved approximation, namely a performance of $(1+\varepsilon, (1+\varepsilon)^2)$ for any fixed $\varepsilon > 0$.

We note here that using first part of Theorem 7.14 instead, we could also construct a $(1+1/\varepsilon, (1+\varepsilon)^2)$-approximation algorithm for (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) on general graphs for any fixed $\varepsilon > 0$, if we restrict ourselves to linear reduction costs. Since our algorithm from Section 7.6 already gives us a performance of $(1+1/\varepsilon, 1+\varepsilon)$ and the algorithm from Section 7.4 has the same performance (plus the additive constant of ε' which can be made arbitrarily small) for far more general classes of cost functions this is not as interesting. Also, our approximation algorithms from Sections 7.4 and 7.6 do not need any additional space, while the construction presented below transforms the original graph into a graph having more nodes and edges.

We now describe our improved approximation algorithm for linear reduction costs on treewidth-bounded graphs. First, we transform the original graph to another graph that can be fed into the algorithm from Theorem 7.14. To this end, we replace each edge $e = (u, v)$ of the original graph by a certain subgraph in such a way that the treewidth does not increase. This is done as follows.

For each edge $e = (u, v)$ in the graph let b_e be chosen so that

$$(1+\varepsilon)^{b_e} \le \ell(e) - \ell_{\min}(e) \le (1+\varepsilon)^{b_e+1}.$$

We remove e from G and add $b_e + 2$ new vertices w_k^e, $k = -1, 0, \ldots, b_e$. For $-1 \le k \le b_e$, we join w_k^e to both u and v. For $k \ge 0$, the edge (u, w_k^e) has c-cost $(1+\varepsilon)^k c_e$ and l-cost $d(u, w_k^e) := \ell(e) - (1+\varepsilon)^k$, while the edge (u, w_{-1}^e) has c-cost 0 and d-cost $\ell(e)$. All the edges (w_k^e, v) have their c-cost and l-cost set to zero. The transformation procedure is shown in Algorithm 7.4 on the following page and an example of a transformation is displayed in Figure 7.4 on the next page.

Let G be the original graph and G' be the graph obtained as a result of the transformation. Also, let $\mathrm{tw}(G)$ and $\mathrm{tw}(G')$ denote the treewidths of G and G' respectively. We have the following observation.

Observation 7.15 *Whenever* $\mathrm{tw}(G) \ge 2$*, we have that* $\mathrm{tw}(G') \le \mathrm{tw}(G)$.

Proof: Let $D = (S, T)$ be a tree-decomposition of G with $S = \{X_i : i \in I\}$ and $T = (I, F)$. For each edge $e = (u, v)$ in G we do the following. There must be a set $X_i \in S$ which contains both u and v. Add $b_e + 2$ sets $X_j^e := \{u, v, w_j^e\}$ $(j = -1, 0, \ldots, b_e)$ to S and include the edges (X_i, X_j^e) in T. It is straightforward to verify that this way we obtain a tree-decomposition of G' of width at most $\max\{2, \mathrm{tw}(G)\}$. □

Correctness and Performance Guarantee

Let r^* denote the optimal reduction involving a cost of at most B, let T^* be a minimum spanning tree in G with respect to edge-lengths given by $\ell - r^*$ and let $\mathrm{OPT} := \mathrm{MST}_G(\ell - r^*)$ be its weight in the modified graph.

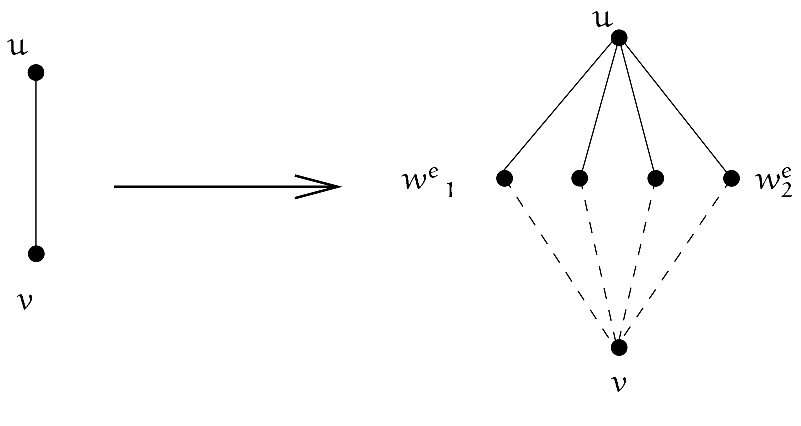
Theorem 7.16 *For the class of treewidth bounded graphs and linear reduction costs the following statement holds: For all fixed* $\varepsilon > 0$*, Algorithm 7.5 is a* $(1+\varepsilon, (1+\varepsilon)^2)$*-approximation algorithm for* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE).

Proof: Consider the optimal tree $T^* = \mathrm{MST}_G(\ell - r^*)$ in G. We can define a tree $\tilde{T}$ in G' in the following way: For an edge $e = (u, v) \in T^*$ that is reduced by

Algorithm 7.4 Transformation procedure for treewidth bounded graphs.

Input: A graph $G = (V, E)$, two edge weight functions ℓ, $\ell_{\min}$, linear cost functions $c_e(t) = c_e \cdot t$ $(e \in E)$, a number B; a *fixed* accuracy parameter $\varepsilon > 0$

$V' \leftarrow V$ {The vertex set of the graph to be constructed}
$E' \leftarrow \varnothing$ {The edge set of the graph to be constructed}
for all edges $e = (u, v) \in G$ **do**
  $b_e \leftarrow \lceil \log_{(1+\varepsilon)}(\ell(e) - \ell_{\min}(e)) \rceil$
  {notice that $(1+\varepsilon)^{b_e} \leq \ell(e) - \ell_{\min}(e) \leq (1+\varepsilon)^{b_e+1}$}
  Choose $b_e + 2$ new vertices w^e_k, $k = -1, 0, \ldots, b_e$
  $E' \leftarrow E' \cup \{(u, w^e_{-1}), (w^e_{-1}, v)\}$
  $c(u, w^e_{-1}) \leftarrow 0$
  $l(u, w^e_{-1}) \leftarrow \ell(e)$
  $c(w^e_{-1}, v) \leftarrow 0$
  $l(w^e_{-1}, v) \leftarrow 0$
  for $k \leftarrow 0, \ldots, b_e$ **do**
    $E' \leftarrow E' \cup \{(w^e_k, v), (w^e_k, u)\}$
    $c(u, w^e_k) \leftarrow (1+\varepsilon)^k c_e$
    $l(u, w^e_k) \leftarrow \ell(e) - (1+\varepsilon)^k$
    $c(w^e_k, v) \leftarrow 0$
    $l(w^e_k, v) \leftarrow 0$
  end for
end for
return $G' = (V', E')$, c, l



G

G' for $b_e = 2$

Figure 7.4: Example for the transformation on treewidth bounded graphs.

Algorithm 7.5 Algorithm for treewidth bounded graphs.

Input: A graph $G = (V, E)$, two edge weight functions ℓ, $\ell_{\min}$,
linear cost functions $c_e(t) = c_e \cdot t$ $(e \in E)$, a number B;
a *fixed* accuracy parameter $\varepsilon > 0$

1 Call Algorithm 7.4 to obtain a new graph G' with two edge weight functions c and l.
2 $B' \leftarrow (1 + \varepsilon)B$
3 Apply the algorithm referred to in Part 2 of Theorem 7.14 with accuracy parameter set to ε to the following instance of (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE):
The graph is G' with the two edge weight functions l and c. The bound on the c-weight of the tree is set to B'.
4 Let T' be the tree generated in Step 3.
5 $E_T \leftarrow \varnothing$ {the edge set of the spanning tree in G to be constructed}
6 **for all** edges $e = (u, v) \in G$ **do**
7   Let $w^e_{-1}, w^e_0, \ldots, w^e_{b_e}$ be the new vertices corresponding to the edge e generated by the transformation procedure Algorithm 7.4.
8   **if** there exists an index k such that both $(u, w^e_k) \in T'$ and $(w^e_k, v) \in T'$ **then**
9     $E_T \leftarrow E_T \cup \{(u, v)\}$
10     $k' = \min\{ k : (u, w^e_k) \in T' \text{ and } (w^e_k, v) \in T' \}$
11   **if** $k' = -1$ **then**
12       $r(e) \leftarrow 0$
13   **else**
14       $r(e) \leftarrow (1 + \varepsilon)^{k'}$
15   **end if**
16   **end if**
17 **end for**
18 **return** $T = (V, E_T)$ and r

$r^*(e)$ we select an edge (u, r_j) in G' of l-cost $\ell(e) - t(e)$, where $t(e)$ is chosen in such a way that

$$\frac{t(e)}{(1+\varepsilon)} \leq r^*(e) \leq t(e).$$

Observe that the edge $e' = (u, r_j)$ selected in the above fashion satisfies the following two conditions:

$$\ell(e) - (1+\varepsilon)r^*(e) \leq l(e') \leq \ell(e) - r^*(e) \tag{7.17}$$
$$c_e r^*(e) \leq c(e') \leq (1+\varepsilon)r^*(e). \tag{7.18}$$

We also add all the edges (r_i, v), $i = -1, \dots, b_e$ to belong to $\tilde{T}$. Notice that all these edges have zero l-costs and c-costs.

From (7.17) and (7.18) we can conclude that $l(\tilde{T}) \leq$ OPT and $c(\tilde{T}) \leq (1+\varepsilon)B = B'$. In particular, $\tilde{T}$ is a feasible solution for the instance of (TOTAL WEIGHT, TOTAL WEIGHT, SPANNING TREE) constructed in Step 3 of Algorithm 7.5. By the performance of the algorithm referred to in Part 2 of Theorem 7.14 we see that the tree T' found in Step 3 must satisfy

$$l(T') \leq (1+\varepsilon)l(\tilde{T}) \leq (1+\varepsilon) \cdot \text{OPT} \tag{7.19}$$
$$c(T') \leq (1+\varepsilon)B' = (1+\varepsilon)^2 B. \tag{7.20}$$

Now consider the construction of the reduction r and the tree $T = (V, E_T)$ in the **for**-loop starting at Step 6. It is straightforward to verify that at the end of the loop $T = (V, E_T)$ is indeed a spanning tree of the original graph G.

Clearly, the reduction r computed in Steps 12 and Steps 14 involves a cost that is bounded above by $l(T')$ which by (7.20) in turn is at most $(1+\varepsilon)^2 B$. Moreover, for each edge $e = (u, v)$ that is added to E_T in Step 9 we have

$$\ell(e) - r(e) \leq \max\{\, l(u, w_k^e) : \text{ both edges } (u, w_k^e) \text{ and } (w_k^e, v) \text{ belong to } T' \}.$$

Hence $(\ell - r)(T) \leq l(T')$, which by (7.19) is at most $(1+\varepsilon)$OPT. This completes the proof of the performance guarantee.

We now show that the algorithm can be implemented to run in polynomial time. For this, observe that for a fixed value of $\varepsilon > 0$, the number of edges added (i.e., the value of $2\sum_{e \in E}(b_e + 2)$) is polynomial in the size of the input. For fixed $\varepsilon > 0$, Algorithm 7.4 runs in time $\mathcal{O}(m \log L_{\max})$, where $L_{\max} = \max_{e \in E} \ell(e)$. Clearly, all the remaining steps in Algorithm 7.5 can be carried out in polynomial time. □

7.8 Hardness Results

In this section we will prove the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) problem to be hard even for very restricted class of graphs and the most simple reduction cost functions. We will also establish hardness results for the corresponding Steiner tree improvement problem.

Theorem 7.17 *The* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *problem is* NP*-hard, even when restricted to series-parallel graphs and even when all the reduction cost functions* c_e *are linear, i.e.,* $c_e(t) = c_e \cdot t$ *for* $e \in E$.

