Algorithmic Approaches for Two Fundamental Optimization Problems: Workload-Balancing And Planar Steiner Trees

Diplomarbeit in Mathematik mit Schwerpunkt Informatik
von Siamak Tazari
Technische Universität Darmstadt

Betreuer:
Dr. habil. Matthias Müller-Hannemann

Fachbereichsinterner Betreuer:
Prof. Dr. Alexander Martin

Darmstadt, August 10, 2006

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Stellen, die dem Wortlaut oder Sinne nach anderen Werken entnommen sind, durch Angabe der Quellen als Entlehnungen kenntlich gemacht habe.

Darmstadt, den 10.8.2006

(Siamak Tazari)

Preface

"The wind, it whispers to the buckeyed trees in rhyme.
Well my heart's in the Highlands,
I can only get there one step at a time."
—Bob Dylan, "Highlands"

In today's life, we encounter optimization problems all the time: from a simple everyday question like the fastest way to get from home to work up to large scale problems such as maximizing the profit of a company, there is an ever growing number of optimization problems that are waiting to be solved. Unfortunately, for many of these problems, it is not known how they can be solved *exactly* in a "reasonable" time. Even worse, there is reasonable evidence that a large class of optimization problems do not have "fast" algorithms at all. Still, these problems are present and we need to calculate *some* solution for them — optimal or not. These problems are investigated both from a theoretical and from a practical point of view.

From a *practical* point of view, we approach these algorithmic challenges with the goal of finding "good" solutions in an acceptable time. This can be achieved, for example, by the use of heuristics or by mathematical programming. Ideally, one would also like to be able to know the quality of the calculated solutions by specifying how much they are maximally away from the optimal solution. The work in this area is often classified as experimental algorithms, where one designs and implements algorithms and shows their good quality by making experiments on real or artificial test data and comparing them to other known solutions or theoretical lower and upper bounds.

From a *theoretical* point of view, we would like to design algorithms that give us a certain *performance guarantee*, i.e. algorithms that deliver solutions that are provably within some given factor of the optimum. This leads to the area of approximation algorithms and complexity theory. For some classes of hard optimization problems, there even exist algorithms that calculate solutions that are arbitrarily close to the optimum and run in “theoretically” short time. However, many of these algorithms are still too slow in practice. But still, it is this interplay between theoretical and practical results that makes us able to solve harder and larger optimization problems every day.

In this thesis, we are going to look at both of these views. We investigate two fundamental optimization problems. The first one, workload balancing in multi-stage production processes, belongs to the large class of scheduling problems and we approach it from a practical point of view. We consider a specific application, namely, the optimization of a kind of printed circuit board manufacturing machine. We identify the abstract scheduling problem that has to be solved, show that it is hard, and propose a heuristic to solve it fast in practice. For a special but very important case of the problem, we present an exact algorithm that runs in polynomial time and we use this algorithm to derive our heuristic for the original problem. We argue that our algorithm delivers high quality solutions by showing experimental results on real application data and comparing them with lower bounds delivered by mathematical programming tools.

The second problem that we look at is a classical network-design problem, called the Steiner tree problem. Roughly speaking, in this problem, we want to interconnect a given set of points with a shortest possible network. We study this problem from a theoretical point of view. We look at both the Steiner tree problem in graphs and the geometric Steiner tree problem. Our goal was to find a so-called polynomial time approximation scheme for the Steiner tree problem in planar graphs. The case of planar graphs is highly relevant in VLSI design, since it contains the rectilinear and octilinear Steiner tree problems with obstacles as special cases. Also, the question about the existence of polynomial time approximation schemes in planar graphs is a very important theoretical question in its own right. Even though we did not succeed in finding such an approximation scheme for the Steiner tree problem in planar graphs, we conjecture that one exists. We thoroughly review the relevant literature on this topic and suggest a promising way to achieve the desired result. Specifically, we show that if a specific lemma is proved to be true, then a polynomial time approximation scheme follows.

Throughout this work, we will introduce many algorithmic ideas, methodologies and proof techniques, both in the area of experimental algorithms and in the area of approximation algorithms. So, one can also regard this thesis as giving a broad — and often deep — overview on these topics.

Acknowledgments

First of all, I want to thank my supervisor Dr. Matthias Müller-Hannemann for all the time and effort he spent in advising me on this thesis and always being helpful about every question I had and every problem I encountered throughout this work and beyond. I want to thank Prof. Dr. Karsten Weihe and Prof. Dr. Alexander Martin for integrating me into algorithmic research early on and trusting me with challenging tasks since the first year of my studies at the university. I could always seek their advice on every academic and career decision I faced and they have always provided me with the best of their knowledge and experience. I would also like to thank Dr. Thomas Ihringer and Prof. Dr. Klaus Keimel for the advice, knowledge and experience they passed on to me in numerous conversations throughout my studies. My thanks go to Prof. Dr. Marc Alexa who accepted me as a research assistant even before I had started my courses at the university and brought me in touch with the fascination of mathematically challenging research in computer science. For the time I spent at the University of British Columbia in Vancouver, Canada, I would like to thank Prof. Dr. Holger Hoos, Prof. Dr. Will Evans, Prof. Dr. Anne Condon and Prof. Dr. Kevin Leyton-Brown for being excellent instructors and advisors and also Frank Hutter for encouraging me to go there in the first place. I would also like to thank the "Studienstiftung des Deutschen Volkes" for their constant support throughout my years at the university and specifically for making the exchange year to the University of British Columbia possible for me.

I would like to thank everyone in the algorithmics group of the computer science department at TU-Darmstadt for warmly integrating me into the group and always being helpful about various technical and non-technical problems at work. The same thanks go to the discrete optimization group in the mathematics department and to the graphical interactive systems group of the computer science department for the time I spent working with them. My special thanks go to the Informatics Olympiad group in Iran for the wealth of knowledge I acquired and the fascination I experienced throughout my time with them in that young age that greatly influenced my course in life. And most of all, I would like to thank my parents for the world of love and encouragement they have always been providing me with, being there whenever I needed them, always supporting me with all the means that were available to them, and igniting and keeping alive the fire of my fascination for mathematics and computer science ever since my early childhood.

Contents

Part I:
Workload Balancing in Multi-Stage Production Processes

Chapter 1

Problem Introduction

We consider a variant on the general workload balancing problem, which arises naturally in automated manufacturing and throughput optimization of assembly-lines. The problem is to distribute the tasks over compatible machines and phases of the process simultaneously. The total duration of all phases is to be minimized.

In this chapter, we will first state the abstract formulation of our problem. Then we will describe an application that motivated this research, namely printed circuit board (PCB) manufacturing on certain modular machines, such as Philips/Assembléon's AX[1]. After presenting some related work in this area, we will outline our contribution and give an overview of our results.

Most contents of this part of the thesis are published in [79], which is the joint work of Karsten Weihe, Matthias Müller-Hannemann and myself.

1.1 Problem Statement

In this thesis, we study the following general workload balancing problem.

Problem 1.1 (Main Problem). *We are given tasks $T = \{t_1, \dots, t_n\}$, machines $P = \{p_1, \dots, p_m\}$, phases $Q = \{q_1, \dots, q_s\}$, a reset time R, and a set $A \subseteq T \times Q \times P$ of feasible assignments of tasks to machines and phases. Throughout the paper, a pair $(q, p) \in Q \times P$ will be called a* bucket *(see Fig. 1.1).*

A schedule *is an assignment $f : T \mapsto Q \times P$ of every task to a bucket. A schedule is* feasible *if $(t, f(t)) \in A$ for all $t \in T$.*

[1]AX is a fast component mounter for printed circuit boards, Philips/Assembléon B.V., Eindhoven, The Netherlands.

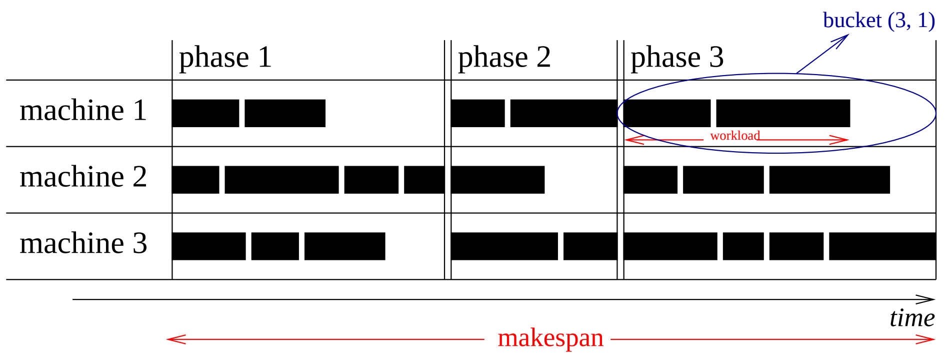

Figure 1.1: Illustration of the makespan. Each box represents a task.

For $t \in T$, let $\ell(t)$ denote the length *of task t and $r_f(q,p)$ the number of times a reset is needed in bucket (q,p) given the schedule f. The* workload *of a machine p in a phase q is the sum of the lengths of the tasks assigned to it plus a possible reset time, i.e.*

$$wl_f(q,p) = \sum_{t \in T, f(t)=(q,p)} \ell(t) + R \cdot r_f(q,p)$$

for a given schedule f. The duration of a phase is the maximum workload of a machine in that phase:

$$d_f(q) = \max_{j=1,\dots,m} \{wl_f(q,p_j)\} \ .$$

The makespan *of the schedule is the sum of the durations of the phases, i.e.*

$$c(f) = \sum_{i=1}^{s} d_f(q_i) \ .$$

The goal is to find a schedule that minimizes the makespan (see Fig. 1.1).

1.2 Application

This problem arises, for example, in throughput optimization of assembly lines. In fact, our original motivation for this work is a particular application of assembly-line balancing: printed circuit board (PCB) manufacturing on certain modular machines such as Philips/Assembléon's AX (Fig. 1.2).

The circuit boards are the workpieces, the stations of the assembly line are the machines. The workpieces are moved forward in steps. In every phase

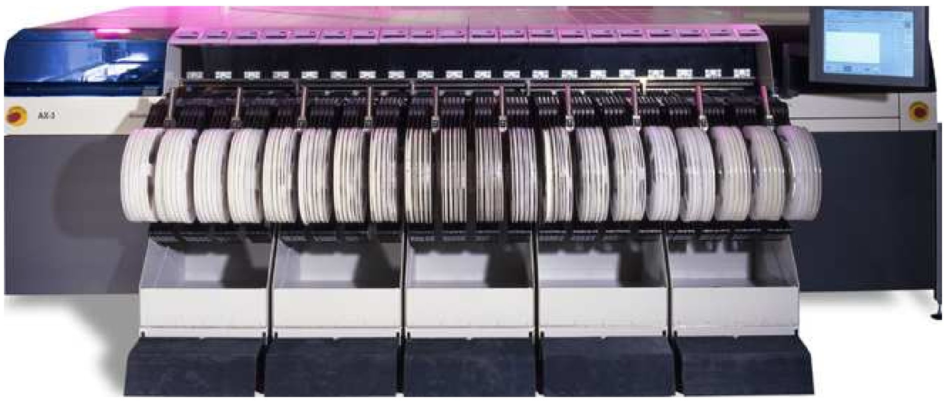

Figure 1.2: Philips/Assembléon's AX-5 consists of 20 pick-and-place stations.

between two moving steps, the machines perform tasks on the workpieces. The moving steps are periodic: after s steps, each workpiece has exactly assumed the position of its predecessor in the line. The actions of a machine are identical on all workpieces, so this is a periodic scheme with period s, too (see Fig. 1.3).

Every machine has a robot arm, which performs the tasks. In a task, a component is picked from a particular position (*feeder*) inside this machine and mounted at a prescribed position on the board. A machine comes with feeders for different component types. The feasibility relation A is induced by the following restrictions. A task may be performed by a machine if the machine has a feeder for the required component type. On the other hand, the task may be processed only in those phases in which the mounting position is in the visibility range of the machine.