Proof: We use a reduction from CONTINUOUS MULTIPLE CHOICE KNAPSACK which is known to be a NP-complete problem (c.f. [GJ79, MP11]). An instance of CMC-KNAPSACK is given by a finite set U of n items, a size $s(u)$ and value $v(u)$ for each item, a partition $U_1 \cup \cdots \cup U_k$ of U into disjoint sets and two integers S and K. The question is, whether there is a choice of a unique element $u_i \in U_i$, for each $1 \leq i \leq k$, and an assignment of rational numbers r_i, $0 \leq r_i \leq 1$ to these elements such that $\sum_{i=1}^{k} r_i s(u_i) \leq S$ and $\sum_{i=1}^{k} r_i v(u_i) \geq K$.

Given an instance of CMC-KNAPSACK we construct a graph $G = (V, E)$ in the following way: We let $V = U \cup \{x, t, t_1, \ldots, t_k\}$, $E := E_1 \cup E_2 \cup E_3$ with $E_1 := \{(x, u) : u \in U\}$, $E_2 := \{(u, t_i) : u \in U_i, i = 1, \ldots, k\}$ and $E_3 := \{(t_i, t) : i = 1, \ldots, k\}$. The graph constructed this way is obviously series-parallel with terminals x and t.

Define $D := \max\{v(u) : u \in U\}$. For each edge $(x, u) \in E_1$, let $\ell(x, u) := D$, $\ell_{\min}(x, u) := D - v(u)$, $c(x, u) := s(u)/v(u)$. For all edges $e \in E_2$ we let $\ell(e) := \ell_{\min}(e) := c_e := 0$, and for all edges $e \in E_3$ we define $\ell(e) := \ell_{\min}(e) := 3D$ and $c_e := 0$. Set the bound B on the total cost to be S.

The graph is shown in Figure 7.5 on the following page. The dotted edges are of weight 0 while the dashed ones have weight $3D$. Any MST in G has ℓ-weight $kD + 3D$.

By construction, any feasible reduction can only reduce the length of the edges in E_1. Assume that r is a feasible reduction. Observe that the MST in G with edge lengths given by $\ell - r$ will always include *all* edges from E_2 (which are of weight 0) and *exactly one* edge from E_3, regardless of which edges from E_1 are affected by the reduction. Observe also that for any fixed $i \in \{1, \ldots, k\}$, any MST in the modified graph will contain *exactly one* of the edges of the form (x, u'), where $u' \in U_i$. Consequently, reducing the length of more than one edge (x, u') with $u' \in U_i$ will *not* improve the quality of the solution, but cost money from the budget B. We thus have:

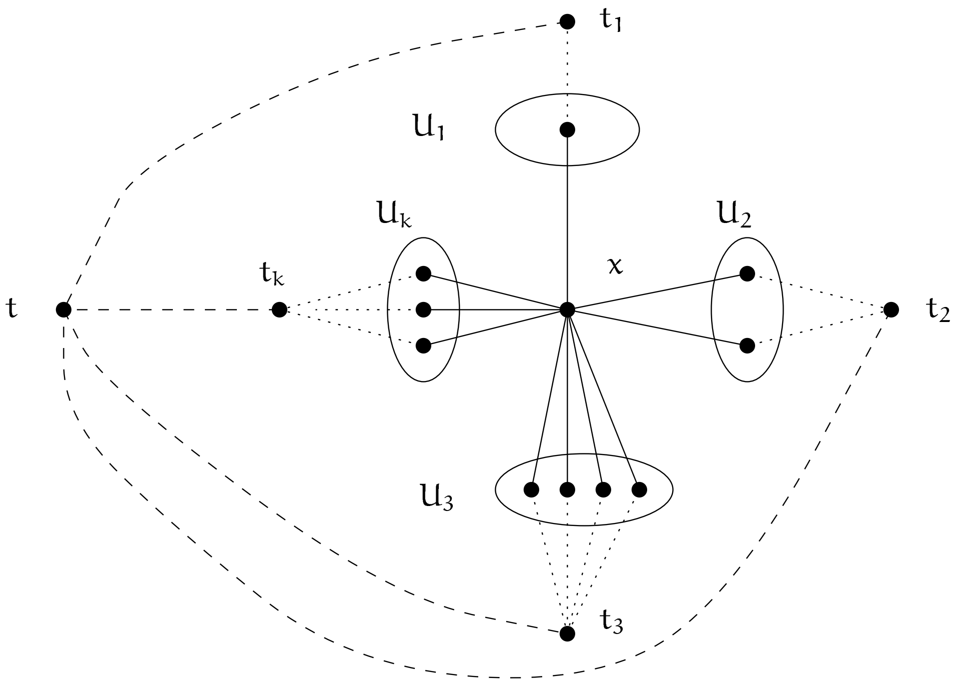


Figure 7.5: Graph used in the reduction from CMC-KNAPSACK.

Observation: *If* r *is a feasible reduction for the instance of* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) *defined above and the weight of an* MST *in the modified graph is* Y*, then there is always a feasible reduction* r'*, which for each* $i \in \{1, \ldots, k\}$ *reduces at most one of the edges* (x, u)*,* $u \in U_i$ *and the weight of an* MST *with respect to* $\ell - r'$ *is also equal to* Y*.*

Let r be any reduction as defined in the above observation and for $i = 1, \ldots, k$ let $e_i = (x, u_i)$ be the unique edge from x to U_i affected by the reduction. The weight of an MST in with respect to $\ell - r$ is then given by

$$3D + \sum_{i=1}^{k} (\ell(e_i) - r(e_i)) = 3D + k \cdot D - \sum_{i=1}^{k} r(e_i). \tag{7.21}$$

The cost of reduction r is given by

$$\sum_{i=1}^{k} r(e_i) c_{e_i} = \sum_{i=1}^{k} r(e_i) \cdot \frac{s(u_i)}{v(u_i)} = \sum_{i=1}^{k} \frac{r(e_i)}{v(u_i)} \cdot s(u_i) \leq B. \tag{7.22}$$

We now prove the following: There is a feasible reduction r such that $\mathrm{MST}_G(\ell - r) \leq (3 + k)D - K$, if and only if there exists a choice of a unique element

$u_i \in U_i$, $1 \leq i \leq k$ and an assignment of rational numbers $r_i, 0 \leq r_i \leq 1$ to these elements such that $\sum_{i=1}^{k} r_i s(u_i) \leq B$ and $\sum_{i=1}^{k} r_i v(u_i) \geq K$.

First, assume that there is a feasible reduction r such that $\mathrm{MST}_G(\ell - r) \leq (3 + k)D - K$. Without loss of generality, we can assume that r has the properties as stated in the above observation. Then for $i = 1, \dots, k$ there is at most one edge $e_i = (x, u)$ with $u \in U_i$ such that $r(e_i) > 0$. If there is such an edge e_i, we define

$$r_i := \frac{r(e_i)}{v(u_i)} = \frac{r(e_i)}{\ell(e_i) - \ell_{\min}(e_i)} \tag{7.23}$$

and let $u_i := u$. If for all edges (x, u) with $u \in U_i$ we have $r(e_i) = 0$, we simply let $r_i := 0$ and choose $u_i \in U$ arbitrarily. It follows readily from the definition and the feasibility of the reduction r that $r_i \in [0, 1]$. Moreover, using Equation (7.22) we see that $\sum_{i=1}^{n} r_i s(u_i) \leq B \leq S$. Using equation (7.21) and the fact that the weight $\mathrm{MST}_G(\ell - r)$ is no more than $(3 + k)D - K$ we obtain that

$$\sum_{i=1}^{n} r_i v(u_i) = \sum_{i=1}^{n} \frac{r(e_i)}{v(u_i)} \cdot v(u_i) = \sum_{i=1}^{n} r(e_i) \geq K.$$

Conversely, if we can pick unique elements u_i from the sets U_i and find rational numbers $r_i \in [0, 1]$ such that $\sum_{i=1}^{n} r_i s(u_i) \leq B$ and $\sum_{i=1}^{n} r_i v(u_i) \geq K$. We can define a reduction r by $r(x, u_i) := r_i v(u_i) = r_i(\ell(x, u_i) - \ell_{\min}(x, u_i))$ for $i = 1, \dots, k$ and $r(e) := 0$ for all other edges. It follows that r is indeed feasible, and using equation (7.21) we see that the MST in the modified graph is no heavier than $(3 + k)D - K$. □

Theorem 7.18 *Unless* $\mathsf{P} = \mathsf{NP}$*, for any* $\alpha > 1$*, there is no polynomial time* $(\alpha, 1)$ *approximation for the* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE) *problem even when restricted to bipartite graphs.*

Proof: Suppose there is a $(\alpha, 1)$-approximation algorithm A for the problem for some $\alpha > 1$. We will show that A can be used to solve an arbitrary instance of SET COVER. Given an instance of $(Q, \mathcal{F}, k)$ of SET COVER, we construct the natural bipartite graph with one partition for set nodes (denoted by Q_1, Q_2, $\dots$, Q_m) and the other for element nodes (denoted by q_1, q_2, $\dots$, q_n), and edges representing element inclusion in the sets. To this bipartite graph, we add an "enforcer" node (denoted by x) which is adjacent to each of the set nodes. Let G denote the resulting bipartite graph. The set K of terminals for the Steiner tree instance is given by $K = \{x, q_1, q_2, \dots, q_n\}$. For each edge e in G, we set $\ell(e) = 1$ and $\ell_{\min}(e) = \varepsilon$, where ε is a positive quantity chosen so that

$$\varepsilon < \frac{1}{(\alpha - 1)(n + k)}. \tag{7.24}$$

For each edge e the reduction cost function c_e is given by $c_e(t) = t/(1-\varepsilon)$ for $t \geq 0$. To complete the construction, we set $B := n + k$.

Suppose there is a set cover $Q' = \{Q_{i_1}, Q_{i_2}, \ldots, Q_{i_k}\}$ of size k. Consider the Steiner tree T in G consisting of x, the edges (x, Q_{i_j}), $1 \leq j \leq k$, and one edge from each element node to some set node in Q'. (Since Q' is a cover, each element node must be adjacent to some set node in Q'.) Let r be the reduction defined by $r(e) = 1 - \varepsilon$ if $e \in T$ and $r(e) = 0$ otherwise. The total cost of r is $n + k$ and the total length of T in the modified graph is $(n + k)\varepsilon$. Thus, if there is a set cover of size k, then the modified graph has a Steiner tree of length at most $(n + k)\varepsilon$. Since A is a $(\alpha, 1)$-approximation algorithm, the length of a Steiner tree returned by A is at most $\alpha(n + k)\varepsilon$.

Conversely, assume that there is no set cover of size at most k. Then at least $k + 1$ sets are needed to cover the elements. We claim that in this case, any Steiner tree T that connects together the $n + 1$ nodes in K must have at least $n + k + 1$ edges. To see this, let T be an arbitrary Steiner tree. Suppose that there exists a set node Q_j with $(x, Q_j) \notin T$ such that the addition of (x, Q_j) to T induces a cycle. By the bipartiteness, this cycle must contain an edge (Q_j, q'), where q' is an element node. Replace (Q_j, q') by (x, Q_j). We repeat the above steps as long as possible. Clearly, this "algorithm" must terminate, since in every replacement step the number of set nodes adjacent to x in T increases by one. At the end of the above steps every element node is adjacent to *exactly* one set node in T, since otherwise our procedure would not have stopped. Thus, T contains exactly n edges of the form (Q_j, q_i). Moreover, for each set node q_i the unique element node Q_j with $(Q_j, q_i) \in T$ must be adjacent to x. Otherwise, T would not be connected. Hence, the sets Q_j such that $(x, Q_j) \in T$ form a set cover. It now follows that T contains at least $k + 1$ edges of the form (x, Q_j).

We have seen above that, if there is no set cover of size at most k, then any Steiner tree T in G has at least $n + k + 1$ edges. Since for t units of the budget the weight of T can be reduced by at most $(1 - \varepsilon)t$ units, it follows that after modifying the lengths for a budget of $n+k$ the weight of T is at least $1+(n+k)\varepsilon$. Using Equation (7.24), it can be verified that $\alpha(n + k)\varepsilon < 1 + (n + k)\varepsilon$.

Therefore, using A, we can solve an arbitrary instance of SET COVER in polynomial time, contradicting the assumption that $\mathsf{P} \neq \mathsf{NP}$. □

The following symmetric non-approximability result is a direct consequence of the fact that the Steiner tree problem is NP-hard even for bipartite graphs [GJ79, Problem ND12].

Observation 7.19 *Unless* $\mathsf{P} = \mathsf{NP}$*, for any* $\beta > 1$*, there is no polynomial time* $(1, \beta)$ *approximation algorithm for* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE) *even when restricted to bipartite graphs.* □

8

Edge Improvement with Focus on Constrained Steiner Trees

8.1 Introduction

In this chapter we are going to study generalizations of the problems tackled in Chapter 7. We investigate network edge improvement problems where the goal is to find a reduction strategy such that the weight of a constrained Steiner tree in the modified network is as small as possible. In the course of our studies, we consider two constraints: a diameter constraint and a degree constraint.

In the *diameter constrained variant*, we are given additionally d-weights on the edges of the graph. These weights are independent of the ℓ-lengths in the network and can not be changed by a reduction. The goal becomes to find a reduction and a Steiner tree obeying the diameter constraint which is as light as possible after applying the reduction. For the *degree constrained variant*, we are given a bound on the maximum degree of a Steiner tree. The problem consists of finding a reduction and a tree obeying the degree bound such that the weight of the tree is minimal after applying the reduction.

The additional constraints make the above problems far more difficult to cope with. The performance guarantees for our algorithms in this chapter are all of the form $(\mathcal{O}(\log n), \mathcal{O}(\log n), \mathcal{O}(\log n))$, where n denotes the number of nodes in the graph. In contrast, our approximation algorithms from Chapter 7 had $(\mathcal{O}(1), \mathcal{O}(1))$ performances with small constants even for the case of a Steiner tree, i.e., for the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE) problem.

The chapter is organized as follows. After giving formal definitions of the improvement problems in Section 8.2, we present our approximation algorithm for diameter constrained Steiner trees in Section 8.3. Section 8.4 contains the algorithm for improvement with a constraint on the maximum degree of a Steiner tree. In Section 8.5 we comment on the hardness of the problems in this chapter.

8.2 Preliminaries and Problem Definitions

The problems studied in this chapter are based on the edge based upgrading model introduced in Chapter 7. In this chapter we will restrict ourselves to the case when the reduction cost functions c_e on the edges are linear, i.e., $c_e(t) = c_e \cdot t$ for all $e \in E$. Thus, we have the following basic situation:

Assumption 8.1 *Let $G = (V, E)$ be an undirected graph. Associated with each edge $e \in E$, there are three nonnegative values: the current length of edge e, $\ell(e)$, the lower bound for the edge improvement on e, $\ell_{\min}(e)$ and the value c_e telling how expensive it is to shorten e.*

Let d be an additional edge weight function d which is independent of ℓ. The *diameter* $\text{dia}_d(T)$ of a tree T under d is the length of a longest simple path in T with respect to d. We are now ready to define the first of the two problems studied in this chapter.

Definition 8.2 (Diameter Constrained Steiner Tree Problem)
Given an edge-weighted graph G as in Assumption 8.1 with an additional edge weight function d, a set K of terminals, a bound D on the d-diameter and a nonnegative budget value B, the ***diameter and budget constrained minimum total cost Steiner tree problem****,* (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE)*, is to find a subgraph from the set*

$$\left\{ (T, r) : \begin{array}{l} T \textit{ is a Steiner tree of } G \textit{ with the terminals } K, \\ r \textit{ is a reduction on } G, \\ \text{dia}_d(T) \le D \textit{ and } \sum_{e \in T} c_e \cdot r(e) \le B \end{array} \right\}, \tag{8.1}$$

such that $(\ell - r)(T)$ is as small as possible.

The *degree* of a tree T, denoted by $\deg(T)$ is defined to be the maximum degree of a vertex in the tree T, that is $\deg(T) := \max_{v \in T} \deg_T(v)$.

Definition 8.3 (Degree Constrained Steiner Tree Problem)
Given an edge-weighted graph G as in Assumption 8.1, a set K of terminals, a bound b on the degree of T and a nonnegative budget value B, the ***degree and budget constrained minimum total cost Steiner tree problem****,* (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE)*, is to find a subgraph from the set*

$$\left\{ (T, r) : \begin{array}{l} T \textit{ is a Steiner tree of } G \textit{ with the terminals } K, \\ r \textit{ is a reduction on } T, \\ \deg(T) \le b \textit{ and } \sum_{e \in T} c_e \cdot r(e) \le B \end{array} \right\},$$

such that $(\ell - r)(T)$ is as small as possible.

It follows easily from the NP-hardness of the classical Steiner tree problem [GJ79, Problem ND12] that both of the above defined problems are NP-hard. Moreover, even if one is already given an optimal reduction r^* for one of the problems, it remains hard to find a corresponding optimal tree in the modified network. This contrasts to the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) problem investigated in Chapter 7. There, given an optimal reduction r^* we could solve the remaining part of the problem easily by simply computing a minimum spanning tree with edge-weights given by $\ell - r^*$. We discuss the hardness of the problems in this chapter in Section 8.5.

In our notation the two problems defined above are *tricriteria problems*. Consequently, we will be concerned with *tricriteria approximation algorithms* in this chapter. Consider (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE). The objective function of the problem is the length $(\ell - r)(T)$ of the Steiner tree in the modified network. We consider the diameter constraint to be the first constraint, the budget constraint as the second one. Let A be a tricriteria approximation algorithm with performance (α, β, γ) for the problem. If the set (8.1) of feasible solutions is nonempty, then A must find a reduction of cost at most γB and a tree T of diameter at most βD whose modified length $(\ell - r)(T)$ is at most α times the optimum. If there is no feasible solution, A has the choice of either providing the information that the set (8.1) is empty or returning a reduction r of cost at most γB and a tree of diameter at most βD.

The case is totally similar for (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE).

Additionally to graphs we will also deal with *multigraphs* (see e.g. [Har72]). Multigraphs are a variation of graphs where more than one edge can join two vertices. We refer to edges in a multigraph as *multiedges*.

8.3 Diameter Constrained Steiner Trees

In this section, we discuss our approximation algorithm for (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE). We first present an informal description of the algorithm in order to illustrate the main ideas behind the steps. We then proceed with a more detailed presentation and depict our approximation algorithm in the usual pseudo code.

Algorithm Outline

Our algorithm roughly works as follows: We first modify the graph by replacing each edge with a number of parallel multiedges. In order to establish the performance guarantee, we must show that the modification of the graph preserves the structure of the problem.

The next main step of the algorithm consists of $\mathcal{O}(\log |K|)$ *phases*, where K is the set of terminals. Initially, the solution consists of the empty set. During each phase of the algorithm we choose a set of edges to add to the solution. The set of edges chosen in each iteration is required to possess three desirable properties:

1. The solution cost with respect to the shortened ℓ-cost must be no more than OPT, where OPT denotes the optimally (under a budget B on the cost of improvement) improved total length of a Steiner tree with d-diameter at most D.
2. The diameter value with respect to d must not increase by more than D.
3. The reduction cost spent in each iteration is no more than B.

Since the number of iterations of the algorithm is $\mathcal{O}(\log |K|)$, this will lead to an approximation algorithm which violates the budget and the diameter constraint by a factor of at most $\mathcal{O}(\log |K|)$ and which is within a factor of $\mathcal{O}(\log |K|)$ of the optimal solution. We use a *solution based decomposition method* (cf. Section 3.5) in the analysis of our algorithm: Its basic idea is to take advantage of the existence of an optimal solution to prove that, in each phase, it is possible to choose a good set of edges. We now refine the specification of our algorithm.

Main Step 1: Discretize the problem by using the transformation procedure shown in Algorithm 8.1 on the next page. This transformation builds a multigraph G' from the original graph G in the following way: Each edge $e = (u, v)$ with length $\ell(e)$ and minimum length $\ell_{\min}(e)$ is replaced by $b_e + 2$ parallel multiedges e^k, $k = -1, 0, \dots, b_e$. These edges reflect discrete steps in improvement similar to our construction in Section 7.7 on page 154.

The edge e^{-1} corresponds to an "untouched" version of e with length $\ell'(e^{-1}) := \ell(e)$ and cost $c'(e^{-1}) := 0$. For $k \geq 0$, e^k, represents edge e shortened to $\ell(e) - (1+\varepsilon)^k$. Thus e^k has length $\ell'(e^k) := \ell(e) - (1+\varepsilon)^k$ and cost $c'(e^k) := (1+\varepsilon)^k c_e$. The d-weights for all the edges e^k coincide with that of e. To avoid additional notation, we also use d to denote these weights. This way, each multiedge e' in G' is assigned three numbers: $\ell'(e')$, $c'(e')$ and $d(e')$.

Main Step 2: The algorithm maintains a set of connected subgraphs or *clusters*, each with its own distinguished vertex or *center*. Initially, each terminal is in a cluster by itself. In each phase of the second main step, we set up an auxiliary graph G_i using the constrained shortest path algorithm of [Has92]. We then solve a variant of a constrained matching problem in G_i and, finally, merge the clusters in pairs by adding paths between their centers. Since the number

Algorithm 8.1 Procedure to transform G into G' in the first step.

Input: A graph $G = (V, E)$, four edge weight functions ℓ, $\ell_{\min}$, c, d, and a number B

$V' \leftarrow V$ {The vertex set of the graph to be constructed}
$E' \leftarrow \varnothing$ {The edge set of the graph to be constructed}
for all edges $e = (u, v) \in G$ **do**
  $b_e \leftarrow \lceil \log_{(1+\varepsilon)}(\ell(e) - \ell_{\min}(e)) \rceil$
  Add $b_e + 2$ parallel edges e^k, $k = -1, 0, \ldots, b_e$ between u and v to E'
  $\ell'(e^{-1}) \leftarrow \ell(e)$
  $d(e^{-1}) \leftarrow d(e)$
  $c'(e^{-1}) \leftarrow 0$
  for $k = 0, \ldots, b_e$ **do**
    $\ell'(e^k) \leftarrow \ell(e) - (1+\varepsilon)^k$
    $d(e^k) \leftarrow d(e)$
    $c'(e^k) \leftarrow (1+\varepsilon)^k c_e$
  end for
end for
return $G' = (V', E')$, ℓ', d, c'

of clusters decreases by a factor of 2 due to the merging in each phase, the algorithm terminates in $\lceil \log_2 |K| \rceil$ phases with one cluster.

Main Step 3: We take a tree T' rooted at the center of the single cluster left at the end of the second main step with small diameter under the d-costs.

Main Step 4: In the final step, we use the information from the three values associated with each edge in T' to construct a reduction strategy r and a Steiner tree T for the original graph G. We output r and T.

A complete specification of our Algorithm is shown in Algorithm 8.2 on the next page.

Correctness and Performance Guarantee

To facilitate the presentation, we will assume in the following that r^* is an optimal reduction strategy for the graph G and that T^* is a corresponding optimal Steiner tree of total weight $\mathrm{OPT} := (\ell - r^*)(T^*)$ and diameter $\mathrm{dia}_d(T^*) \le D$. We will show on page 174 how to modify our algorithm to handle the case when the set of feasible solutions is empty.

Algorithm 8.2 Algorithm for diameter bounded Steiner tree improvement..

Input: A graph $G = (V, E)$, a vertex subset K of terminals, four edge weight functions ℓ, $\ell_{\min}$, c, d, and two numbers B and D; a *fixed* accuracy parameter $\varepsilon > 0$

Call Algorithm 8.1 to obtain a new (multi-) graph $G' = (V, E')$ with weights $\ell'(e)$, $c'(e)$, $d(e)$ on the edges $e \in E'$.
$i \leftarrow 1$
{Initialize the phase count}
$\mathcal{C}_1 \leftarrow \{\{v\} : v \in K\}$
{Initialize the set of clusters $\mathcal{C}_1$ to contain $|K|$ singleton sets, one for each terminal in K}
for all $v \in K$ **do**
  center$(\{v\}) \leftarrow v$
end for
{For each cluster in $\mathcal{C}_1$, define the single node in the cluster to be its center.}
while there is more than one cluster in $\mathcal{C}_i$ **do**
  Call Algorithm 8.3 to merge clusters.
  $i \leftarrow i + 1$
  {End of phase i}
end while
Let $\hat{C}$, with *center*$(\hat{C}) = \hat{v}$ be the single cluster left.
Compute a shortest path tree $\hat{T}$ of $\hat{C}$ rooted at $\hat{v}$ with respect to the d-weights (using only edges in $\hat{C}$).
For each multiedge $e^k = (u, v) \in \hat{T}$ define the reduction r on the corresponding edge $e = (u, v)$ in the original graph by

$$r(e) := \begin{cases} 0 & \text{if } k = -1, \\ (1+\varepsilon)^k & \text{if } k \geq 0. \end{cases}$$

Define a tree T in G by including the corresponding edges e.
return r and T

Algorithm 8.3 Subroutine to merge clusters.