The robot arm needs a *toolbit* for picking and holding a component. Different types of components require different toolbits. Each machine has a *toolbit repository*, and the reset time is induced by the necessary exchange of the toolbit at that repository.

A phase is finished once all machines have accomplished all of their tasks that are assigned to this phase. The time to produce one board is the sum of the durations of the s steps (up to a constant value). This time is to be minimized.

1.3 Related Work

Workload balancing (makespan minimization) of parallel machines for a single production phase is a prototypal scheduling problem which has been

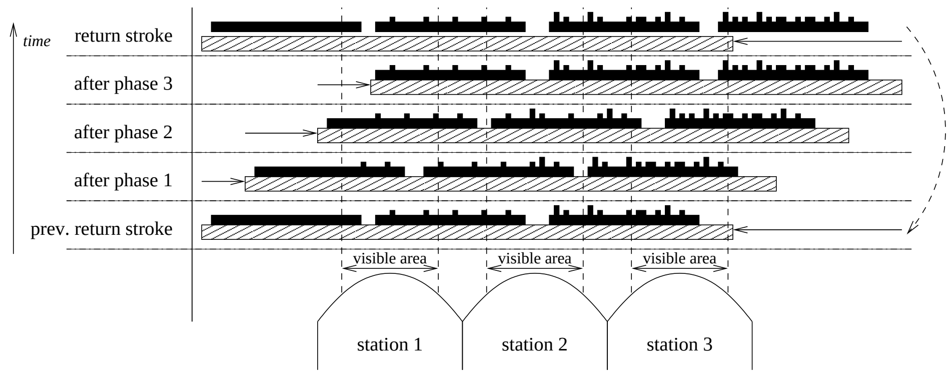

Figure 1.3: Five snapshots of a board cycle with $s = 3$; the arrows on each line indicate the moves of the belt from the previous phase to the current one, respectively. In the lowest row, a new board is mounted on the belt; then, in each of the three phases, the belt moves forward and components get placed on the boards; after the last phase, a finished board leaves the machine, the belt detaches from the boards and returns to its initial position, ready to mount a new board.

widely considered [17]. A number of overview papers on printed circuit board optimization and assembly lines have appeared quite recently. Scholl [77] surveys general problems and approaches to the balancing and sequencing of assembly lines. An overview on printed circuit board assembly problems has been given by Crama et al. [25].

Ayob et. al. [6] compare different models and assembly machine technologies for surface mount placement machines. Multi-station parallel placement machines like the Fuji QP-122 or Assembléon's AX found only little attention. For this type of machine, evolutionary (genetic) algorithms have been proposed in [83]. These authors consider a greatly simplified model in comparison to the one studied in this paper. Their computational results lack a comparison with optimal solutions or lower bounds. Müller-Hannemann and Weihe [65] recently studied a different variant of the workload balancing problem, where for each task exactly one machine is compatible (instead of a subset of the machines as in Problem 1.1). However, in that variant also the movement scheme of the assembly line was part of the optimization.

It is an empirical observation that task lengths differ not by too much in typical applications. Therefore, it is interesting to study the case of uniform task lengths. For example, this has been done by Grunow et al. [38].

Very large-scale neighborhood search has been applied quite successfully to several hard combinatorial optimization problems [1], in particular to the classical machine scheduling problems [32].

The problem of optimizing the setup of placement machines has been studied in the thesis of Gaudlitz [36]. Some of our methods are used in his work to evaluate the quality of the resulting setups.

1.4 Our Contribution and Overview

In Chapter 2 we show that our problem is $\mathcal{NP}$-hard in the strong sense, even if all tasks are of unit length and the reset time is zero. In Chapter 3, we will present a novel algorithmic approach for this type of problem. In the first stage of our algorithm, we will compute an optimal solution to the special case of unit task-lengths and zero reset time. This solution will be used in the second stage as a start solution to compute a good solution to the original problem. As noted above, this special case is of interest in its own right.

Our research on this special case led to the following nice byproduct: for a fixed number of phases, Problem 1.1 can be solved in polynomial time if all tasks are of unit length and there is no reset time. We would like to note that if the tasks are not of unit-length, the problem remains $\mathcal{NP}$-hard even when we have zero reset time and only 2 phases [65].

The first stage of the algorithm is based on a reduction to a certain max-flow problem. Basically, the second stage may be characterized as sort of a shortest-path based multi-exchange local search where we generalize and extend the work of Frangioni et al. [32].

It might be worth mentioning that our handling of the reset times is quite generic and may be easily adapted to other types of complications. This observation demonstrates that our algorithm is quite robust in the sense that various types of complications may be naturally incorporated.

A major part of this work is an extensive computational study on all real-world examples that are available to us (Chapter 4). We compare our solution to an integer linear programming (ILP) based approach using the ILP solver of CPLEX. Typically, our algorithm comes close to the lower bound computed by CPLEX or even hits the optimal value although it is faster than CPLEX by orders of magnitude.

The comparatively small run time of our algorithm is particularly interesting for the complementary problem, to determine a favorable setup of the assembly line. In fact, here our problem naturally occurs as a subproblem, in which the quality of a trial setup is evaluated by the quality of the induced optimal task distribution [36]. In this context, the algorithm is to be applied repeatedly a large number of times, so a small run time is mandatory.

Chapter 2

$\mathcal{NP}$-Hardness and ILP Formulation

In this chapter, we will first prove our hardness result. Then we will present the ILP formulation that we used to compare our computational results to CPLEX.

2.1 $\mathcal{NP}$-Hardness

Theorem 2.1. *Problem 1.1 is $\mathcal{NP}$-hard in the strong sense, even if all tasks are of unit length and the reset time is zero.*

Proof. We reduce the minimum vertex cover problem to an instance of Problem 1.1 with unit task-lengths and no reset time. The vertex cover problem is well-known to be $\mathcal{NP}$-hard [45].

Let an undirected graph $G = (V, E)$ be given. Construct an instance of Problem 1.1 in the following way: let $n = m = |E|$ and $s = |V|$, i.e. for every edge we introduce a task and a machine and for every vertex we create a phase. Let $T = \{t_e : e \in E\}$, $P = \{p_e : e \in E\}$, and $Q = \{q_v : v \in V\}$ denote the task, machine, and phase sets, respectively. For every task t_e with $e = \{v, w\}$, we allow exactly 2 buckets on each machine p_e: the buckets (q_v, p_e) and (q_w, p_e). Thus, the set of feasible assignments is given by

$$A = \{(t_e, q_v, p_e), (t_e, q_w, p_e) : e \in E, e = \{v, w\}\} \ . \tag{2.1}$$

We claim that for every vertex cover $W \subseteq V$ there exists a schedule with $c(f) \leq |W|$ and for every solution to (2.1) there exists a vertex cover W in

G with $c(f) = |W|$. If this is true, then any function f minimizing c also results in a minimum vertex cover and the theorem is proved.

Note in (2.1) that every machine is given only one job; we just have to decide in which one of the given two phases to do it. Thus, the duration of every phase is at most one, i.e.

$$(\forall q \in Q)\, d_f(q) \leq 1 \ . \tag{2.2}$$

Suppose a vertex cover $W \subseteq V$ is given. For every edge e we can find a vertex $\mathrm{cov}(e) \in W$ which covers it; if an edge is covered by 2 vertices in W, we can just take any one. We set

$$(\forall t_e \in T)\, f(t_e) = (q_{\mathrm{cov}(e)}, p_e) \ . \tag{2.3}$$

In this way, we get a feasible assignment for every task in T. Furthermore, no task is done in a phase corresponding to a vertex in $V \setminus W$ and we get

$$c(f) = \sum_{v \in V} d_f(q_v) = \sum_{v \in W} d_f(q_v) \overset{(2.2)}{\leq} \sum_{v \in W} 1 = |W| \ . \tag{2.4}$$

Now suppose a feasible assignment function f is given. We set $W = \{v : d_f(q_v) = 1\}$. Then we get

$$c(f) = \sum_{v \in V} d_f(q_v) \overset{(2.2)}{=} \sum_{v \in W} d_f(q_v) = \sum_{v \in W} 1 = |W| \ . \tag{2.5}$$

But W also covers all edges in G: For an edge $e = \{v, w\} \in E$, we have $f(t_e) = (q_v, p_e)$ or $f(t_e) = (q_w, p_e)$. Then $d_f(q_v) = 1$ or $d_f(q_w) = 1$ and thus $v \in W$ or $w \in W$ and thus, e is covered. $\square$

This proof suggests that the parameter that makes this problem hard is the number of phases, s. Indeed, for any fixed s, the problem can be solved in polynomial time as we will show in the next chapter.

2.2 Integer Linear Programming Model

To obtain lower bounds in our computational study and to compare our algorithm to CPLEX, we modeled our problem — applied to the scenario of PCB manufacturing on the AX with toolbit exchanges — as an integer linear program. This model is basically taken from [36]. The toolbit exchanges induce the reset times and modeling them correctly is somewhat complicated as can be seen below.

We use the following **variables:**

- d_i denotes the duration of phase i
- $w_{i,j}$ denotes the workload of bucket (i, j)
- $r_{i,j}$ is the number of toolbit exchanges (i.e. reset times) in bucket (i, j)
- $f_{t,i,j} \in \{0, 1\}$ decides if task t is assigned to bucket (i, j)
- $x_{b,i,j} \in \{0, 1\}$ decides if toolbit b is used in bucket (i, j)
- $x^f_{b,i,j} \in \{0, 1\}$ specifies if toolbit b is the first toolbit used in bucket (i, j)
- $x^l_{b,i,j} \in \{0, 1\}$ specifies if toolbit b is the last toolbit used in bucket (i, j)
- $x^{f \vee l}_{b,i,j} \in \{0, 1\}$ is 1 iff toolbit b is the first or last toolbit used in bucket (i, j)
- $x^{\text{prec}}_{b,i,j} \in \{0, 1\}$ is 1 iff toolbit b is the first one of this phase or the last one of the preceding phase in bucket (i, j)
- $x^u_{i,j} \in \{0, 1\}$ tells if there is exactly one toolbit used in bucket (i, j)
- $x^{f=l}_{i,j} \in \{0, 1\}$ tells if bucket (i, j) uses at least 2 different toolbits and the first toolbit equals the last one

For all the indices in this section, we have $i = 1, \ldots, s$, $j = 1, \ldots, m$, $t = 1, \ldots, n$ and $b = 1, \ldots, h$ where h denotes the number of available toolbits.

The **objective** is to minimize the makespan, i.e.,

$$\min \sum_{i=1}^{s} d_i$$

subject to the following **constraints**:

- the duration of a phase is the maximum workload over all machines:

$$\forall i, j \qquad d_i \geq w_{i,j}$$

- the workload of a bucket is the sum of the task-lengths performed in that bucket plus reset times:

$$\forall i, j \qquad w_{i,j} = r_{i,j} \cdot R + \sum_{t=1}^{n} f_{t,i,j} \cdot \ell(t)$$

- the number of toolbit exchanges in a bucket consists of (i) the number of toolbits used in the bucket minus 1; (ii) an additional exchange if the first toolbit differs from the last one from the preceding phase; (iii) an additional exchange if there are at least 2 toolbits used and the first and the last one are the same (in this case, this toolbit has been counted only once in the first term and has to be added here):

$$\forall i,j \qquad r_{i,j} = (\sum_{b=1}^{h} x_{b,i,j}) - 1 + (\sum_{b=1}^{h} x^{\text{prec}}_{b,i,j}) - 1 + x^{f=l}_{i,j}$$

- all tasks must be assigned exactly once:

$$\forall t \qquad \sum_{i=1}^{s}\sum_{j=1}^{m} f_{t,i,j} = 1$$

- a task may only be assigned to a bucket if it is feasible in that bucket:

$$\forall (t,i,j) \notin A \qquad f_{t,i,j} = 0$$

- a toolbit may only be used in a bucket if it is available to that machine:

$$\forall b,i,j \qquad x_{b,i,j} = 0 \text{ if toolbit } b \text{ not available in bucket } (i,j)$$

- a task may only be assigned to a bucket if a compatible toolbit is used:

$$\forall t,i,j \qquad f_{t,i,j} \leq \sum_{b \text{ compat. with } t} x_{b,i,j}$$

- every bucket uses at least one toolbit:

$$\forall i,j \qquad \sum_{b=1}^{h} x_{b,i,j} \geq 1$$

- there is a first/last toolbit associated with every bucket:

$$\forall i,j \qquad \sum_{b=1}^{h} x^{f}_{b,i,j} \geq 1$$
$$\forall i,j \qquad \sum_{b=1}^{h} x^{l}_{b,i,j} \geq 1$$

- a first/last toolbit must be used in the bucket:

$$\forall b,i,j \qquad x^{f}_{b,i,j} \leq x_{b,i,j}$$
$$\forall b,i,j \qquad x^{l}_{b,i,j} \leq x_{b,i,j}$$

- set $x^{f\vee l}_{b,i,j} = 1$ iff toolbit is the first or last one in this bucket:

$$\begin{aligned} \forall b,i,j &\qquad x^{f\vee l}_{b,i,j} \leq x^{f}_{b,i,j} + x^{l}_{b,i,j} \\ \forall b,i,j &\qquad x^{f\vee l}_{b,i,j} \geq x^{f}_{b,i,j} \\ \forall b,i,j &\qquad x^{f\vee l}_{b,i,j} \geq x^{l}_{b,i,j} \end{aligned}$$

- set $x^{\text{prec}}_{b,i,j} = 1$ iff toolbit is first one of this phase or last one of preceding phase in this bucket (where $i-1$ has to be replaced by s if $i = 1$):

$$\begin{aligned} \forall b,i,j &\qquad x^{\text{prec}}_{b,i,j} \leq x^{f}_{b,i,j} + x^{l}_{b,i-1,j} \\ \forall b,i,j &\qquad x^{\text{prec}}_{b,i,j} \geq x^{f}_{b,i,j} \\ \forall b,i,j &\qquad x^{\text{prec}}_{b,i,j} \geq x^{l}_{b,i-1,j} \end{aligned}$$

- set $x^{u}_{i,j} = 1$ iff a bucket uses exactly one toolbit:

$$\begin{aligned} \forall i,j &\qquad h \cdot (1 - x^{u}_{i,j}) + 1 \geq \sum_{b=1}^{h} x_{b,i,j} \\ \forall i,j &\qquad 2 - h \cdot x^{u}_{i,j} \leq \sum_{b=1}^{h} x_{b,i,j} \end{aligned}$$

- set $x^{f=l}_{i,j} = 1$ iff bucket uses at least 2 different toolbits and the first one equals the last:

$$\forall i,j \qquad x^{f=l}_{i,j} = 2 - x^{u}_{i,j} - \sum_{b=1}^{h} x^{f\vee l}_{b,i,j}$$

Chapter 3

Our Approach

In Section 3.1, we will first give an overview of our approach. In particular, we will formally state the subproblems that constitute the ingredients of our approach. In Sections 3.2 – 3.6, we will consider all of these ingredients individually and show how they build up on each other to produce a solution to our main problem. Specifically, in Section 3.4, we show that for a fixed number of phases, our problem can be solved in polynomial time if all tasks are of unit length and there is no reset time.

3.1 The Big Picture

Our algorithm consists of two stages: in the first stage, Sections 3.2 - 3.4, we will focus on the special case of unit task-lengths and zero reset time:

Problem 3.1 (Uniform Case). *Solve Problem 1.1 with the assumption that* $\ell(t) = 1$ *for all* $t \in T$ *and* $R = 0$*.*

The structure of our approach can be seen in Fig. 3.1. It is based on a network flow model introduced in Section 3.2. We will derive a fast heuristic for Problem 3.1 in Section 3.3 that, in fact, finds the exact solution to the following variant of the problem, which considers a different objective function:

Problem 3.2 (MinMax Variant). *Solve Problem 3.1 replacing the objective function with*

$$c_{\max}(f) = \max_{i=1,\dots,s} \{d_f(q_i)\} \ .$$

We will use the solution delivered by the MinMax heuristic as an input to the algorithm in Section 3.4, which solves the uniform case optimally.

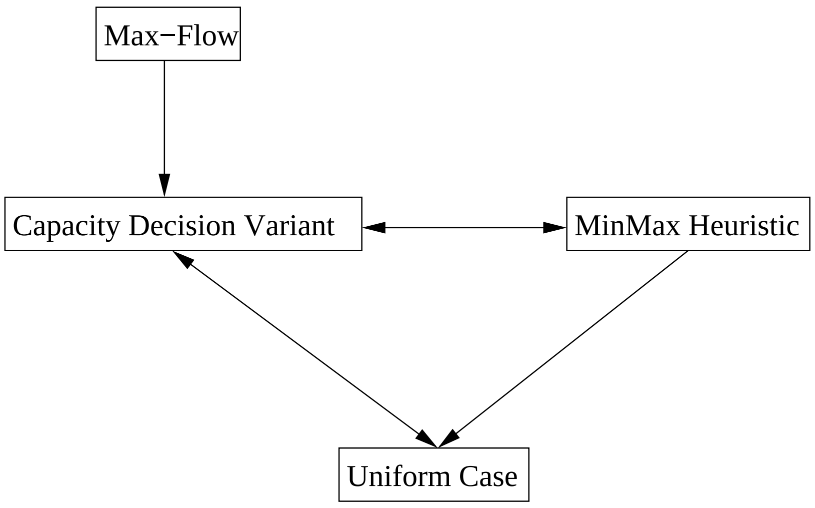

Figure 3.1: The solution to the uniform case takes the solution of the MinMax heuristic as input and repeatedly uses the capacity decision variant to find the best possible capacity function. This is, in turn, based on the network flow model of our problem.

Throughout our work, we make use of a function to control the workload in every phase by an upper bound. We call such a function $\kappa : Q \mapsto \mathbb{Z}$ a *capacity function*, and the upper bound it imposes on a phase, the *capacity* of that phase (see Fig. 3.2). A capacity function is called *feasible* if, and only if, there exists a schedule f such that

$$d_f(q) \leq \kappa(q) \text{ for all } q \in Q \ .$$

In this case, we also say that f is feasible for κ. The following problem plays a central role in our approach:

Problem 3.3 (Capacity Decision Variant). *Given the setting in Problem 3.1 and a capacity function κ, decide whether κ is feasible.*

Let $c(\kappa) = \sum_{i=1}^{s} \kappa(q_i)$ and $c_{\max}(\kappa) = \max_{i=1,\ldots,s} \kappa(q_i)$. Note that if f is a feasible schedule for κ we have

$$c(f) \leq c(\kappa) \text{ and } c_{\max}(f) \leq c_{\max}(\kappa) \ .$$

We use the solution of the uniform case as a start solution for the second stage of our algorithm (Sect. 3.6), which is basically a multi-neighborhood local search. Here we will finally address Problem 1.1 in full generality.

Note that the check whether an instance is feasible is trivial: there is a feasible schedule if, and only if, every task may be assigned to *some* bucket. We assume that this is always given.

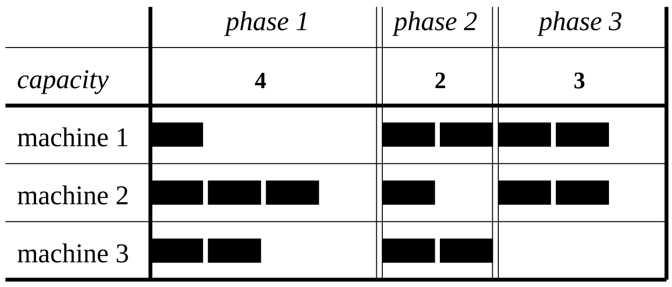

Figure 3.2: A feasible capacity function

3.2 The Network Flow Model and the Capacity Decision Variant

Given an instance of the capacity decision variant, we construct a directed network flow graph $G_\kappa = (V, E)$ as follows (see Fig. 3.3). We introduce one vertex for every task, one vertex for every bucket, and in addition one source vertex and one sink vertex. Next we add one edge with capacity 1 from the source to every task vertex, one edge with capacity 1 from every task vertex to every bucket vertex in which this task may be performed, and one edge with capacity $\kappa(q)$ from every bucket vertex with phase q to the sink. We denote the value of a flow $\overline{f}$ in G with $v(\overline{f})$. It is easy to see that the following lemma holds.

Lemma 3.4. *There exists a feasible flow $\overline{f}$ in G_κ with $v(\overline{f}) = n$ if and only if κ is feasible. If such a flow exists, a feasible schedule f for the capacity decision variant can be derived as follows: for $t \in T$, set $f(t) = (q, p)$ for the unique pair (q, p) such that $\overline{f}((v, w)) = 1$, where node v represents the task t and node w represents the bucket (q, p).*

This results in an algorithm whose runtime, using the Ford-Fulkerson method [31], is $O(n \cdot |E|) = O(n^2ms)$. This is because the value of the flow is at most n, so it can be constructed by at most n augmentations.

3.3 The MinMax Variant

We can now easily derive an algorithm for the MinMax variant. As said in the big picture (Sect. 3.1), this is the first step of our overall algorithm and its result will be used as a starting point for solving the uniform case optimally (Sect. 3.4).

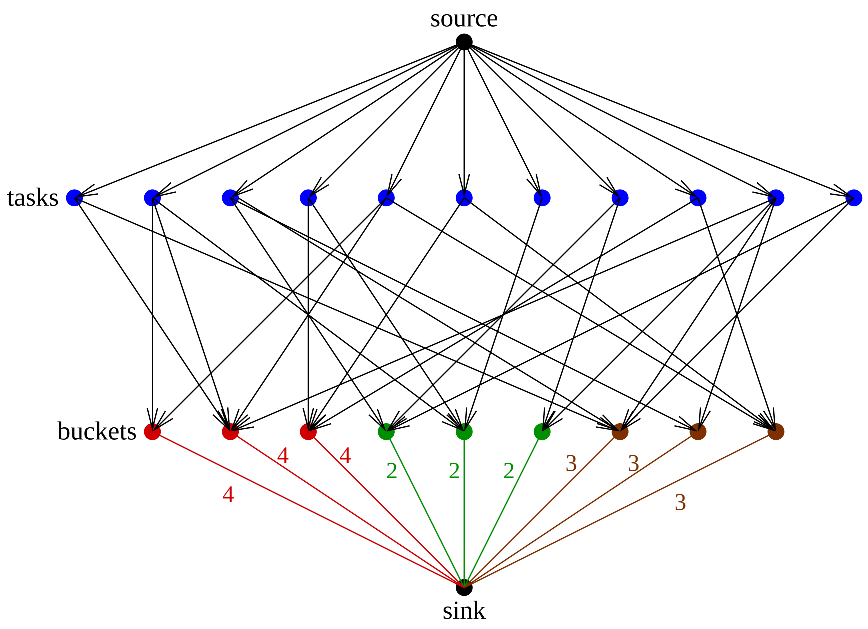

Figure 3.3: The network flow model for an instance with 3 machines and 3 phases. The first 3 buckets (red) belong to phase 1 with capacity 4, the next 3 (green) belong to phase 2 with capacity 2 and the last 3 (brown) belong to phase 3 with capacity 3.

The MinMax problem itself is equivalent to finding the minimal value k such that the uniform capacity function $\kappa(\cdot) \equiv k$ is feasible. For that, we may simply test all values $k = 0, \ldots, n$ and take the smallest one for which κ becomes feasible. Alternatively, we may use binary search on $0, \ldots, n$ and obtain a complexity of $O(n^2 ms \log n)$.

In view of Sect. 3.4, it is heuristically reasonable to construct a tighter capacity function κ, which is not necessarily uniform, in the hope to also minimize the sum. We can continue decreasing the capacities of individual phases as long as it is possible. This idea is carried out in Algorithm 3.1 and illustrated in Figure 3.4.