Input: A graph $G' = (V, E')$, a vertex subset K of terminals, three edge weight functions ℓ', c', d, and two numbers B and D,
a set $\mathcal{C}_i = \{C_1 \ldots, C_{k_i}\}$ of clusters;
a *fixed* accuracy parameter $\varepsilon > 0$

Let the set of clusters at the beginning of the ith phase be $\mathcal{C}_i = \{C_1 \ldots, C_{k_i}\}$
$s \leftarrow \lceil \log_{(1+\varepsilon)} B \rceil + 1$
$\mathcal{M} \leftarrow \{0, (1+\varepsilon)^0, (1+\varepsilon)^1, \ldots, (1+\varepsilon)^s\}$
Construct a (complete) multigraph $G_i = (V_i, E_i)$ as follows
$V_i \leftarrow \{v_x : v_x$ is the center of cluster $C_x \in \mathcal{C}_i.\}$
for all pairs (C_x, C_y) of clusters in $\mathcal{C}_i$ **do**
  for all $B' \in \mathcal{M}$ **do**
    Let v_x and v_y be the centers of C_x and C_y, respectively.
    Let the path $P_{xy}(B')$ be a $(1 + 1/\varepsilon, 1, 1)$-approximation to the restricted shortest path problem in G' between the centers v_x and v_y, with edge weights ℓ', c', d and the bounds B' on the c'-cost and D on the d-length respectively.
    Include a multiedge (v_x, v_y) in G_i of weight $\ell'(v_x, v_y)$ equal to the ℓ'-cost of $P_{xy}(B')$. The weight $c'(v_x, v_y)$ of the edge is set to that of the c'-cost of $P_{xy}(B')$.
  end for
end for
Find a $(2, 2)$-approximation to the (TOTAL WEIGHT, TOTAL WEIGHT, MAXIMUM MATCHING) problem with edge weights ℓ', c' and the bound $(1+\varepsilon)^2 B$ on the c'-weight of the matching.
For each edge $e = (v_x, v_y)$ in the matching, merge the clusters C_x and C_y, for which v_x and v_y were centers respectively, by adding the corresponding path P_{xy} to form a new cluster C_{xy}. The node (edge) set of the cluster C_{xy} is defined to be the union of the node (edge) sets of C_x, C_y and the nodes (edges) in P_{xy}. One of v_x and v_y is (arbitrarily) chosen to be the center v_{xy} of the cluster C_{xy}. C_{xy} is added to the cluster set $\mathcal{C}_{i+1}$ for the next phase, while C_x and C_y are removed.

Some Basic Lemmas

Our algorithm needs as subroutines two approximation algorithms for certain bicriteria problems on graphs. The first is a constrained variant of the shortest path problem and is hence referred to as the *restricted shortest path problem*. The second one is a *constrained minimum weight matching problem*.

Definition 8.4 (Restricted Shortest Path Problem)
The input for the ***restricted shortest path problem****,* (TOTAL WEIGHT, TOTAL WEIGHT, TOTAL WEIGHT, x-y-PATH)*, consists of a (multi-) graph* $G = (V, E)$*, two distinguished vertices* $x, y \in V$*, three nonnegative edge weight functions* d_1*,* d_2 *and* d_3 *and two numbers* D_2 *and* D_3*. The problem is to find a subgraph from*

$$\left\{ P : P \textit{ is a path between } x \textit{ and } y \textit{ in } G, \sum_{e \in P} d_2(e) \leq D_2 \textit{ and } \sum_{e \in P} d_3(e) \leq D_3 \right\}$$

having minimum value $d_1(P) := \sum_{e \in P} d_1(e)$.

The result of the following lemma is due to Warburton [War92], but can also be proved by an extension of Hassin's work [Has92].

Lemma 8.5 ([War92, Has92]) *For any fixed* $\varepsilon > 0$*, there is an approximation algorithm for* (TOTAL WEIGHT, TOTAL WEIGHT, TOTAL WEIGHT, x-y-PATH) *with a performance of* $(1 + \varepsilon, 1, 1)$. □

As mentioned, we also need results about finding *minimum weight matchings* in graphs. Recall that a matching in a graph $G = (V, E)$ is a subset $M \subseteq E$ of the edges such none of the edges in M are incident. A maximum matching is a matching of maximum cardinality.

Definition 8.6 (Minimum Weight Matching Problem)
An instance of the ***minimum weight matching problem****,* MIN MATCHING*, is given by a graph* $G = (V, E)$ *and an edge weight function* d*. The problem is to find a maximum matching* $M \subseteq E$ *in* G *having minimum weight* $d(M) := \sum_{e \in M} d(e)$.

The minimum weight matching problem can be solved in polynomial time, in fact in time $\mathcal{O}(n^3)$ on a graph with n nodes, by classical algorithms, see e.g. [Law76, LP86]. What we will need for the construction of our approximation algorithm in this section is an approximation algorithm for a bicriteria version of MIN MATCHING. In this problem, which we will denote by (TOTAL WEIGHT, TOTAL WEIGHT, MAXIMUM MATCHING), we are given *two* edge weight functions d_1 and d_2 on the edges of G and the goal is to find a maximum matching M of weight $d_2(M)$ at most a given bound D_2, having minimum weight $d_1(M)$. The results of Marathe et al. in [MR+95a] show that the problem (TOTAL WEIGHT, TOTAL WEIGHT, MAXIMUM MATCHING) has an efficient bicriteria approximation algorithm:

Lemma 8.7 ([MR+95a]) *There is a* $(2, 2)$*-approximation algorithm*[1] *for* (TOTAL WEIGHT, TOTAL WEIGHT, MAXIMUM MATCHING). □

We now show that the transformation procedure does not destroy important properties with respect to the problem.

Lemma 8.8 *For any* $\varepsilon > 0$*, the graph* G' *obtained as a result of applying Algorithm 8.1 to* G *satisfies the following properties:*

1. *There exists a Steiner tree* T' *in* G' *of total* ℓ'*-length at most* OPT*,* c'*-cost no more than* $(1+\varepsilon)B$ *and of* d*-diameter at most* D*.*
2. *Let* $u, v \in K$ *be two terminals. Let* P *be the unique path between* u *and* v *in the optimally improved tree* T^**. Let* $(\ell - r^*)(P)$ *denote the length with respect to* $\ell - r^*$ *and* $C_P := \sum_{e \in P} r(e) \cdot c_e$ *denote the money spent by* r^* *on the edges of* P *respectively. Then, in* G' *there exists a path between* u *and* v *of* d*-diameter at most* D*,* ℓ'*-length at most* $(\ell - r^*)(P)$ *and* c'*-cost at most* $(1+\varepsilon)C_P$*.*

Proof: We only show the first part of the lemma. The second part can be proved similarly. Consider the optimal reduction r^* and a corresponding optimal constrained Steiner tree T^* in G of minimum weight OPT $:= (\ell - r^*)(T^*)$.

We now define a tree T' in G' as follows. For each edge $e = (u, v) \in T^*$ which is reduced by $r^*(e)$ we select a multiedge e^k in G' with the following properties:

$$\begin{aligned}
\ell(e) - (1+\varepsilon)r^*(e) \le \ell'(e^k) &\le \ell(e) - r^*(e) \\
c_e r^*(e) \le c'(e^k) &\le (1+\varepsilon)r^*(e) \\
d(e^k) &= d(e)
\end{aligned}$$

The tree T' defined this way has a total ℓ'-length $\ell'(T')$ which is bounded from above by $(\ell - r)(T^*) = $ OPT. Also, the c'-cost of T' is at most $1+\varepsilon$ times the cost of the reduction r and thus does not exceed $(1+\varepsilon)B$. Clearly, the d-diameters of T' and T^* coincide. □

[1] Actually, the article [MR+95a] contains the stronger result that for any $\varepsilon > 0$ there is a $(1 + 1/\varepsilon, 1+\varepsilon)$-approximation algorithm. For our purposes, it suffices to use the slightly weaker result.

A Bound on the Diameter of the Tree

Observation 8.9 *The set of cluster centers maintained by Algorithm 8.2 is always a subset of the terminals* K *of the original graph* G. □

Lemma 8.10 *Algorithm 8.2 terminates in* $\lceil \log_2 |K| \rceil$ *phases.*

Proof: Let k_i denote the number of clusters in phase i. Note that $k_{i+1} = \lceil \frac{k_i}{2} \rceil$ since we pair up the clusters (using a matching in Step 14 of the merging procedure in Algorithm 8.3). Hence we are left with one cluster after phase $\lceil \log_2 |K| \rceil$ and the algorithm terminates. □

Lemma 8.11 ([RM+93]) *Suppose* T *is a tree with an even number of marked nodes. Then there is a pairing* $(v_1, w_1), \ldots, (v_k, w_k)$ *of the marked nodes such that the* $v_i - w_i$ *paths in* T *are edge-disjoint.* □

Lemma 8.12 *Let* $C \in \mathcal{C}_i$ *be any cluster in phase* i *of Algorithm 8.2 and let* v *be its center. Then any node* u *in* C *is reachable from* v *by a path in* C *of* d*-weight at most* iD.

Proof: By Observation 8.9, the set of cluster centers in phase i is a subset of the terminals K. By Lemma 8.8, there is a Steiner tree in G' of d-diameter at most D and of total c'-cost no more than $(1 + \varepsilon)B$. This implies that all the paths $P_{xy}(B')$ which are constructed in Step 9 of Algorithm 8.3 with the help of Hassin's restricted shortest path algorithm have d-weight at most D.

We now establish the claim of the lemma by induction on i. For $i = 1$ the claim is trivial. Assume now that all the clusters in phase i have the property stated in the claim. Consider the merging of C_x and C_y with centers v_x and v_y to a cluster C_{xy} that then appears in phase $i + 1$. Without loss of generality assume that v_x is chosen to be the center of the cluster C_{xy}. By induction hypothesis, for any node u in C_{xy} that was in C_x there is a path from u to v_x of d-weight at most iD. This path is also present in C_{xy}. Consider a node $u \in C_y$. Then, again by induction hypothesis, there is a path of length at most iD from u to v_y in C_y (and thus also in C_{xy}). Since we have seen that the path added in phase i between v_x and v_y is of d-weight at most D, we can connect u to v_x by a path of d-weight at most $(i + 1)D$. □

Using the result of the last lemma in conjunction with Lemma 8.10 we can establish the following lemma.

Lemma 8.13 *Algorithm 8.2 outputs a Steiner tree* T *with* d*-diameter at most* $2\lceil \log_2 |K| \rceil \cdot D$.

Proof: It follows from Lemmas 8.12 and 8.10 that the tree $\hat{T}$ found in Step 12 of Algorithm 8.2 has d-diameter at most $2\lceil \log_2 |K| \rceil \cdot D$. By construction of r and T in Step 13, the d-diameter of T in G is no more than the d-diameter of T' in G'. □

This completes the proof of the performance guarantee with respect to the d-cost. We now proceed to prove the performance guarantee with respect to the modified ℓ-lengths.

The Total Modified Length

Lemma 8.14 *Let T' be any minimum ℓ'-cost Steiner tree in G' subject to the constraints that its d-diameter is bounded by D and its total c'-cost does not exceed $(1+\varepsilon)B$. The ℓ'-weight of the constrained minimum weight matching found in Step 14 of Algorithm 8.3 in any phase of Algorithm 8.2 is at most $2(1+1/\varepsilon)\ell'(T')$. Also, the c'-cost of the matching is at most $2(1+\varepsilon)^2 \cdot B$.*

Proof: Notice that by Lemma 8.8 a tree satisfying both the constraints as stated in the claim does exist.

Let i be fixed and consider the ith phase of Algorithm 8.2. By Observation 8.9 all the centers $v_1, v_2, \dots, v_{k_i}$ in the ith stage are terminals. Thus, all these vertices are contained in the Steiner tree T'. We mark the nodes $v_1, v_2, \dots, v_{2\lfloor \frac{k_i}{2} \rfloor}$ in T' and apply Lemma 8.11 to T' with these vertices marked.

The lemma states that there is a pairing of the centers such that the paths between the paired vertices in T' are edge disjoint. Without loss of generality assume that this pairing is $(v_1, v_2), \dots, (v_{2\lfloor \frac{k_i}{2} \rfloor - 1}, v_{2\lfloor \frac{k_i}{2} \rfloor})$.

By the disjointness of these paths, their total ℓ'-cost is at most $\ell'(T')$. Similarly, the total c'-cost of the edges in these paths does not exceed $(1+\varepsilon)B$, and none of the paths has length more than D under the d-weight function. By construction of the paths $P_{xy}(B')$ in Step 9 of Algorithm 8.3, for each of the paths P between the centers from above we can find a path P' among the paths $P_{xy}(B')$ in G' with the following properties:

1. The d-length of P' is at most that of P. Thus, the d-length of P' is bounded by D.
2. The total c'-cost of P' is at most $1+\varepsilon$ times the c'-cost of P. The factor $1+\varepsilon$ stems from the discrete budget values, which are used for the constrained paths.
3. The ℓ'-length of P' is at most $1+1/\varepsilon$ times that of P. Here, the factor $1+1/\varepsilon$ comes from the approximation algorithm for the restricted shortest path problem.

Using the edges in G_i corresponding to this selection from the paths $P_{xy}(B')$ yields a matching of the vertices $(v_{2\lfloor \frac{k_i}{2} \rfloor - 1}, v_{2\lfloor \frac{k_i}{2} \rfloor})$ in G_i of ℓ'-length at most $(1 + 1/\varepsilon)\ell'(T')$ and of c'-cost at most $(1 + \varepsilon)c'(T') \leq (1 + \varepsilon)^2 \cdot B$.

In Step 13 of Algorithm 8.3, a $(2, 2)$-approximation for the (TOTAL WEIGHT, TOTAL WEIGHT, MAXIMUM MATCHING) problem is found. This matching M must have c'-cost at most $2(1 + \varepsilon)^2 \cdot B$ and ℓ'-weight at most $2(1 + 1/\varepsilon)\ell'(T')$. This completes the proof. □

We are now ready to estimate the ℓ'-length of the Steiner tree T.

Lemma 8.15 *Algorithm 8.2 outputs a Steiner tree* T *with total length* $(\ell - r)(T)$ *under the modified length function* $\ell - r$ *of at most* $2(1 + 1/\varepsilon)\lceil \log_2 |K| \rceil \cdot \mathrm{OPT}$.

Proof: By the results of Lemma 8.10 and Lemma 8.14, the total ℓ'-length of the edges in the single cluster $\hat{C}$ is at most $2(1+1/\varepsilon)\lceil \log_2 |K| \rceil \cdot \ell'(T')$, where T' is an optimal constrained Steiner tree in G'. Thus, the total ℓ'-length of the shortest path tree $\hat{T}$, which is computed in Step 12 of Algorithm 8.2 is also no more than $2(1 + 1/\varepsilon)\lceil \log_2 |K| \rceil \cdot \ell'(T')$.

By Lemma 8.8, $\ell'(T') \leq \mathrm{OPT}$ and hence

$$\ell'(\hat{T}) \leq 2(1 + 1/\varepsilon)\lceil \log_2 |K| \rceil \cdot \mathrm{OPT}. \tag{8.2}$$

By the construction of the reduction r and the tree T in the final Step 13 of Algorithm 8.2, the length of T under the modified length function $\ell - r$ is bounded from above by $\ell'(\hat{T})$. Now using (8.2) establishes the lemma. □

Bounding the Cost of the Reduction Strategy

We now address the c'-cost of the edges that are added during an arbitrary phase of the algorithm. From Lemma 8.14 and Lemma 8.10 we get:

Lemma 8.16 *The total* c'*-cost of the tree* $\hat{T}$ *found in Step 12 of Algorithm 8.2 is at most* $2(1 + \varepsilon)^2\lceil \log_2 |K| \rceil \cdot B$. *Moreover, the cost of the reduction* r *constructed in Step 13 is also bounded by* $2(1 + \varepsilon)^2\lceil \log_2 |K| \rceil \cdot B$. □

Handling Infeasibility of an Instance

We have already all the main ingredients to establish the performance of our Algorithm 8.2. As long as there exists a feasible solution, the algorithm will find a reduction r of cost $\mathcal{O}(\log |K|) \cdot B$ and a tree T of d-diameter $\mathcal{O}(\log |K|) \cdot D$ such that $(\ell - r)(T) \leq \mathcal{O}(\log |K|) \cdot \mathrm{OPT}$.

To make our algorithm have indeed a $(\mathcal{O}(\log|K|), \mathcal{O}(\log|K|), \mathcal{O}(\log|K|))$ performance, we must assure that, if the algorithm outputs a solution, this solution will violate the constraints on the d-diameter and on the cost of the reduction by at most the logarithmic factors. We have proved that if there is a feasible solution, then the multicriteria approximation algorithms for the restricted shortest path problems and the bicriteria matching problem will not stop and inform us about the infeasibility of the instances of these problems constructed during the run of our algorithm. Thus, if one of these two subroutines detects infeasibility of a subproblem, we can also stop our algorithm and inform about the infeasibility of the instance of (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) to be handled. Finally, by testing the d-diameter of the tree and the cost of the reduction produced by our algorithm and outputting these only if the constraints are violated by at most the logarithmic factors we make sure that we will always output an "almost feasible" solution as stated in Definition 3.5.

Combining all Ingredients

One can verify that Algorithm 8.2 can be implemented to run in polynomial time. Thus, combining the results of Lemmas 8.13, 8.15 and 8.16, we obtain the following theorem.

Theorem 8.17 *For any fixed $\varepsilon > 0$, Algorithm 8.2 is an approximation algorithm for* (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE) *with a performance of $(\alpha_\varepsilon, \beta_\varepsilon, \gamma_\varepsilon)$, where*

$$\begin{aligned}\alpha_\varepsilon &= 2(1+1/\varepsilon)\lceil\log_2|K|,\rceil\\ \beta_\varepsilon &= 2\lceil\log_2|K|\rceil \quad \textit{and}\\ \gamma_\varepsilon &= 2(1+\varepsilon)^2\lceil\log_2|K|\rceil.\end{aligned}$$

Here, K is the set of terminals given in the input. □

8.4 Degree Constrained Steiner Trees

In this section, we present our approximation algorithm for the problem (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) given in Definition 8.3. The basic idea is very similar to the (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE) problem and so we only highlight the details.

Algorithm Outline

As in the case of the approximation of (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE), we first transform the input graph G to another graph G' using the transformation procedure Algorithm 8.1.[2]. After this step, our algorithm for (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) works again in *phases*. There are $\mathcal{O}(\log \frac{|K|}{b})$ phases. At the beginning, the solution subgraph is empty and each terminal is in a connected component by itself. We call a connected component *active*, if it contains at least one terminal. At each iteration, we add edges between the components, reducing the number of active components in the current solution by at least a factor of one half. When the number of active components falls to $\mathcal{O}(b)$, we compute a (constrained) minimum Steiner tree in the current graph to connect up all the remaining active components.

The choice of edges to be included in each phase will be done in the following way:

1. The solution cost with respect to the modified ℓ costs is bounded from above by OPT.
2. The degree of any node in the graph increases by at most $\mathcal{O}(b)$.
3. The reduction cost spent on the edges is no more than the given budget B.

The edges will be selected by solving a variant of a matching problem, called the b-*bounded even* DCS *problem*. This problem is introduced formally in the next section. Algorithm 8.4 on the facing page displays our approximation algorithm. This algorithm uses Algorithm 8.5 to merge clusters in each phase. We note that each iteration, the solution to the DCS problem may contain one or more copies of an edge $e \in E'$. In any case, in Step 6 of Algorithm 8.5, we include only one copy of such an edge in the set F. The results on the performance are summarized in the following theorem.

Theorem 8.18 *For any fixed $\varepsilon > 0$, Algorithm 8.4 is an approximation for* (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) *with performance $(\alpha_\varepsilon, \beta_\varepsilon, \gamma_\varepsilon)$, where*

$$\alpha_\varepsilon \in \mathcal{O}(\log \frac{|K|}{b}),$$
$$\beta_\varepsilon \in \mathcal{O}(\log \frac{|K|}{b}) \quad \textit{and}$$
$$\gamma_\varepsilon \in (1+\varepsilon) \cdot \mathcal{O}(\log \frac{|K|}{b}).$$

[2] Actually, the transformation procedure used here is a trivial modification of Algorithm 8.1, where no d-weights are involved.

Here, K *is the set of terminals and* b *is the bound on the degree of the tree given in the input.*

The proof of Theorem 8.18 is again based on several lemmas which are presented in the sequel.

Algorithm 8.4 Algorithm for degree bounded Steiner tree improvement.

Input: A graph $G = (V, E)$, a vertex subset K of terminals, three edge weight functions ℓ, $\ell_{\min}$, c, an integer b, and number B; a *fixed* accuracy parameter $\varepsilon > 0$

Call Algorithm 8.1 obtain a new graph $G' = (V, E')$ with two weights $\ell'(e)$, $c'(e)$ on the edges $e \in E'$
$F \leftarrow \varnothing$
$i \leftarrow 1$
Initialize the set of active components to contain $|K|$ singleton sets, one for each terminal in K
repeat
  Call Algorithm 8.5 to merge clusters
  $i \leftarrow i + 1$
until the number of active components is $\mathcal{O}(b)$
{The number of active components of (V, F) is now $\mathcal{O}(b)$}
Contract each active component of (V, F) to a single node
In the resulting graph, use the active components as terminals and compute a Steiner tree of c'-cost at most $4(1+\varepsilon)B$ and of ℓ'-weight at most 4 times that of an minimum ℓ'-weight Steiner tree subject to the constraint that its c'-cost does not exceed $(1+\varepsilon)B$
Add the edges of this Steiner tree to the set F
Find a spanning tree T' of (V, F). For each edge $e' = (u, v) \in T'$ define the reduction r on the corresponding edge e in the original graph by $r(e) := c'(e')/c_e$. Define a tree T in G by including the corresponding edges e.
return r and T

Some Basic Lemmas

We first introduce the degree constrained subgraph problem, DCS, which we are going to solve in each phase to select a good set of edges to add to the solution. In its most general form this problem can be stated as follows: Given an undirected graph with nonnegative costs on the edges, and an integer-valued function f defined on the vertices of the graph, find a minimum-cost subgraph

Algorithm 8.5 Merging clusters in the degree case.

Input: A graph $G' = (V, E')$, a vertex subset K of terminals, two edge weight functions ℓ', c' an integer b, a number B, a set of active components $\mathcal{C}_i = \{ C_1 \dots , C_{k_i} \}$; a *fixed* accuracy parameter $\varepsilon > 0$

Let the set of active components at the beginning of the ith phase be $\mathcal{C}_i = \{ C_1 \dots , C_{k_i} \}$

Construct an auxiliary graph $G_i = (V \cup \mathcal{C}_i, E_i)$ as follows:

- Between all the nodes of V in each active component $C_j \in \mathcal{C}_i$, include edges of zero ℓ'- and c'-cost to form a clique in G_i.

- Let $\tilde{E} \subseteq E'$ represent the set of edges of G whose endpoints are in different components in $\mathcal{C}_i$. For each edge $e = (u, v) \in \tilde{E}$ of length $\ell'(e)$ and cost $c'(e)$, include four sets of edges in G_i: let $u \in C_u$ and $v \in C_v$. We include the sets of edges (u, v), (u, C_v), (v, C_u) and (C_u, C_v), each of ℓ'-length $\ell'(e)$ and c'-cost $c'(e)$, in G_i.