For this purpose, we make use of a helper-function: Given G_κ, one can easily write a function `decreasePhaseCapacity(`q`)` that tries to decrease the capacities of all the sink-edges belonging to phase q by one unit while preserving the value of the flow; it should change the function κ accordingly and return `true` if successful, otherwise do nothing and return `false`. A straightforward implementation runs in $O(m \cdot |E|) = O(nm^2 s)$-time.

The algorithm tries to decrease the capacities of the phases at most n times, i.e. `decreasePhaseCapacity` is called at most ns times. So the overall runtime is $O(n^2 m^2 s^2)$. This is very fast if m and s are small; and this is the case in some practical situations like in our application.

Algorithm 3.1: MinMax Heuristic

```
Input  : An instance of Problem 3.1 / 3.2.
Output: Optimal assignment w.r.t. Problem 3.2;
        at the same time, heuristic assignment w.r.t. Problem 3.1.
begin
    create a set $F = \emptyset$ of phases with fixed capacity;
    create a capacity function $\kappa$ and set it constant equal to $n$;
    solve Prob. 3.3 for $\kappa$ to find an initial feasible solution;
                        // this also gives us $G_\kappa$ and the flow $\overline{f}$
    repeat
        forall $q \in Q \setminus F$ do
            if not decreasePhaseCapacity $(q)$ then
                $F = F \cup \{q\}$;
    until $F = Q$ ;
    use Lemma 3.4 to derive a feasible assignment $f$ from $\overline{f}$;
    return $f$;
end
```

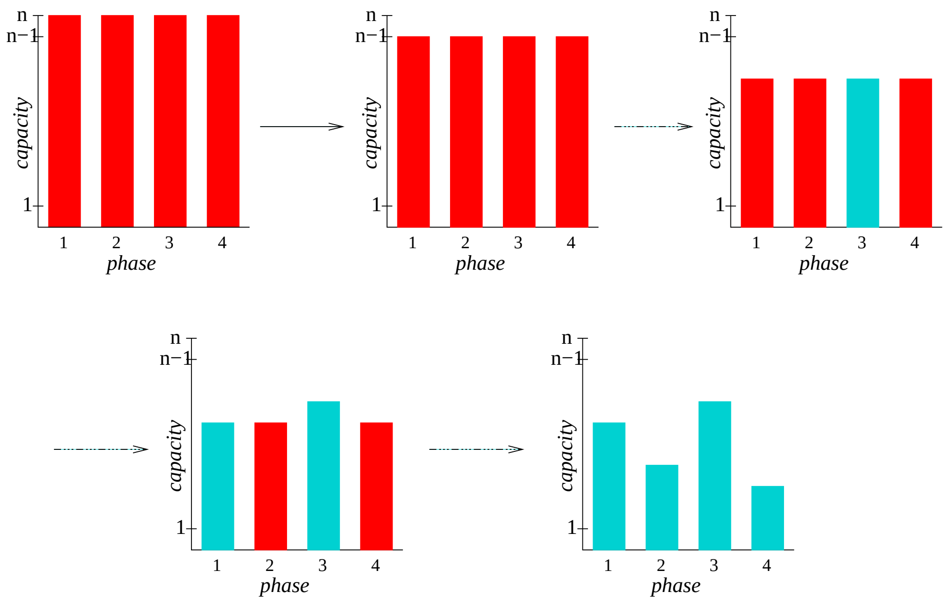

Figure 3.4: The idea of our MinMax heuristic is to decrease the capacities of the phases simultaneously and freeze (blue in the picture) each one that can not be decreased any-more.

3.4 Solving the Uniform Case Optimally

By Theorem 2.1, we know that Problem 3.1 is $\mathcal{NP}$-hard. We found out that the hardness of the problem lies in the parameter s, the number of phases. The algorithm we present now runs in $O(u^{s-1}nm^2s)$-time (where $u \leq n$ is a parameter we will introduce shortly). If we take s to be a fixed parameter, this is polynomial. Indeed, in our application we have $s \leq 7$ and $m \leq 20$ and hence our algorithm is fast enough for our purpose. Also, it is possible to stop it at any time and take the best solution it has found so far.

Our algorithm is basically a branch-and-bound method that searches over the space of all feasible capacity functions for the one with the minimum sum. There are n^s possible capacity functions. Using the network flow model, one can test the feasibility of a capacity function in $O(n^2ms)$-time, so an exhaustive search would end up with a runtime of $O(n^{s+2}ms)$. Now let κ_0 be the capacity function returned by the MinMax algorithm (Alg. 3.1). We know there exists a solution with value $c(\kappa_0)$. Let $u = c(\kappa_0) - 1$. Now we need only search in the space of capacity functions summing up to a value $\leq u$. There are $\binom{u+s}{s}$ possibilities and again, each one has to be tested. We now show how to search in this space and dynamically manipulate the flow graph to achieve a theoretical runtime of $O(u^{s-1}nm^2s)$ and what bounding strategy we use that leads to very short runtimes in practice (see computational results in Chapter 4). Bear in mind that $u \leq n$ and is usually much less than n — in fact, it is usually equal or very close to optimum as can be seen in the computational results.

We first give an informal description of the idea of our method. Consider this table:

phase	1	2	...	s
capacity	$0 \ldots u$	$0 \ldots u$	...	$0 \ldots u$

Our algorithm chooses values for the capacities of the phases from right to left. It tries the values u *downto* 0 for the phase s, then for each one, recursively tries all values for phase $s - 1$ and so on. It always maintains the flow graph G_κ and every time the value of the capacity of a phase is changed, it dynamically changes the capacities of the corresponding edges in G_κ and re-maximizes the flow. This way, the feasibility check can be performed efficiently, in fact, "automatically". The interesting thing about our method is that the return value of a recursive call can be used to bound the solution in the upcoming subtrees and hence, effectively cut away a lot of them (see Fig. 3.5).

More formally, let us denote $\sum_{i=1}^{j} \kappa(q_i)$ by $c_j(\kappa)$. A call to our method must be supplied by three parameters: j, (a not necessarily feasible) κ and u. Suppose we have already chosen the values for $\kappa(q_s), \ldots, \kappa(q_{j+1})$ and would like to find the minimum value for $c_j(\kappa)$ such that κ is feasible. Let

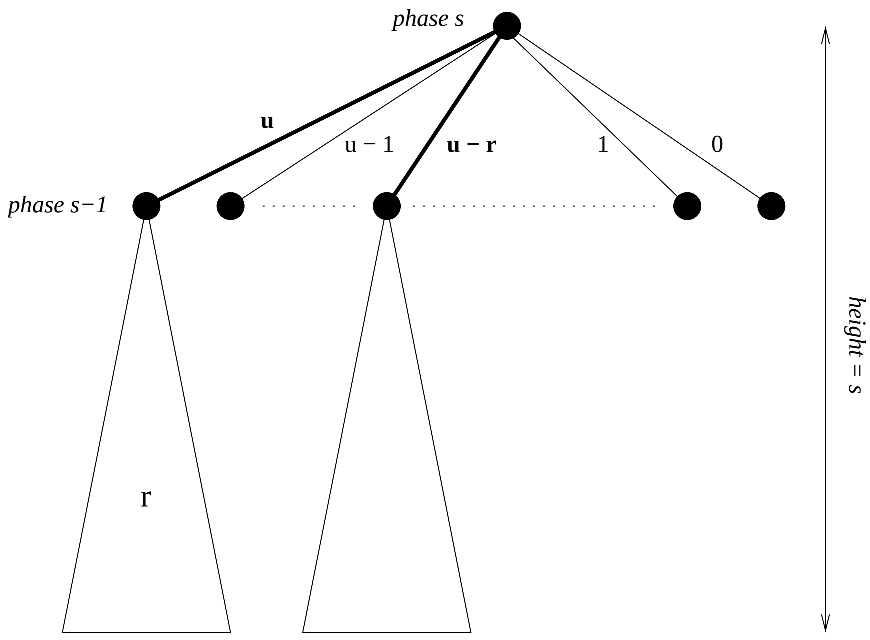

Figure 3.5: The idea of our bounding in the search tree: only the bold branches will be followed down.

us denote this value by $\min_j(\kappa)$. This is the return value of our method. The third parameter of our method, u, indicates the maximum value we expect for $\min_j(\kappa)$. If we determine that $\min_j(\kappa) > u$, our algorithm will return ∞. To obtain a value for u for our initial call, we can take any feasible capacity function κ_0 and set $u := c_j(\kappa_0) - 1$. As stated earlier, we use the solution delivered by our MinMax algorithm. The smaller the value of u, the faster will be our method; hence, it is fortunate that Alg. 3.1, when used as a heuristic for Problem 3.1, supplies near-optimum solutions.

Our bounding is based on the following observation:

Observation 3.5. *Let* $\kappa(q_j) = t$ *and* $\min_{j-1}(\kappa) = r$. *If we decrease* $\kappa(q_j)$ *to any value* $\leq t$*, we will have* $\min_{j-1}(\kappa) \geq r$.

In other words, when we try the capacities for phase j from large to small (i.e. from s downto 0), the return value of our recursive function call (i.e. $\min_{j-1}(\kappa)$) can only increase. This is because any κ with $\kappa(q_j) \leq t$ remains feasible if we set $\kappa(q_j) = t$. So we know that if the current best solution has value $u+1$ and our last recursive call has returned the value $r := \min_{j-1}(\kappa)$, then the next value we should try for $\kappa(q_j)$ is $u - r$; all larger values can not possibly lead to better solutions (see Fig. 3.5).

Our algorithm works as follows: we set $\kappa(q_j) = u$ and find $r := \min_{j-1}(\kappa)$. By the observation above, we know that in any solution with

$c_j(\kappa) \leq u$, $\kappa(q_j)$ can be at most $u - r$. So, we assign this value to $\kappa(q_j)$, calculate $r = \min_{j-1}(\kappa)$ with this new fixation and repeat this procedure until either a solution with $c_j(\kappa) \leq u$ is found, u becomes less than r or no feasible κ with the given fixations exists anymore. If a solution is found, we decrease the value of u and continue the search. Note that there is no need to restart the search in this case.

We call our method `minimizeSum(`j`, `u`, var `κ`)` and it is summarized in Alg. 3.2. The parameter κ is passed by reference and we assume that G_κ and a maximum flow $\overline{f}$ in G_κ are given with it. Our method minimizes $c_j(\kappa)$ and returns it, adjusting κ, G_κ and $\overline{f}$ accordingly. But only if such a feasible solution exists and is at most u. Otherwise the method returns ∞. We need two additional simple methods: `increasePhaseCapacity(`q`)` that increases the capacity of phase q by one and `setPhaseCapacity(`q`, `t`)` that sets the capacity of phase q to the value t; in contrast to `decreasePhaseCapacity`, neither of these methods need to preserve the value of the flow but only have to re-maximize it.

Algorithm 3.2: `minimizeSum(`j`, `u`, var `κ`)`

Input : j: the index of the current phase,
u: the maximum value expected for $\min_j(\kappa)$,
var κ: the current capacity function, passed by reference.
Output: $\min_j(\kappa)$, if it is $\leq u$; otherwise returns ∞;
κ, G_κ and $\overline{f}$ are changed accordingly.

```
1  begin
2      if $j = 2$ then return minimizeSum2 ($u$, $\kappa$); // described below
3      create new capacity function $\kappa^\star$ and set it equal to null;
4      setPhaseCapacity ($q_j$, $u$);                       // modifies $G_\kappa$
5      $r$ = minimizeSum ($j-1$, $u$, $\kappa$);              // recursive call
6      while $r \leq u$ do
7          if $\kappa(q_j) + r \leq u$ then                   // solution is found
8              $\kappa^\star = \kappa$;                      // store new best solution
9              $u = \kappa(q_j) + r - 1$;                         // reduce $u$
10         else
11             setPhaseCapacity ($q_j$, $u - r$);            // modifies $G_\kappa$
12             $r$ = minimizeSum ($j-1$, $u$, $\kappa$);      // recursive call
13     if $\kappa^\star = null$ then return $\infty$;          // no solution found
14     $\kappa = \kappa^\star$; adjust $G_\kappa$ and $\overline{f}$ accordingly;
15     return $u + 1$;
16 end
```

Lemma 3.6. *Algorithm 3.2 is correct and its complexity is $O(u^{j-1}nm^2s)$.*

Proof. We prove by induction on j. We get the base case, $j = 2$, by Lemma 3.8. Suppose Lemma 3.6 holds for values less than j. We show the correctness of the loop in line 6. The loop invariants are every time line 6 is reached :

(i) $\kappa^\star$ is `null` or $c_j(\kappa^\star) = u + 1$ and $\kappa^\star$ stores best solution found so far;

(ii) $r = \min_{j-1}(\kappa)$ (or ∞ if $\min_{j-1}(\kappa) > u$);

(iii) if $\kappa(q_j)$ is *increased*, we will have $\kappa(q_j)+\min_{j-1}(\kappa) \geq u+1 \geq \min_j(\kappa)$.

Remember that the value of $\min_{j-1}(\kappa)$ depends on the value of $\kappa(q_j)$ and therefore, could change when $\kappa(q_j)$ is changed. Condition (i) holds because the values of $\kappa^\star$ and u will only be changed together in lines 8 and 9 and this occurs only when a new best solution is found. The correctness of condition (ii) follows from the induction hypothesis. Together with Observation 3.5 we see that if $\kappa(q_j)$ is *decreased*, then $\min_{j-1}(\kappa)$ will have a value $\geq r$. So, if $\min_j(\kappa)$ is indeed $\leq u$, then $\kappa(q_j)$ must be $\leq u - r$. Hence, the assignment in line 11 is correct and so is condition (iii).

When the loop is finished, we have $r > u$ (line 6), so with condition (iii) we see that there are no possible better solutions to look for. From condition (i) and (iii) and since the value of $\kappa(q_j)$ is never increased during the loop, it follows that the optimal solution, i.e. $\min_j(\kappa)$, will always be found, if its value is $\leq$ the initial value of u. Otherwise $\kappa^\star$ will be `null` and ∞ is returned, as expected.

The loop variant is $\kappa(q_j)$, which decreases by at least one unit every time the loop is executed: if the condition in line 7 holds, then u will be reduced in line 9 otherwise it means that r has just increased; both cases imply that $\kappa(q_j)$ decreases in line 11. Thus, the whole loop will be repeated at most u times.

By the argument above, we also see that a recursive call takes place at most $u+1$ times and this is the dominating term in the complexity analysis. So, with the induction hypothesis we get a runtime of $O(u^{j-1}nm^2s)$.

□

As already mentioned, in order to solve the uniform case, we first run Alg. 3.1, our MinMax algorithm, to get an initial κ. We set $u = c(\kappa) - 1$ and run `minimizeSum(`s`, `u`, `κ`)`. By Lemma 3.6, this call will return us the optimal solution to Problem 3.1. Altogether we get

Theorem 3.7. *For a fixed number of phases, the uniform case of our problem can be solved in polynomial time, specifically in time $O(u^{s-1}nm^2s)$.*

The Case of Two Phases

The recursion above reaches its base case when j becomes 2. In this case, the specialized function `minimizeSum2` is called in line 2 of Alg. 3.2. This function is basically the same as `minimizeSum` with the only difference that the recursive function calls are replaced by some direct manipulation of the flow graph. These changes are sketched in Alg. 3.3 below.

Algorithm 3.3: `minimizeSum2(`u`, var `κ`)`

```
Input  : $u$: the maximum value expected for $\min_2(\kappa)$,
         var $\kappa$: the current capacity function, passed by reference.
Output: $\min_2(\kappa)$, if it is $\leq u$; otherwise returns $\infty$;
         $\kappa$, $G_\kappa$ and $\overline{f}$ are changed accordingly.
same as Alg. 3.2 except the following:
the recursive call in line 5 is replaced by:
   setPhaseCapacity ($q_1$, 0) ;
   while $v(\overline{f}) < n$ and $\kappa(q_1) \leq u$ do increasePhaseCapacity ($q_1$) ;
   $r = \kappa(q_1)$ ;
the recursive call in line 12 is replaced by:
   while $v(\overline{f}) < n$ and $\kappa(q_1) \leq u$ do increasePhaseCapacity ($q_1$) ;
   $r = \kappa(q_1)$ ;
```

Lemma 3.8. *Algorithm 3.3 is correct and its runtime is $O(unm^2s)$.*

Proof. We have replaced the call to `minimizeSum(1, `u`, `κ`)` by its equivalent: we set the capacity of phase 1 to 0 and increase it until we get a feasible flow, i.e. one with $v(\overline{f}) = n$. The only important point is that resetting the capacity back to 0 is only required once in the beginning. After that we know by Observation 3.5 that its value might only increase, so we do not need to reset it. The correctness follows now from Lemma 3.6.

The runtime of `minmizeSum2(`κ`, `u`)` gets dominated by the runtime of its main loop. Throughout the loop, the capacity of q_2 is decreased at most by u units and the capacity of q_1 gets increased at most by u units. This is equivalent to calling `increase/decreasePhaseCapacity` at most $O(u)$ times (sometimes through `setPhaseCapacity`). As discussed in Section 3.3 , these functions have a runtime of $O(nm^2s)$. Thus the overall runtime of our method is $O(unm^2s)$, which is exactly what we needed for the base case of our induction. □

3.5 Lower Bounds

Having good lower bounds can be useful to cut down the search in the branch-and-bound algorithm described above. An easy global lower bound for Problem 3.1, also mentioned in [38], can be given by $c(f) \geq lb_1 := \lceil \frac{n}{m} \rceil$. Using our MinMax algorithm, we can find additional lower bounds: suppose we have a capacity function κ and would like to minimize $c_j(\kappa)$ while keeping the capacities of $q_{j+1}, \ldots, q_s$ fixed (as in the input of Alg. 3.2). We can find a lower bound lb_2^j for $c_j(\kappa)$ by merging the phases $q_1, \ldots, q_j$ into one phase q'. We set

$$Q' = \{q', q_{j+1}, \ldots, q_s\}$$

and

$$A' = \{(t, q_i, p) \in A : \, i > j\} \cup \{(t, q', p) : \, (\exists 1 \leq i \leq j) \, (t, q_i, p) \in A\} \ .$$

Let κ' be the corresponding capacity function. We can minimize $\kappa'(q')$ — while keeping $\kappa'(q_{j+1}) = \kappa(q_{j+1}), \ldots, \kappa'(q_s) = \kappa(q_s)$ fixed — using a simplified version of Alg. 3.1: just set the capacity of q' equal to n and decrease it until this is not further possible. Let $f^\star$ be the assignment resulting from this minimization. For arbitrary values of $\kappa(q_1), \ldots, \kappa(q_j)$, let f be some feasible assignment for κ. We can construct an assignment f' for the merged problem by setting

$$(\forall t \in T) f'(t) = \begin{cases} f(t) & \text{if } f(t) = (q_i, p) \text{ and } i > j \\ (q', p) & \text{if } f(t) = (q_i, p) \text{ and } i \leq j \end{cases} \ .$$

Then we have

$$\begin{aligned} lb_2^j = d_{f^\star}(q') &\leq d_{f'}(q') \\ &= \max_{i=1,\ldots,m} \{wl_{f'}(q', p_i)\} \\ &= \max_{i=1,\ldots,m} \{\sum_{k=1}^{j} wl_f(q_k, p_i)\} \\ &\leq \max_{i=1,\ldots,m} \{\sum_{k=1}^{j} d_f(q_k)\} \\ &= \max_{i=1,\ldots,m} \{c_j(f)\} \\ &= c_j(f) \leq c_j(\kappa) \ . \end{aligned}$$

In addition to the local lower bounds for any j, that can be used in Alg. 3.2, we also get a global lower bound $c(f) = c_s(f) \geq lb_2 := lb_2^s = c(f^\star)$.

3.6 Shortest-Path Based Local Search and Reset Times

In order to solve our main problem, Problem 1.1, we take the optimal solution of the uniform case, substitute the actual task-lengths and perform local search on it. We used the idea of multi-exchange neighborhoods presented by Frangioni et al. in [32] and incorporated the existence of multiple phases and reset times into it. Specifically, we do the following:

Let a feasible assignment f be given. We call a bucket (q, p) *loaded* if the workload of bucket (q, p) is equal to the duration of phase q. Let $r_f(t, q, p) \geq 0$ denote the reset time needed, if task t is added to bucket (q, p). We create an improvement graph by introducing one vertex for every task and one vertex for each bucket (see Fig. 3.6). We connect a task t_1 to a bucket (q, p) with a directed edge if t_1 can be added to bucket (q, p) without making it loaded, i.e. if

$$wl_f(q,p) + \ell(t_1) + r_f(t_1, q, p) < d_f(q) \ .$$

Let t_2 be a task currently assigned to a bucket (q, p). We connect a task t_1 to the task t_2 with a directed edge if it is possible to remove t_2 from bucket (q, p) and add t_1 instead, so that the bucket (q, p) is not loaded after this exchange, i.e. if

$$wl_f(q,p) + \ell(t_1) - \ell(t_2) + r_f(t_1, q, p) < d_f(q) \ .$$

By the *bucket of a vertex v*, we mean the bucket it represents or the bucket where the task it represents is assigned to. We call a path or a cycle in this graph *disjoint* if the buckets of its vertices are all different (see Fig. 3.6). Also, in case of a path, it must end with a bucket vertex. Now every disjoint path or cycle in this graph that includes at least one vertex whose bucket is loaded can be used to reduce the total number of loaded buckets and thus, most probably, also reduce the makespan: simply by performing the changes that every edge on that path or cycle represents.

In our specific application, reset times are caused by toolbit exchanges. We have that $r_f(t, q, p) > 0$ if a new toolbit exchange is needed to perform task t in bucket (q, p). For the details about the number of necessary toolbit exchanges, we refer to the ILP model presented in Section 2.2. When updating along a disjoint path or cycle that contains toolbit exchanges, we need to add the occurring toolbit exchanges to the corresponding buckets. Adding this feature improved our results considerably in some cases, see Sect. 4.

Finding optimal disjoint paths and cycles, i.e. ones that result in the best improvement, is $\mathcal{NP}$-hard, as shown in [32]. In order to find disjoint paths

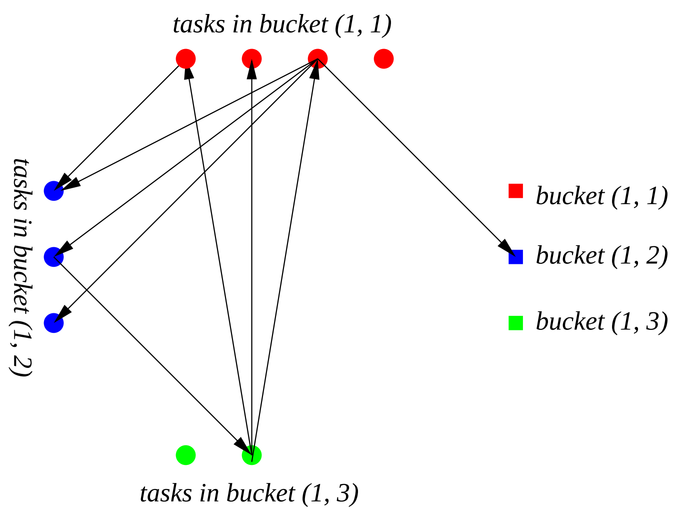

Figure 3.6: A simple example of an improvement graph with 3 buckets and 9 tasks. A path or cycle is disjoint if the colors of its vertices are all different.

and cycles we chose to implement the 1-SPT heuristic described in [32], adapted to our case, using a priority queue. In this heuristic, we set the following costs on the edges and look for shortest disjoint paths and cycles with respect to these costs. For an edge that connects a task t_1 to a task t_2, such that t_2 is assigned to a *loaded bucket* (q, p), the cost is

$$c_{t_1 t_2} = \ell(t_1) - \ell(t_2) + r_f(t_1, q, p) \ .$$

Otherwise the cost is 0. The skeleton of the algorithm is similar to Dijkstra's standard shortest path algorithm with the difference that the considered subpaths have to be checked for disjointness throughout the algorithm. If a subpath is not disjoint, a disjoint cycle might be found within it. For details we refer to [32].