If $|\mathcal{C}_i|$ is odd, we include an extra dummy node z in $\mathcal{C}_i$ and include zero-cost edges between z and every $C_j \in \mathcal{C}_i$ in G_i.

Find an $(2, 2)$-approximation $H = (V \cup \mathcal{C}_i, E_H)$ to the bicriteria $2b$-bounded even DCS problem on G_i with nodes in V having even degree (between 0 and $2b$) and the nodes in $\mathcal{C}_i$ having degree 1, where the bound on the c'-cost of the subgraph is set to $(1 + \varepsilon)B$.

for all $e \in E_H$ **do**
    if e is an edge of the original graph **then**
        $F \leftarrow F \cup \{e\}$
    else $\{e$ is of the form (u, C_v) or $(C_u, C_v)\}$
        $F \leftarrow F \cup \{(u, v)\}$
    end if
end for
Update the set of clusters

(if one exists) such that the degree of any vertex v in this subgraph is $f(v)$. This problem is referred to as the f-*factor problem* [LP86], and is known to be polynomial time solvable [EJ73, LP86]. The variant that we are going to need can be defined as follows.

Definition 8.19 (b-Bounded Even DCS Problem)
An instance of the ***b-bounded even*** **DCS** ***problem*** *is given by an undirected graph* $G = (V, E)$ *such that* $V = S \cup U$ *and* $S \cap U = \emptyset$ *with weights* $d(e)$*,* $e \in E$ *on the edges, and an integer* $b \geq 2$*. The problem consists of finding a subgraph* H *(if one exists) of* G *of minimum cost* $d(H)$ *such that*

- *for all vertices* $v \in U$*, we have* $\deg_H(v) = 1$*, and*
- *for all vertices* $v \in S$*, we have* $0 \leq \deg_H(v) \leq b$ *and* $\deg_H(v) \equiv 0 \pmod 2$*.*

Here, $\deg_H(v)$ *denotes the degree of the node* v *in the subgraph* H*.*

As shown in [LP86] the b-bounded even DCS problem can be solved in polynomial time. Again, one can define a bicriteria version of the problem. In this case, one is given two edge weight functions and an additional constraint on the weight of the subgraph with respect to the second weight function is added.

Definition 8.20 (Bicriteria DCS Problem)
An instance of the ***bicriteria*** **DCS** ***problem****,* (TOTAL WEIGHT, TOTAL WEIGHT, DCS)*, is given by an undirected graph* $G = (S \cup U, E)$*, where* $S \cup U = \emptyset$*, with two nonnegative edge weight functions* d_1*,* d_2 *and an integer* $b \geq 2$*. The problem consists of finding a subgraph from*

$$\left\{ H : \begin{array}{l} H \textit{ satisfies the conditions given in Definition 8.19} \\ \textit{and } \sum_{e \in H} d_2(e) \leq B \end{array} \right\},$$

such that $d_1(H) := \sum_{e \in H} d_1(e)$ *is minimized.*

The results in [MR+95a] show that this problem has an efficient bicriteria approximation algorithm.

Lemma 8.21 ([MR+95a]) *There is a polynomial time* $(2, 2)$*-approximation algorithm for the bicriteria* b*-bounded even* DCS *problem.* □

We need one more result on the structure of degree constrained subgraphs:

Lemma 8.22 ([Rav93]) *Let* H *be any subgraph satisfying the conditions given in Definition 8.19. Then there is a pairing of the nodes of* U *in* H *such that there are edge-disjoint paths in* H *between each pair of vertices.* □

Define a bicriteria variant of the minimum Steiner tree problem in the following way. We are given two edge weight functions instead of just one and there is a bound on the weight of the Steiner tree with respect to the second weight function. Using techniques from [MR+95a] in conjunction with the approximation algorithm for the minimum weight Steiner tree problem given in [GG+94] (see Theorem 7.9 on page 150), the following lemma can be established.

Lemma 8.23 ([MR+95a]) *There is a* $(4,4)$*-approximation algorithm for the bicriteria Steiner tree problem.* □

Proof of the Performance

Throughout the analysis we adopt the following notation: We use b to denote the degree bound specified in the problem. We let r^* be an optimal reduction strategy, T^* a corresponding minimum Steiner tree of bounded degree and $\mathrm{OPT} = (\ell - r^*)(T^*)$ denote its total weight in the modified graph.

First we have to show that during the transformation of the graph basically no structure is lost. The following lemma can be proved in a manner similar to Lemma 8.8:

Lemma 8.24 *Let* G' *be the graph resulting from the transformation. Then the following statements hold:*

1. *There exists a Steiner tree* T' *in* G' *of total* ℓ'*-length at most* OPT*,* c'*-cost no more than* $(1+\varepsilon)B$ *such that each node in* T' *has degree no more than* b.

2. *Let* $u, v \in V$ *be two vertices. Let* P *be the unique path between* u *and* v *in the optimally improved tree* T^**. Let* $(\ell - r^*)(P)$ *and* $C_P = \sum_{e \in P} r(e) \cdot c_e$ *denote the length with respect to* $\ell - r^*$ *and the money spent by* r^* *on the edges of* P *in* T^* *respectively. Then, in* G' *there exists a path between* u *and* v *of* ℓ'*-length at most* $(\ell - r^*)(P)$ *and* c'*-cost at most* $(1+\varepsilon)C_P$. □

Lemma 8.25 *Algorithm 8.4 terminates after* $\mathcal{O}(\log \frac{|K|}{b})$ *phases.*

Proof: Fix an iteration i. Let $\mathcal{C}_i = \{C_1, \ldots, C_{k_i}\}$ be the active clusters at the beginning of the ith iteration. Now consider the solution H of the bicriteria DCS problem constructed in Step 3 of Algorithm 8.5.

First, assume that k_i is even, i.e., that no dummy node was added. We apply Lemma 8.22 to the subgraph H with the nodes in $\mathcal{C}_i$ marked. This gives us a pairing of the active components in $\mathcal{C}_i$ such that there are (edge-disjoint) paths between the active components. By the way we add edges in the loop in Step 4 to the set F it follows that after this addition, there will also be paths between these clusters in (V, F). But this means that each of the pairs in the pairing will have merged into one active cluster at the end of the ith iteration. Thus, in the $(i+1)$st iteration we will have at most $k_i/2 \leq \lceil k_i/2 \rceil$ active components.

In the case that k_i is odd, we have added a dummy node to $\mathcal{C}_i$. In this case, using the above pairing argument, we see that in the next iteration, we have at most $(k_i - 1)/2 + 1 \leq \lceil k_i/2 \rceil$ active clusters.

Since we start with $k_1 := |K|$ active clusters, after $\mathcal{O}(\log \frac{|K|}{b})$ phases we will be left with $\mathcal{O}(b)$ active components. □

By the constraint on the degree of the nodes in the subgraph added at each iteration of the algorithm, the degree of any node increases only by $\mathcal{O}(b)$ in each step. Observe also that the edges added to the solution resulting from the Steiner tree computed in Step 12 of Algorithm 8.4 will add only $\mathcal{O}(b)$ to the degree. The following lemma is now almost immediate:

Lemma 8.26 *The degree of the spanning tree output by the above algorithm is* $\mathcal{O}(b \log \frac{|K|}{b})$. □

Lemma 8.27 *At each iteration* i *of the algorithm, the total* ℓ'*-length of the approximate solution to the* DCS *problem computed in Step 3 of Algorithm 8.5 during this iteration is at most* $2 \cdot \mathrm{OPT}$.

Proof: Fix an iteration i and let $\mathcal{C}_i = \{ C_1, \ldots, C_{p_i} \}$ be the active components in this iteration. We show that there exists a solution of ℓ'-cost at most OPT to the DCS problem set up in the iteration.

Consider again the Steiner tree T' of bounded degree b. Lemma 8.24 ensures that its ℓ'-weight is no more than OPT, while its c'-cost is bounded from above by $(1 + \varepsilon)B$. Contract the nodes in each of the connected components of (V, F) to a single vertex. It is then straightforward to obtain a Steiner tree T_i from T' spanning all the active components and of ℓ'- and c'-cost no more than those of T'. However, T_i might have nodes of degree larger than b.

Assume now for the sake of simplicity that $|\mathcal{C}_i|$ is even.[3] We apply Lemma 8.11 to T_i where we mark all the active components. This yields a pairing of the active components such that all paths between the nodes are pairwise edge-disjoint. We now construct a feasible solution H to the DCS problem:

(i) First, consider all pairs C_u, C_v of active components such that the path between them in T_i consists of a single edge $e' = (C_u, C_v)$. This edge

[3]The case when $|\mathcal{C}_i|$ is odd can be treated similarly by adding the dummy node included to the set $\mathcal{C}_i$.

stems from an edge $(u, v) \in T'$, where $u \in C_u$ and $v \in C_v$. By the construction in Step 2 of Algorithm 8.5, there is an edge (C_u, C_v) in G_i of ℓ'-length $\ell'(e')$ and c'-cost $c'(e')$. We include this edge (C_u, C_v) in H.

(ii) Now, look at pairs of components C_u, C_v between which the paths in T_i consist of more than a single edge. Each node on such a path is either an active component or a node which is not contained in any active component. Without loss of generality let the path in T_i be such that starting with C_u it first hits only active components and then original nodes before ending in C_v. Formally, let the path be

$$((C_u, C_1), (C_1, C_2), \ldots, (C_{q-1}, C_q), (C_q, w_1), (w_1, w_2), \ldots, (w_r, C_v)),$$

where the C_i are active components and the w_j are original nodes. For each edge (C_i, C_{i+1}) in this path, there is a corresponding edge in T' from which this edge is derived (see Figure 8.1 on the facing page for an illustration). Let these edges be $(z_u, b_1), (a_1, b_2), \ldots, (a_q, z_v)$. For all j, the vertices b_j and a_j are in the same active component. Thus, they either coincide or there is an edge in G_i of zero ℓ'-length and c'-cost joining them. We define a path P_{C_u,C_v} in G_i as follows:

$$\begin{aligned} P_{C_u,C_v} := ((C_u, b_1), (b_1, a_1), (a_1, b_2), \ldots, (a_{q-1}, b_q), \\ (b_q, a_q), (a_q, w_1), (w_1, w_2), \ldots, (w_r, C_v)), \end{aligned}$$

where we include the edge (b_j, a_j) only if $a_j \neq b_j$. This way we find a path P_{C_u,C_v} between every pair C_u, C_v in the pairing where the path in T_i consists of more than one edge. We add those edges to H which occur in an odd number of the paths P_{C_u,C_v}.

Since the original paths were edge-disjoint it follows by the construction above that the total ℓ'-cost of all the edges added to H is no more than the $\ell'(T')$, which is bounded by OPT. Similarly, the c'-cost of H does not exceed $(1 + \varepsilon)B$.

We now verify that the degree constraints for the DCS problem are satisfied by H. We must show that all active components C_j have degree equal to one and that the degree of each node $v \in V$ is an even number between 0 and 2b.

Each cluster node C_j has degree equal to one in H since we have found a pairing of the cluster nodes and each cluster node occurs only once and only as an end node in all the paths constructed in the two steps above.