We chose not to build the improvement graph explicitly since by sorting the tasks in each bucket in descending order according to their lengths, it is possible to decide about the existence of edges in constant time. This eliminates the need for an expensive update. We start the search once from every vertex and then repeat this procedure until no further improvement can be found. In the next chapter, we will see that this local search heuristic worked very well in practice.

Chapter 4

Computational Results

4.1 Test Instances and Environment

In this study, we used 20 widely different test jobs. Each job represents a PC board type with a standard machine setup for Assembléon's AX-5 from which we obtained an instance of our problem. All test jobs stem from customers of Assembléon and have been kindly provided by Assembléon to us. Table 4.1 shows some of the characteristics of these jobs. The number of tasks varies between 190 and 940 tasks, the number of placement phases s between 3 and 7. Since the AX-5 has 20 parallel robots (stations), the number of buckets varies between 60 and 140.

All computations are executed on a standard Intel P4 processor with 3.2 GHz and 4 GB main memory running under Suse Linux 9.2. Our algorithms are implemented in Java, we used JDK 5.0.

4.2 The Uniform Case

Table 4.1 displays the results for our instances under unit-length assumption (Prob. 3.2). Columns labeled lb_1 and lb_2 give the values obtained for the two lower bounds introduced in Section 3.5, respectively. It turns out that lb_1 is a rather weak bound, but lower bound lb_2 is much closer to the optimum.

Algorithm 3.1 (MinMax) performs surprisingly well as a heuristic for the uniform case, i.e. Problem 3.1. Its running time is negligible (and therefore not reported). It actually hits the optimal value quite often as can be seen by comparing the columns labeled "MinMax" and "opt". Intuitively, the reason might rely on the following observation: if the capacity of a phase can not be decreased in the algorithm, it will also be impossible to decrease

job	n	s	b	lb_1	lb_2	MinMax	opt(100)	opt	time(100)	time
j1p1	304	4	80	16	18	18	18	18	0.00	0.00
j1p2	190	3	60	10	12	16	16	16	0.08	0.08
j1p3	138	4	80	7	9	11	11	11	0.11	0.10
j2p1	940	5	100	47	284	284	284	284	0.00	0.00
j2p2	804	5	100	41	68	79	78	78	1.20	7.77
j2p3	792	5	100	40	74	83	83	81	1.12	5.81
j2p4	786	5	100	40	73	82	82	81	1.11	5.14
j2p5	750	5	100	38	98	117	117	109	1.20	89.19
j2p6	634	5	100	32	120	120	120	120	0.00	0.00
j2p7	532	5	100	27	46	59	59	59	0.55	7.82
j3p1	912	5	100	46	148	148	148	148	0.00	0.00
j3p2	660	5	100	33	120	120	120	120	0.00	0.00
j3p3	376	6	120	19	44	64	64	64	0.33	2.52
j4p1	312	7	140	16	18	18	18	18	0.00	0.00
j4p2	300	7	140	15	17	17	17	17	0.00	0.00
j4p3	288	7	140	15	16	16	16	16	0.00	0.00
j4p4	288	7	140	15	18	20	20	20	0.46	0.80
j4p5	212	5	100	11	24	24	24	24	0.00	0.00
j5p1	362	6	120	19	26	42	42	42	0.58	0.98
j5p2	345	6	120	18	23	46	46	46	0.63	1.10

Table 4.1: Results for the uniform case. The first four columns contain job characteristics (b is the number of buckets). Columns labeled with lb_1, lb_2 give lower bound values, "MinMax" reports the outcome of Algorithm 3.1, "opt" give the value of optimal solution (computed by Algorithm 3.2), "opt(100)" denotes the solution value, if Algorithm 3.2 is stopped after 100 recursive function calls. The two last columns report the CPU time in seconds for the "opt(100)" and "opt" versions, respectively.

it later on. That is, the only way to arrive at a better sum would be to *increase* the capacities of some phases and see if one can in return, decrease the capacities of other phases more than that. And this seems not to be much too likely.

Except for one extremely hard case (j2p5), the running time of our exact algorithm, Alg. 3.2, is below 8 seconds. In the exceptional case, the absolute difference of the result provided by the MinMax-heuristic is quite large. This results in a runtime of about 89 seconds.

Usually, Algorithm 3.2 requires most of its time to prove optimality. Since this algorithm is used as a starting solution for the local search, we also experimented with a "fast version" called opt(100) which terminates after 100 recursive function calls (and so terminates after about one second). In all but three cases this heuristic already found the optimum solution.

job	make-span	LS impact	#TE	cplex 60s	cplex 300s	cplex lb	our gap	gap cplex60	gap cplex300
j1p1	9.00	11.24%	0	-	8.62	7.87	14.43%	-	9.56%
j1p2	8.12	3.24%	0	8.64	8.47	8.12	0.00%	6.29%	4.22%
j1p3	5.64	4.89%	0	5.6	5.51	5.27	7.00%	6.18%	4.47%
j2p1	85.83	44.38%	8	82.55	82.46	80.84	6.18%	2.12%	2.00%
j2p2	38.33	5.17%	0	38.27	38.26	38.26	0.18%	0.03%	0.01%
j2p3	40.45	3.96%	0	40.19	40.19	40.18	0.67%	0.01%	0.01%
j2p4	40.37	3.96%	0	39.91	39.91	39.86	1.28%	0.12%	0.12%
j2p5	49.04	13.12%	6	44.36	44.36	44.36	10.56%	0.00%	0.00%
j2p6	39.74	36.36%	6	36.98	36.98	36.98	7.49%	0.00%	0.00%
j2p7	30.42	4.38%	0	30.42	30.42	30.42	0.00%	0.00%	0.00%
j3p1	82.98	0.00%	0	82.98	82.98	82.98	0.00%	0.00%	0.00%
j3p2	63.23	0.69%	0	63.23	63.23	63.23	0.00%	0.00%	0.00%
j3p3	47.19	2.42%	18	42.14	42.14	42.14	12.00%	0.00%	0.00%
j4p1	8.98	3.46%	0	-	9.57	8.13	10.53%	-	17.70%
j4p2	8.29	4.49%	0	-	8.38	7.72	7.37%	-	8.54%
j4p3	8.09	2.35%	0	-	8.58	7.48	8.21%	-	14.67%
j4p4	9.47	4.83%	0	11.26	10.83	8.99	5.24%	25.24%	20.40%
j4p5	11.42	3.73%	0	11.91	11.9	11.42	0.00%	4.28%	4.20%
j5p1	25.47	11.14%	10	23.33	23.33	23.29	9.35%	0.18%	0.18%
j5p2	24.81	8.94%	17	24.12	24.12	24.12	2.86%	0.00%	0.00%

Table 4.2: Results for the real-time scenario including toolbit exchanges. Comparison with CPLEX 9.1.

4.3 Impact of Local Search and Toolbit Exchanges

Table 4.2 shows our computational results for the scenario with real times and toolbit exchanges. The second column shows the makespan (cycle time) in seconds after running the local search procedure. The local search was started with the result of an optimal unit-length solution. The third column gives the percentage reduction obtained in comparison with this starting solution. Figure 4.1 shows the impact of the local search graphically.

We observe that local search helps a lot to reduce the cycle time if the addition of toolbit exchanges allows a better workload balancing. Remember that in the unit-length solution, we do not allow any toolbit exchanges and this feature is only added in the local search phase. Hence, in some cases, the solutions found during the local search were not even feasible beforehand and this explains some major improvements that were achieved for example in jobs j2p1 and j2p6.

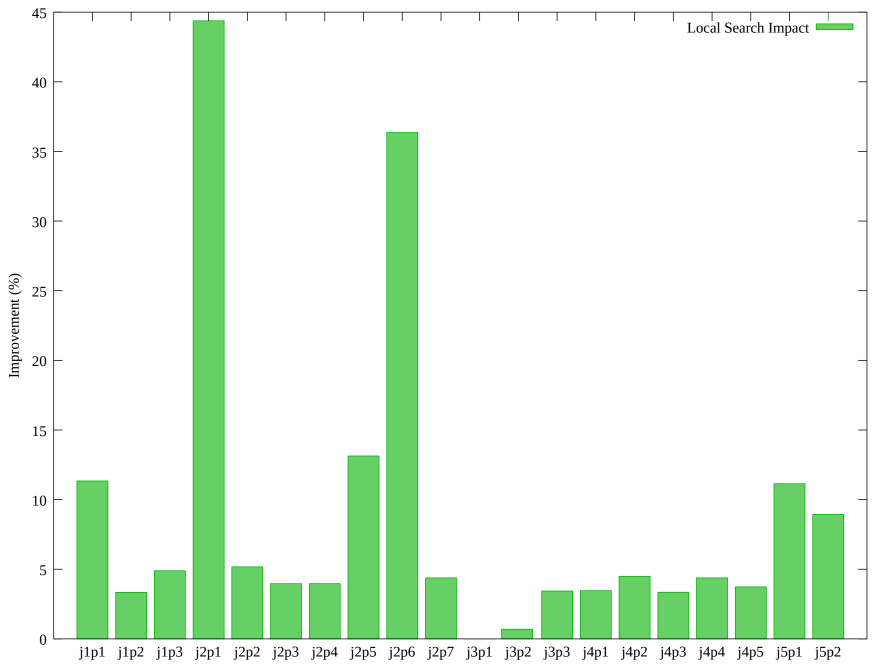

Figure 4.1: The impact of the local search in addition with toolbit exchanges.

4.4 Lower Bounds and Comparison with CPLEX

We compared the quality of the solutions obtained by our approach with a lower bound on the solution value (in several cases with the exact optimum). To this end, our problem including toolbit exchanges has been modeled as an integer linear program (ILP). Formulating the constraints for toolbit exchanges is a bit tricky and is presented in Section 2.2. It is mostly taken from [36].

To solve the ILP problems, we used ILOG CPLEX 9.1 with standard settings. In Table 4.2, we display the solutions values obtained by CPLEX after 60 and 300 seconds CPU time, the lower bound value obtained after 300s, as well as the optimality gap after 60 and 300 seconds, respectively. A graphical representation of this data is given in Figure 4.2.

The given time limits to CPLEX are relatively small, but in comparison to the running time of our method considerably longer. Recall that short time limits are justified by the fact that instances of this type have to be solved several hundreds of times for different setups. Thus we can afford in practice only a few seconds per instance.

The computational results are quite interesting. In 7 out of 20 cases, CPLEX managed to find even the optimal solution within 60 seconds. In

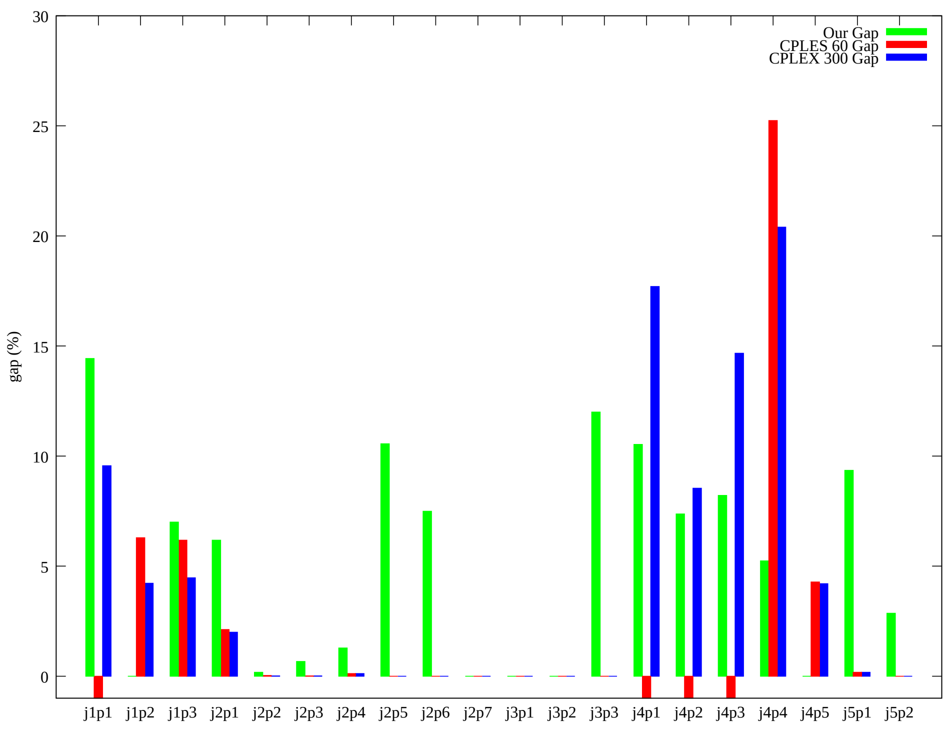

Figure 4.2: Maximal gap to the optimum (the lower these bars, the better). For each instance, the leftmost bar (green) shows our gap, the one in the middle (red) shows the gap of cplex after 60s of computation and the rightmost bar (blue) shows the cplex gap after 300s. When the bar goes downwards, it means that the solver could not find a solution for that instance in the given time limit.

contrast, CPLEX failed to find even some feasible solution in 4 cases in the same time limit. Within 300s, there was at least one feasible solution in all but one case. For one hard case (instance j1p1) we had to use a different CPLEX option (MIP Integer Feasibility) to find a feasible solution within 300s.

The final gap of our solution to the lower bound provided by CPLEX is relatively small, in 5 cases we provably found the optimum solution, and in 4 further cases (j4p1–j4p4) our gap is considerably smaller than the CPLEX gap after 300s. On the other hand, there are several instances which are seemingly easier to handle for CPLEX than for our approach. Thus there is no clear winner, but our approach is much faster and always guarantees at least a feasible solution after a few seconds.

4.5 An Experiment about the Unit-Length Assumption

We also ran an experiment to study how important it is to solve the unit-length case to optimality. In Table 4.3, we consider the jobs where the optimal unit-length solution was not found in version opt(100). In these cases, we observe that also the final gap is somewhat larger.

	opt (100)	initial makespan	makespan after LS	gap	opt	initial makespan	makespan after LS	gap
j2p3	83	42.97	40.75	1.40%	81	42.12	40.45	0.67%
j2p4	82	42.46	40.60	1.87%	81	42.03	40.37	1.28%
j2p5	117	60.92	51.10	15.19%	109	56.45	49.04	10.56%

Table 4.3: Impact of solving the unit-lengths problem to optimality for the solution quality after local search (LS). Columns 2-5 give values for our results started with the opt(100) solution, the remaining columns for the true optimum of the unit-length case.

Part II:
Towards a PTAS for the Planar Steiner Tree Problem

Chapter 5

The Steiner Minimum Tree Problem

The Steiner minimum tree (SMT) problem is one of the most fundamental and mostly studied problems in mathematics and computer science. Like the traveling salesman problem (TSP), it has been the testbed and motivation for many new algorithmic ideas and proof techniques. The SMT problem and its many variations are also of great importance in many practical applications such as all kinds of network design (e.g. VLSI design) and phylogenetic trees. For an extensive study on Steiner trees we refer to the excellent books of Prömel and Steger [71] and Hwang et al. [43].

In this part of the thesis, we will study the SMT problem in planar graphs. While this case is very interesting and important in its own right, it also encompasses some important geometric cases, such as the rectilinear and octilinear steiner tree problem *with obstacles*. All these variants are known to be $\mathcal{NP}$-hard [35] but it is not known weather there exists a polynomial time approximation scheme (PTAS) for them. We will introduce various approaches for designing PTASs for geometric problems and PTASs in planar graphs that have been devised during the past 15 years or so and discuss their applicability and limitations to our problem. We believe that one particular approach is very promising and we will thoroughly discuss how it *could* be applied to derive a PTAS for the SMT problem in planar graphs, provided that a conjecture of ours is true.

In this chapter, we will first introduce the two basic forms of this problem, namely the Steiner tree problem in graphs and the geometric Steiner tree problem. For each one, we will state some of the most important relevant results from the literature. Then we will move on to some more specific variants of the problem that are of special interest to this work: the problem in planar graphs and the rectilinear/octilinear problem with obstacles.

5.1 The SMT Problem in Graphs

Problem 5.1. *Given a connected undirected graph $G = (V, E)$ with edge weights $w_e \geq 0$ and a subset $R \subseteq V$ of terminal vertices, find a subtree of G that includes all the terminals and has minimum total edge weight.*

This is one of the 21 problems that were first shown to be $\mathcal{NP}$-hard in the landmark paper by Karp [45] in 1972. In 1989, Bern and Plassmann [10] showed that it is even $\mathcal{APX}$-complete, that is, it does not admit a PTAS unless $\mathcal{P} = \mathcal{NP}$. The best known bound for the non-approximability is $163/162 \approx 1.0062$ and is due to Thimm [81]. It is based on the assumption that $\mathcal{RP} \neq \mathcal{NP}$. An efficient and easy-to-implement approximation algorithm that achieves a performance ratio of $2 - \frac{2}{|R|}$ was given by Mehlhorn in [58] (a less efficient algorithm with the same approximation guarantee was suggested several times before in the literature [24, 70]). The best known approximation algorithm for this general problem is given by Robins and Zelikovsky [75] and its approximation guarantee is $1 + \frac{\ln 3}{2} + \epsilon \approx 1.55 + \epsilon$ for any given $\epsilon > 0$.

There are two well-known special cases of this problem that can be solved efficiently: when the terminal set equals the whole set of vertices, we have the minimum spanning tree problem and when the terminal set consists of only two vertices we have the shortest path problem.

5.2 The Geometric SMT Problem

Problem 5.2. *Given a set of points in the plane, find a shortest network that interconnects them.*

This problem is known as the *Euclidean Steiner Problem*. Another way to look at it is that we are given a finite set of points P and we are looking for a finite set of Steiner points Q in the plane, such that the minimum spanning tree of $P \cup Q$ has minimum total length. Garey, Graham and Johnson [34] showed its $\mathcal{NP}$-hardness in 1977. Note that it is not known weather the decision version of this problem is in $\mathcal{NP}$, since computing the location of the Steiner points could potentially involve computing many square roots. It is a highly celebrated result of Arora [3] and independently of Mitchell [60], first published in the years 1996/97, that this problem and some other geometric problems admit a PTAS. Later on, Rao and Smith [73] improved the performance of Arora's approach. We will discuss their approaches in detail in Chapter 7. One can also consider this problem in higher dimensions and indeed, Arora's result holds for any constant dimension d. However, one has to notice that both Arora's and Mitchell's algorithms involve very large constants in their running time, so that they are not (yet) practical to implement.

5.3 Uniform Orientation Metrics and Obstacles

An important variant of the geometric SMT problem is when only a finite set of directions are allowed to interconnect the given points. In VLSI design, until recently, only vertical and horizontal wires were used. This corresponds to the *rectilinear (or Manhattan) Steiner problem* where the distance between points is measured in the rectilinear, i.e. $\mathscr{L}_1$, metric. However, more complicated architectures such as the Y-architecture (with three directions at 120 degrees each) and the X-architecture or *octilinear metric* (with eight directions at 45 degrees each) are gaining more and more attractivity and importance due to their potentially high reduction of total wire length compared to the Manhattan architecture [20, 21, 51, 80]. More generally, one considers *uniform orientation metrics*, also called *λ-geometries*, where routing is allowed only along $\lambda \geq 2$ orientations forming consecutive angles of π/λ. The rectilinear case corresponds to the 2-geometry, the octilinear case to the 4-geometry.

The rectilinear Steiner tree problem was shown to be $\mathcal{NP}$-hard by Garey and Johnson [35] in 1977. In this case, the decision version is known to be in $\mathcal{NP}$. Only very recently (in 2005), Müller-Hannemann and Schulze [63] showed that the octilinear Steiner problem is also $\mathcal{NP}$-hard (and again the decision version is in $\mathcal{NP}$). It has not been proven yet — though widely believed — that the Steiner problem in λ-geometries is $\mathcal{NP}$-hard for general λ. Arora's approach [3] also works for λ-geometries, so all these cases admit PTASs. Hanan [40] showed in 1966 that the rectilinear problem can be reduced to the problem in graphs by using the so called *Hanan grid*: consider all vertical and horizontal lines going through the given terminals, create a vertex at every terminal and every intersection and let the line segments connecting adjacent vertices be edges. Hanan showed that a minimum Steiner tree in this graph corresponds to a minimum Steiner tree for the rectilinear problem. For the octilinear case, Müller-Hannemann and Schulze [63] first showed how it can be reduced to a graph problem of polynomial size that contains a $(1 + \epsilon)$-approximation of the minimum octilinear Steiner tree. The structure of solutions and exact approaches for the Steiner problem in uniform orientation metrics have been widely studied in the literature, for example in [13, 14, 15, 16, 42, 44, 50, 52, 66, 86].

Another complication is the introduction of *obstacles*, i.e. regions of the plane that are not allowed to be crossed by the Steiner tree. This problem is also of great practical relevance in VLSI design where certain areas of a board may not be crossed by wire. *It is not known weather the Steiner problem with obstacles admits a PTAS, not even in the rectilinear case.* It is however possible to reduce the rectilinear problem with obstacles to the graph problem using the Hanan grid [33, 40]. So, it can be approximated at least as good as the Steiner problem in graphs. For the case when the number of obstacles is a *constant*, Liu et al. [57] presented a PTAS for the rectilinear problem based on Mitchell's [60] approach. Provan [72] presented an FPTAS for the Euclidean Steiner problem with obstacles under the restriction that the terminals lie on a constant number of boundary polygons and interior points. The reduction of the octilinear Steiner problem given in [63] can also be used in the presence of obstacles and results again in a graph that contains a $(1+\epsilon)$-approximation of the optimal obstacle-avoiding octilinear Steiner tree.

Another interesting case occurs when we allow *soft obstacles*, i.e. obstacles that may be crossed by the tree but every connected component inside an obstacle may not exceed a given length restriction. This case is also very important in VLSI design where a large Steiner tree requires the insertion of buffers and inverters and these elements may not be placed on top of obstacles. Müller-Hannemann and Peyer [62] showed how to reduce this problem in the rectilinear case to the graph problem and obtain a $(1+\epsilon)\alpha$-approximation, where α is the best known approximation guarantee for the SMT in graphs. Very recently, Müller-Hannemann and Schulze [64] proved a similar result about octilinear Steiner trees with soft obstacles.

5.4 On Planar Graphs

Planar graphs constitute a very well-studied class of graphs in the literature, namely graphs that can be drawn on the plane without intersection of its edges other than at the vertices. Even though most hard combinatorial problems remain $\mathcal{NP}$-hard for their planar instances [35, 54], they are often "easier" to handle; for example, most of them admit a PTAS. Lipton and Tarjan [55] proved the original planar separator theorem in 1979 which they used in [56] to prove that the independent set problem admits a PTAS on planar graphs. Another milestone in this regard was Baker's paper [7] from 1994 where a general technique was introduced to obtain many PTAS results for planar graphs . The traveling salesman problem was first shown to admit a PTAS in 1995 by Grigni et al. [37], though only for unweighted graphs. Later on, Arora et al. [5] showed that there exists a PTAS for the TSP also in weighted planar graphs and recently, Klein [47] presented an improved algorithm that is linear in the number of vertices. In a follow-up paper, Klein [49] also showed that there exists a PTAS for subset TSP, the problem when the tour has to go only through a given set of terminals (as opposed to the entire graph). Another general approach for obtaining PTASs on planar graphs was introduced recently by Demaine and Hajiaghayi [26] in their theory of bidimensionality. Even the planar maximum satisfiability problem admits a PTAS [46]. We will discuss PTAS techniques on planar graphs in detail in Chapter 8.