Recall that each vertex $v \in V$ has degree at most b in T'. The only place where we add edges to H that are adjacent to original vertices from V and which are not in T' is in Step (ii) above. By construction, we add at most one extra edge of zero costs for every edge of T' incident on v. Thus, there could be at most 2b edges incident with v. Since we chose only edges that occurred an

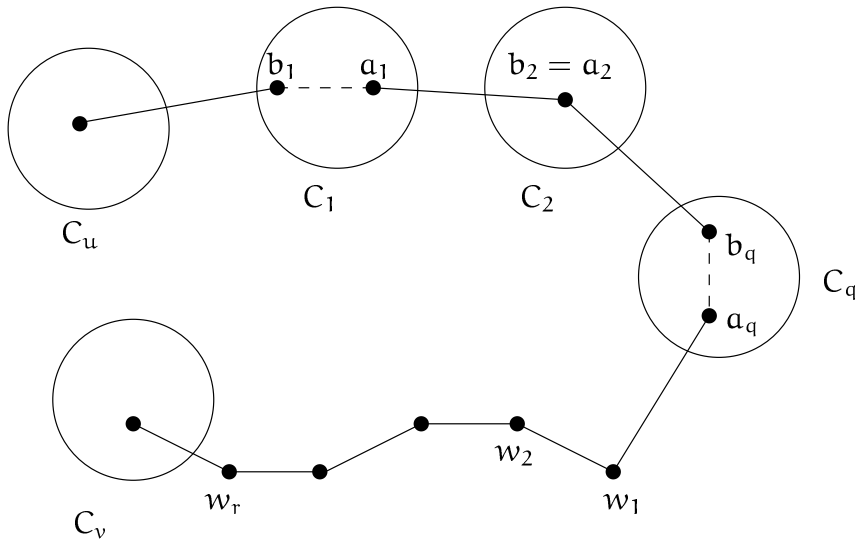


Figure 8.1: Converting the paths in T' to paths in G_i.

odd number of times in the paths, the degree of v is clearly bounded by $2b$. Observe also that all vertices $v \in V$ occur only as internal nodes of the paths that are constructed in the second step. Thus, since we add only edges to H that occur in an odd number of paths, the degree of each such vertex in H must be even.

Thus, we have shown that there is a solution to the DCS problem in Step 3 of Algorithm 8.5 of total ℓ'-length at most OPT and of c'-cost at most $(1+\varepsilon)B$. Thus, the DCS found in Step 3 of Algorithm 8.4 has total c'-cost at most $2(1+\varepsilon)B$ and total ℓ'-length at most $2 \cdot \text{OPT}$. □

Lemma 8.28 *The total ℓ'-length of the edges added to the solution in Step 10 of Algorithm 8.4 is at most* $4 \cdot \text{OPT}$.

Proof: Contract each active component of (V, F) to a single supernode. Also, consider the $\mathcal{O}(b)$ active components as terminals. A minimum ℓ'-weight Steiner tree T' in G' subject to the bounds b on its degree and $(1+\varepsilon)B$ on its c'-cost, induces a Steiner tree for the $\mathcal{O}(b)$ active components, whose c'-costs and ℓ'-lengths are bounded by those of T'. By Lemma 8.24 the ℓ'-weight of T' is no more than OPT. The claim for Step 10 now follows from Lemma 8.23. □

To complete the analysis, observe that the DCS found in Step 3 of Algorithm 8.5 has c'-cost at most $2(1+\varepsilon)B$. Also, the Steiner tree which is found in Step 12 of Algorithm 8.4 has c'-cost that is bounded by twice this value. Using these results in conjunction with the result from Lemma 8.25 that the number of phases is $\mathcal{O}(\log \frac{|K|}{b})$, we can conclude that the total c'-cost of the edges added to the solution is bounded as specified in Theorem 8.18. Clearly, this is also a bound for the c'-cost of the tree T' computed in Step 12.

Using Lemmas 8.27 and 8.28 with the number of iterations yields the corresponding bound on the ℓ'-length of T'. The degree bound has been established in Lemma 8.26. By construction of the reduction r and the Steiner tree T in Step 12 the Theorem 8.18 now follows.

8.5 Hardness Results

In this section we briefly mention results on the hardness of the problems studied in this chapter. Since (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE) includes both the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, SPANNING TREE) and the (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE) problem from Chapter 7 as a special case ($K = V$, $d \equiv 0$, $D = 0$), from Theorems 7.17 and 7.18 we get the following hardness results.

Theorem 8.29 *The problem* (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE) *is* NP*-hard, even when restricted to series-parallel graphs. Unless* P = NP*, for any* $\alpha, \beta \geq 1$*, there is no polynomial time* $(\alpha, \beta, 1)$*-approximation for the* (TOTAL WEIGHT, EDGE IMPROVEMENT COST, STEINER TREE) *problem even when restricted to bipartite graphs. Unless,* P = NP*, for any* $\beta, \gamma \geq 1$*, there is no polynomial time* $(1, \beta, \gamma)$*-approximation algorithm for* (TOTAL WEIGHT, DIAMETER, EDGE IMPROVEMENT COST, STEINER TREE). *All of the above hardness results continue to hold, if the* d*-costs are all equal to zero.*

Again by the NP-hardness of the Steiner tree problem, we can conclude that there can be no approximation algorithm for (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) with a performance of $(1, \beta, \gamma)$ for any $\beta, \gamma \geq 1$.

Theorem 8.30 *Unless* P = NP*, for any* $\alpha, \gamma \geq 1$ *there can be no* $(\alpha, 1, \gamma)$*-approximation algorithm for the* (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) *problem. This result continues to hold if* α *and* γ *are replaced by polynomial time computable functions and the set* K *of terminal equals* V*.*

Proof: We use a reduction from HAMILTONIAN PATH problem, which is well known to be NP-complete [GJ79, Problem GT39]. An instance of HAMILTONIAN PATH is given by an undirected graph $G = (V, E)$. The question is whether there is a Hamiltonian path in G, i.e., a simple path that touches all the vertices in G exactly once.

Given an instance of HAMILTONIAN PATH we construct an instance of (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) as follows. We take a complete graph $G' = (V, E')$ with the same vertex set as G. We define

$$\ell(e) := \ell_{\min}(e) := \begin{cases} 1 & \text{if } e \in E \\ \alpha \cdot n + 1 & \text{if } e \notin E \end{cases}$$

and set $c_e \equiv 0$. To complete the construction, set $b = 2$ and $B := 0$. Thus, the only possible reduction in G' is $r \equiv 0$ and (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) reduces to finding a minimum weight spanning tree of degree at most 2.

It is easy to see that G has a Hamiltonian path if and only if an algorithm with performance $(\alpha, 1, \gamma)$ returns a solution to the instance of (TOTAL WEIGHT, DEGREE, EDGE IMPROVEMENT COST, STEINER TREE) with objective function value $n - 1$. □

It should be noted that the construction in the proof of the last theorem can be easily adapted for values of the degree bound b different from 2. To this end, for each vertex v in the graph, one adds an appropriate number of vertices $u_{v,1}, \dots, u_{v,b-2}$ that are adjacent only to v. Any spanning tree in G must include the edges $(v, u_{v,i})$, $i = 1, \dots, b-2$. Thus any vertex v has degree at least $b - 1$. The proof for arbitrary values of b now follows along the lines of the proof of the case that $b = 2$.

9 Conclusion

Approximation algorithms are *one* useful way to deal with hard problems. Although an approximation algorithm neither cures a problem from being hard nor always finds an optimal solution, its theoretically proven performance gives the certainty that in any case one is never too far away from the optimum.

The concept of multicriteria optimization problems on graphs developed in this thesis tries to enlarge the class of problems which can be approximated within guaranteed bounds. Finding a good solution that strictly obeys the constraints given in a problem could be extremely hard. However, a multicriteria algorithm might deliver solutions of good quality while violating the constraints within the bounds specified in the performance guarantee vector.

The most challenging question raised by our work is which problems do have efficient, i.e., constant factor, multicriteria approximation algorithms. Possibly this question can be answered by developing a concept of approximation preserving reducibility between multicriteria problems, similar to the L-reduction by Papadimitriou and Yannakakis [PY91] for the unicriterion case.

A

Notational Conventions

$\varnothing$	the empty set
$\mathbb{Z}$	the set of integers, that is $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Z}_+$	the set of positive integers
$\mathbb{N}$	the set of natural numbers without zero
$\mathbb{Q}$	the set of rational numbers
$\mathbb{Q}_+$	the set of positive rational numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{R}_+$	the set of positive real numbers
2^A	the set of subsets of the set A
$\|A\|$	the cardinality of the set A
$A \subseteq B$	A is a (not necessarily proper) subset of B
$A \subset B$	A is a proper subset of B
$f(n) \in \mathcal{O}(g(n))$	f grows at most as fast as g (see page 11)
$f(n) \in \Omega(g(n))$	f grows at least as fast as g (see page 11)
$f(n) \in \Theta(g(n))$	f grows exactly as fast as g (see page 11)
$f(n) \in o(g(n))$	f grows more slowly than g (see page 11)
$\log_a$	logarithm to the basis of a
$\log$	logarithm to the basis of 2
$\ln$	natural logarithm (basis e)
$\mathcal{A}$	Ackerman's function (see page 12)
$\alpha(m, n)$	inverse of Ackerman's function, $\alpha(m, n) = \min\left\{\, i \geq 1 : \mathcal{A}(i, \lfloor m/n \rfloor) > \log_2 n \,\right\}$ (see page 12)
$\beta(m, n)$	$\beta(m, n) = \min\{\, i : \log^{(i)} n \leq m/n \,\}$ (see page 12)

Bibliography

[AA$^+$95] B. Awerbuch, Y. Azar, A. Blum, and S. Vempala, *Improved approximation guarantees for minimum-weight* k*-trees and prize-collecting salesmen*, Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC'95), 1995, pp. 277–376.

[AAN82] V. Aggarwal, V. Aneja, and K. Nair, *Minimal spanning tree subject to a side constraint*, Computational Operations Research **9** (1982), 287–296.

[AC$^+$93] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese, *An algebraic theory of graph reductions*, Journal of the ACM **40** (1993), no. 5, 1134–1164.

[AH77] K. Appel and W. Haken, *Every planar map is four colorable, part I: Discharing*, Illinois Journal of Mathematics **21** (1977), 429–490.

[AHK77] K. Appel, W. Haken, and J. Koch, *Every planar map is four colorable, part II: Reducibility*, Illinois Journal of Mathematics **21** (1977), 491–567.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1974.

[AI$^+$91] A. Aggarwal, H. Imai, N. Katoh, and S. Suri, *Finding* k *points with minimum diameter and related problems*, Journal of Algorithms **12** (1991), no. 1, 38–56.

[AL$^+$92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and intractability of approximation problems*, Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science (FOCS'92), October 1992, pp. 13–23.

[BCV95] A. Blum, P. Chalasani, and S. Vempala, *A constant-factor approximation for the* k*-MST problem in the plane*, Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC'95), 1995, pp. 294–302.

[BE95] M. Bern and D. Eppstein, *Approximation algorithms for geometric problems*, (unpublished manuscript), January 1995.

[Ber92] O. Berman, *Improving the location of minisum facilities through network modification*, Annals of Operations Research **40** (1992), 1–16.

[BGS95] M. Bellare, O. Goldreich, and M. Sudan, *Free bits, PCPs and non-approximability — towards tight results*, Proceedings of the 36th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'95), 1995, pp. 422–431.

[BIKP93] J. Bar-Ilan, G. Kortsarz, and D. Peleg, *How to allocate network centers*, Journal of Algorithms **15** (1993), no. 3, 385–414.

[BLW87] M. W. Bern, E. L. Lawler, and A. L. Wong, *Linear-time computation of optimal subgraphs of decomposable graphs*, Journal of Algorithms **8** (1987), 216–235.

[Bod92] H. L. Bodlaender, *A tourist guide through treewidth*, Tech. Report RUU-CS-92-12, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 1992.

[BS94] M. Bellare and M. Sudan, *Improved non-approximability results*, Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC'94), 1994, pp. 184–193.

[CH96] B. Chandra and M. M. Halldórsson, *Facility dispersion and remote subgraphs*, Proceedings of the 8th Scandinavian Workshop on Algorithm Theory, Reykjavik, Iceland, Lecture Notes in Computer Science, vol. 1097, 1996, pp. 53–65.

[Chr76] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Tech. report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[CK+92] J. Cong, A. B. Kahng, G. Robins, M. Sarafzadeh, and C. K. Wong, *Provably good performance driven global routing*, IEEE Transactions on Computer Aided Design **11** (1992), no. 6, 739–752.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1990.

[Col88] R. Cole, *Parallel merge sort*, SIAM Journal on Computing **17** (1988), no. 4, 770–785.

[Coo94] C. Cooper, *Probabilistic analysis of two* k*-cluster problems*, Discrete Applied Mathematics **50** (1994), 27–51.

[CP91] P. Crescenzi and A. Panconesi, *Completeness in approximation classes*, Information and Computation **93** (1991), no. 2, 241–262.

[CW87] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Proceedings of the 19th Annual ACM Symposium on the Theory of Computing (STOC'87), 1987, pp. 1–6.

[DF85] M. E. Dyer and A. M. Frieze, *A simple heuristic for the* p*-center problem*, Operations Research Letters **3** (1985), no. 6, 285–288.

[EJ73] J. Edmonds and E. L. Johnson, *Matching, Euler tours and the Chinese postman*, Mathematical Programming **5** (1973), 88–124.

[EN89] E. Erkut and S. Neuman, *Analytical models for locating undesirable facilities*, European Journal of Operations Research **40** (1989), 275–291.

[Fag74] R. Fagin, *Generalized first-order spectra and polynomial time recognizable sets*, SIAM-AMS Proceedings **7** (1974), 43–73.

[Fei96] U. Feige, *A threshold of* $\ln n$ *for approximating set cover*, Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC'96), 1996, pp. 314–318.

[FG88] T. Feder and D. Greene, *Optimal algorithms for approximate clustering*, Proceedings of the 20th Annual ACM Symposium on the Theory of Computing (STOC'88), 1988, pp. 434–444.

[FSO96] G. N. Frederickson and R. Solis-Oba, *Increasing the weight of minimum spanning tree*, Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96), March 1996.

[GG+86] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica **6** (1986), 109–122.

[GG+94] M. X. Goemans, A. V. Goldberg, S. Plotkin, D. B. Shmoys, E. Tardos, and D. P. Williamson, *Improved approximation algorithms for network design problems*, Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94), January 1994, pp. 223–232.

[GH94] N. Garg and D. S. Hochbaum, *An* $\mathcal{O}(\log k)$ *approximation algorithm for the* k *minimum spanning tree problem in the plane*, Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC'94), May 1994, pp. 432–438.

[GJ76] M. R. Garey and D. S. Johnson, *Some simplified* NP*-complete graph problems*, Theoretical Computer Science **1** (1976), 237–267.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and intractability (a guide to the theory of* NP*-completeness)*, W.H. Freeman and Company, New York, 1979.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver, *Geometric algorithms and combinatorial optimization*, Springer-Verlag, Berlin Heidelberg, 1988.

[Gon85] T. F. Gonzalez, *Clustering to minimize the maximum intercluster distance*, Theoretical Computer Science **38** (1985), 293–306.

[GW92] M. W. Goemans and D. P. Williamson, *A general approximation technique for constrained forest problems*, Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'92), January 1992, pp. 307–316.

[Hal93] M. M. Halldórsson, *Approximating the minimum maximal independence number*, Information Processing Letters **46** (1993), 169–172.

[Ham95] H. W. Hamacher, *Mathematische Lösungsverfahren für planare Standortprobleme*, Lehrbuch Wirtschaftsmathematik, Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 1995.

[Har72] F. Harary, *Graph theory*, Addison-Wesley Publishing Company, Inc., 1972.

[Has92] R. Hassin, *Approximation schemes for the restricted shortest path problem*, Mathematics of Operations Research **17** (1992), no. 1, 36–42.

[HM79] G. Y. Handler and P. B. Mirchandani, *Location on networks: Theory and algorithms*, MIT Press, Cambridge, MA, 1979.

[HS86] D. S. Hochbaum and D. B. Shmoys, *A unified approach to approximation algorithms for bottleneck problems*, Journal of the ACM **33** (1986), no. 3, 533–550.

[Joh74] D. S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences **9** (1974), 256–278.

[Kan92] V. Kann, *On the approximability of* NP*-complete optimization problems*, Ph.D. thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, May 1992.

[KJ83] B. Kadaba and J. Jaffe, *Routing to multiple destinations in computer networks*, IEEE Transactions on Communication **COM-31** (1983), 343–351.

[Klo93] T. Kloks, *Treewidth*, Ph.D. thesis, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 1993.

[Klo94] Ton Kloks, *Treewidth: computations and approximations*, Lecture Notes in Computer Science, vol. 842, Springer-Verlag Inc., New York, NY, USA, 1994.

[KM$^+$97] S. O. Krumke, M. V. Marathe, H. Noltemeier, V. Radhakrishnan, S. S. Ravi, and D. J. Rosenkrantz, *Compact location problems*, Theoretical Computer Science (1997), To appear.

[KN$^+$95a] S. O. Krumke, H. Noltemeier, S. S. Ravi, and M. V. Marathe, *Compact location problems with budget and communication constraints*, Proceedings of the First International Conference on Computing and Combinatorics (Cocoon'95), Xi'an, PR China, Lecture Notes in Computer Science, vol. 959, August 1995, pp. 510–519.

[KN$^+$95b] S. O. Krumke, H. Noltemeier, S. S. Ravi, and M. V. Marathe, *Complexity and approximability of some bicriteria location problems*, Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science, Aachen, Germany, Lecture Notes in Computer Science, vol. 1017, June 1995, pp. 73–87.

[KN$^+$96a] S. O. Krumke, H. Noltemeier, M. V. Marathe, R. Ravi, and S. S. Ravi, *Improving Steiner trees of a network under multiple constraints*, Tech. Report LA-UR 96-1374, Los Alamos National Laboratory, Los Alamos, New Mexico, USA, 1996.

[KN$^+$96b] S. O. Krumke, H. Noltemeier, M. V. Marathe, and S. S. Ravi, *Node weighted upgrading problems*, Tech. Report LA-UR 96-1467, Los Alamos National Laboratory, Los Alamos, New Mexico, USA, 1996.

[KN$^+$96c] S. O. Krumke, H. Noltemeier, S. S. Ravi, and M. V. Marathe, *Bicriteria compact location problems*, Recent Advances in Locational Analysis (H. W. Hamacher, K. Klamroth, and S. Nickel, eds.), Studies in Locational Analysis, vol. 10, October 1996, pp. 37–51.

[KN$^+$96d] S. O. Krumke, H. Noltemeier, S. S. Ravi, M. V. Marathe, and K. U. Drangmeister, *Modifying networks to obtain low cost trees*, Proceedings of the 22nd International Workshop on Graph-Theoretic Concepts in Computer Science, Cadenabbia, Italy, Lecture Notes in Computer Science, June 1996, pp. 293–307.

[KPP92a] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, *Multicasting for multimedia applications*, Proceedings of the IEEE INFOCOM'92, January 1992, pp. 2078–2085.

[KPP92b] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, *Two distributed algorithms for the constrained Steiner tree problem*, Tech. Report CAL-1005-92, Computer Systems Laboratory, University of California, San Diego, October 1992.

[KPP93] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, *Multicast routing for multimedia communication*, IEEE/ACM Transactions on Networking (1993), 286–292.

[KR93] P. Klein and R. Ravi, *A nearly best-possible approximation for node-weighted Steiner trees*, Proceedings of the 3rd MPS conference on Integer Programming and Combinatorial Optimization, 1993, pp. 323–332.

[Kru95a] S. O. Krumke, *On a generalization of the* p*-center problem*, Information Processing Letters **56** (1995), 67–71.

[Kru95b] S. O. Krumke, *On the convergence of a modified barrier function method for convex quadratic and linear programming*, Proceedings of the Third International Conference on Industrial and Applied Mathematics (ICIAM'95), Hamburg, Germany, July 1995.

[KV95] S. O. Krumke and J. Valenta, *Finding tree-2-spanners*, Tech. Report 131, University of Würzburg, Würzburg, Germany, December 1995.

[Law76] E. L. Lawler, *Combinatorial optimization: Networks and matroids*, Holt, Rinehart and Winston, 1976.

[Lee82] D. T. Lee, *On* k*-nearest neighbor Voronoi diagrams in the plane*, IEEE Transactions on Computing **C-31** (1982), 478–487.

[LP86] L. Lovász and M. D. Plummer, *Matching theory*, Akadémiai Kiadó, Budapest (1986), 1986.

[LR88] F. T. Leighton and S. Rao, *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms*, Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'88), 1988, pp. 422–431.

[LV92] J. H. Lin and J. S. Vitter, ε*-approximations with minimum packing constraint violation*, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing (STOC'92), 1992, pp. 771–781.

[LY93] C. Lund and M. Yannakakis, *On the hardness of approximating minimization problems*, Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC'93), May 1993, pp. 286–293.

[Meg83] N. Megiddo, *Applying parallel computation algorithms in the design of serial algorithms*, Journal of the ACM **30** (1983), no. 4, 852–865.

[MF90] P. B. Mirchandani and R. L. Francis, *Discrete location theory*, Wiley-Interscience Series in Discrete Mathematics and Optimization, 1990.

[MNK96] C. Mauckner, H. Noltemeier, and S. O. Krumke, *Finding placement sequences and bin locations for pick and place robots*, (H. W. Hamacher, K. Klamroth, and S. Nickel, eds.), Studies in Locational Analysis, vol. 10, October 1996, pp. 67–89.

[Mot92] R. Motwani, *Lecture notes on approximation algorithms: Volume I*, Tech. Report CS-TR-92-1435, Department of Computer Science, Stanford University, Stanford, CA 94305-2140, 1992.

[MR$^+$95a] M. V. Marathe, R. Ravi, R. Sundaram, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III, *Bicriteria network design problems*, Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP'95), Lecture Notes in Computer Science, vol. 944, 1995, pp. 487–498.

[MR95b] R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.

[Nol75] H. Noltemeier, *Graphentheorie: mit Algorithmen und Anwendungen*, de Gruyter Lehrbuch, 1975.

[Nol81] H. Noltemeier, *Informatik I: Einführung in Algorithmen und Berechenbarkeit*, Carl Hanser Verlag, München Wien, 1981.

[Pap94] C. M. Papadimitriou, *Computational complexity*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[Phi93] C. Phillips, *The network inhibition problem*, Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC'93), May 1993, pp. 288–293.

[PS82] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization*, Prentice-Hall, Inc., 1982.

[PS85] F. M. Preparata and M. I. Shamos, *Computational geometry*, Springer Verlag, New York, Inc., 1985.

[PS95] D. Paik and S. Sahni, *Network upgrading problems*, Networks **26** (1995), 45–58.

[PY91] C. H. Papadimitriou and M. Yannakakis, *Optimization, approximation, and complexity classes*, Journal of computer and system sciences **43** (1991), 425–440.

[Rav93] R. Ravi, *Steiner trees and beyond*, Ph.D. thesis, Brown University, Providence, Rhode Island 02912, USA, 1993.

[Rav94] R. Ravi, *Rapid rumor ramification*, Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science (FOCS'94), November 1994, pp. 202–213.

[RK+93] V. Radhakrishnan, S. O. Krumke, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi, *Compact location problems*, Proceedings of the 13th Conference on Software Technology and Theoretical Computer Science (FSTCS'93), Bombay, India, Lecture Notes in Computer Science, vol. 761, December 1993, pp. 238–247.

[RM+93] R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt III, *Many birds with one stone: Multi-objective approximation algorithms*, Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC'93), May 1993, pp. 438–447.

[RRT94] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi, *Heuristic and special case algorithms for dispersion problems*, Operations Research **42** (1994), no. 2, 299–310, An extended abstract of this paper appeared in *Proceedings of the 1991 Workshop on Algorithms and Data Structures* (WADS'91), Ottawa, Lecture Notes in CS, Vol. 519, Springer-Verlag, New York, NY, August 1991, pp 355–366.

[RS90] N. Robertson and P. Seymour, *Graph minors IV, treewidth and well-quasi-ordering*, Journal of Combinatorial Theory, Series B **48** (1990), 227–254.

[RS+94] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi, *Spanning trees short or small*, Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94), January 1994, complete version to appear in SIAM Journal on Discrete Mathematics, pp. 546–555.

[Tam91] A. Tamir, *Obnoxious facility location on graphs*, SIAM Journal on Discrete Mathematics **4** (1991), 550–567.

[VTL82] J. Valdes, R. E. Tarjan, and E. L. Lawler, *The recognition of series-parallel digraphs*, SIAM Journal on Computing **1** (1982), 1–12.

[War92] A. Warburton, *Approximation of pareto optima in multiple-objective shortest path problems*, Operations Research **35** (1992), no. 1, 70–79.

[ZPD94] Q. Zhu, M. Parsa, and W. Dai, *An iterative approach for delay bounded minimum Steiner tree construction*, Tech. Report UCSC-CRL-94-39, University of California, Santa Cruz, October 1994.

Index