However, *it is not known weather the Steiner minimum tree problem on planar graphs admits a PTAS or not, not even for the unweighted case.* While this problem is very important and highly interesting in its own right, such a result would also immediately yield a PTAS for the rectilinear Steiner problem with obstacles by using the Hanan grid [33, 40]: the Hanan grid is planar. Using the techniques of Müller-Hannemann et al. [63] one also obtains a PTAS for the octilinear Steiner problem with obstacles. In [84], Zachariasen presents a number of problems that can be solved on the Hanan grid. Some of them would also be shown to admit a PTAS if the planar SMT problem admits a PTAS. In Chapter 9, we will discuss how Klein's framework [47] could be applied to obtain a PTAS for the planar SMT problem. Specifically, we will state one missing lemma and show that if this lemma is proved true, then a PTAS will follow.

Chapter 6

On Spanners

Let $G = (V, E)$ be a graph with edge weights $w_e \geq 0$. For two vertices $u, v \in V$, let $d_G(u, v)$ denote the length of a shortest path between u and v according to the weights. A *spanner* of G is a subgraph G' that approximately preserves the distances between the vertices. More precisely, a t–spanner is a subgraph G' of G, such that $d_{G'}(u, v) \leq t \cdot d_G(u, v)$ (of course, we always have $d_G(u, v) \leq d_{G'}(u, v)$). The factor t is called the *stretch factor* of the spanner. The *size* of a spanner $s(G')$ is its number of edges, its weight $w(G')$ is its total weight. One is in particular interested in *sparse spanners*, i.e. spanners with a small size, and in *weight sparse spanners*, i.e. spanners with a small weight. Since the sparsest possible spanner – both in size and weight – is the minimum spanning tree, it is of interest to compare the weight of a spanner to the weight of the minimum spanning tree (MST). In this regard, one defines the tree weight $tw(G', G) := w(G')/w(\text{MST}(G))$. The problem of finding sparse spanners with low tree weight has been of great interest in the literature of the past two decades.

Spanners were first introduced by Peleg and Ullman [69] in 1987, where they used spanners to synchronize asynchronous networks. Other applications include communication networks, distributed computing [67] and phylogenetic trees [8]. For a given stretch factor t, finding the minimum t–spanner in size or in weight is NP–hard. For unweighted graphs, this was first shown in [68] for $t = 2$ and in [18] for $t \geq 3$. For weighted graphs, Elkin and Peleg [29] showed its hardness and some approximation limits for any $t \geq 1$. Venkatesan et al. [82] showed the hardness of this problem for various restricted classes of graphs and the case of planar graphs was shown to be hard by Brandes and Handke in [12].

Nevertheless, in many applications, such as in the design of PTASs, it is sufficient to find a spanner of low tree weight. A simple and probably the most cited algorithm for finding sparse spanners in weighted graphs –

both in regard to size and weight – is the natural greedy algorithm given by Althöfer et al. in [2]. They show that their algorithm produces a solution with near-optimal asymptotic weight bounds. We will discuss their work in the first section of this chapter. Some improvements on the analysis were given in [19] and very recently, linear time algorithms were proposed [9, 76], for the case when $t \geq 3$ is an odd integer, that achieve the same weight bounds. For planar graphs, Klein [47] presented a linear time algorithm. He used this result to derive a linear time PTAS for the weighted planar traveling salesman problem. Previously, Arora et al. [5] had used Althöfer et al.'s original result [2] to find a PTAS for the weighted planar TSP but their algorithm had higher complexity (see Chapter 8). In his most recent work [49], Klein presented a *subset spanner for planar graphs* which he used to derive a PTAS for subset TSP. We will discuss this result in detail later on in this chapter.

Another important domain for spanners is Computational Geometry. Given a set of points in the plane, find a sparse graph that interconnects them such that the Euclidean distances are approximately preserved. Some triangulations like the Delaunay triangulation fulfill this requirement very well, see [23, 28, 53] and also the survey by Eppstein [30]. Geometric spanners are for example applied in robotics but also in the design of geometric PTASs [73, 85]. Rao and Smith [73] also introduced the notion of "banyans" which are a generalization of spanners and are particularly of interest with regard to the Steiner minimum tree problem. We will discuss it shortly at the end of this chapter.

We would like to note that this is by no means a complete coverage on the subject of spanners, since it has been studied very extensively in the recent years, but gives an overview of some of the most important results with regard to the subject of this thesis.

6.1 Sparse Spanners of Weighted Graphs

In this section, we take a closer look at the landmark paper of Althöfer et al. [2] from the year 1993. They presented a simple greedy algorithm to find spanners in weighted graphs that are sparse both in size and in weight (see Alg. 6.1).

Obviously this algorithm can be implemented in polynomial time and it is fairly easy to see that it returns indeed an r–spanner: take a shortest path between two vertices u and v in G and look at every edge of it; if any edge e is not included in G', then there is a path in G' connecting its endpoints with a total weight $\leq r \cdot w_e$; by replacing e with that path, we get a path between u and v in G' whose total weight is $\leq r \cdot d_G(u, v)$.

Algorithm 6.1: Greedy Spanner [2]

```
Input  : A graph $G = (V, E)$ and a stretch factor $r$.
Output: An $r$–spanner $G'$ of $G$.
begin
    Sort $E$ by nondecreasing weight;
    $G' = (V, \{\})$;
    for every edge $e = (u, v) \in E$ do
        if $r \cdot w_e < d_{G'}(u, v)$ then
            add $e$ to $G'$;
    return $G'$;
end
```

What is more interesting are the size and weight bounds given in Theorem 6.1 for general graphs and in Theorem 6.2 for planar graphs below (recall that the tree weight $tw(G')$ was defined as $w(G')/w(\mathrm{MST}(G))$):

Theorem 6.1 ([2]). *Given an n–vertex graph G and a $t > 0$, there is a polynomially constructible $(1 + t)$–spanner G' of G such that,*

(i) $s(G') < n\lceil n^{2/t}\rceil$,

(ii) $tw(G') < 1 + n/t$.

Theorem 6.2 ([2]). *Given an n–vertex* planar *graph G and a $t > 0$, there is a polynomially constructible $(1 + t)$–spanner G' of G such that,*

(i) $s(G') < (n - 1)(1 + 2/t)$,

(ii) $tw(G') < 1 + 2/t$.

In proving the size bounds, they use a result from extremal graph theory that says that an n–vertex graph with girth $> r$ has size $< n\lceil n^{2/(r-2)}\rceil$. In the case of planar graphs, they show that the bound on the size is $\leq (n - 2)(1 + 2/(r - 2))$ for a girth $> r$. Recall that the girth of a graph is the length of it smallest circle. Then they show that the output produced by Alg. 6.1 has girth $> r + 1$ and the size bounds follow.

To prove the weight bounds, they first show that the MST of G is contained in G'. They first prove Theorem 6.2, i.e. the case of planar graphs. They consider an embedding of the graph and draw a skinny polygon of perimeter $2 \cdot w(\mathrm{MST}(G))$ around the minimum spanning tree. Then they "grow" this polygon, absorbing adjacent edges one-by-one until the polygon becomes the outer face of the graph. As doing so, they charge the weight of every added edge e to the cycle it creates with the current boundary of the polygon. They show that the weight of this cycle without e is $> (1+t) \cdot w(e)$ and thus, the perimeter of the polygon will decrease by at least $t \cdot w(e)$. Using this idea, the weight bound can be deduced.

To show the weight bound for general graphs in Theorem 6.1, they consider for each vertex v of G, the MST of G together with all edges adjacent to v. They argue that this graph is planar and use the result of Theorem 6.2 to get a weight bound on it. Then they add up these terms to get the desired weight bound of Theorem 6.1.

Further in the paper, the two following lower bounds are also shown:

Theorem 6.3 ([2]). *For every even integer $t \geq 0$ and every $n \geq 3$, there exists an n–vertex graph G for which every $(1+t)$–spanner G' is such that*

(i) $s(G') > \frac{1}{8}n^{1+4/3(t+3)}$,

(ii) $tw(G') > \frac{1}{8}n^{4/3(t+3)}$.

Theorem 6.4 ([2]). *For every even integer $t \geq 2$, there are infinitely many values of n, for which there exist n–vertex planar graphs G with unit edge weights, such that every $(1+t)$–spanner G' satisfies*

(i) $s(G') = \Omega(n(1+2/t))$,

(ii) $tw(G') = \Omega(1+2/t)$.

We see that the bounds for planar graphs are tight. Even for the general case, the size bound is optimal within a constant factor in the exponent of n. But the weight bound could be improved and was indeed improved in [19] through a new analysis of Alg. 6.1 to $O(n^{\frac{2+\epsilon}{t}})$ for any $\epsilon > 0$. This bound is now optimal within a constant factor in the exponent of n and even more, any improvement in the exponent of n would imply a better bound for an open extremal graph theory problem.

Further in the paper, Althöfer et al. present a lower bound on so-called Steiner spanners and also discuss spanners in Euclidean graphs of dimension 2 and higher. We refer the interested reader to their original paper [2].

6.2 More on Planar Graphs

We already saw in the previous section that planar graphs have nicer properties than general graphs with respect to spanners. Here we want to shortly present Klein's algorithm [47] to produce a spanner for planar graphs with the same properties as in Theorem 6.2 but that can be implemented in *linear* *time.* But in order to do so and also since this will be needed throughout this work, we will first give a combinatorial definition of embedded planar graphs [61] and state some elementary results about them. This section is mostly taken from [47].

Before we start, let us shortly review the geometric definition of planar graphs. A planar graph is a graph that can be drawn on the Euclidean plane in a way that its edges do not intersect other than at their endpoints. A planar graph drawn on the plane is called a planar embedded graph or plane graph. A plane graph divides the plane into one or more regions. These regions are called faces. There is always an infinite face, the one “outside” the graph. A plane graph satisfies Euler’s formula: $n-m+\phi=\kappa+1$, where n is the number of vertices, m is the number of edges, ϕ is the number of faces and κ is the number of connected components. The dual of a plane graph G is a plane graph $G^\star$ that has one vertex inside every face of G. For every edge e of G there is a corresponding edge in $G^\star$: it connects the vertices of $G^\star$ that are inside those faces of G that meet at e (note that this might create loops in $G^\star$). It must be drawn in a way that it crosses e exactly once and does not cross any other edge of G or $G^\star$. Note that according to this definition, $G^\star$ is always connected since the infinite face is adjacent to all connected components of G. The number of edges of G and $G^\star$ is the same, the number of vertices of $G^\star$ is equal to the number of faces of G. For connected plane graphs, we also have that the number of faces of $G^\star$ equals the number of vertices of G and that $G^{\star\star}=G$.

Now let us turn to the combinatorial definition. *The combinatorial definition will be the one we will primarily use throughout this work.* As noted below, it deviates slightly from the geometric definition when it comes to the dual graph of disconnected plane graphs.

Let E be any given finite set. We can interprete E as a set of *edges* and we define $E\times\{\pm1\}$ to be the corresponding set of *darts*. The darts of e, namely $(e,1)$ and $(e,-1)$ represent the two opposite orientations of e. We define an *embedded graph* on E to be a pair $G=(\pi,E)$ where π is a permutation on the darts of E. A *vertex* is defined in terms of π: every permutation cycle $(d_1,\dots,d_k)$ of π is defined to be a vertex. For a vertex v, we denote the set $\{d_1,\dots,d_k\}$ by $D(v)$. The set $D(v)$ corresponds to the set of outgoing darts of v. We use $E(G)$ for the set of edges and $V(G)$ for the set of vertices. Let $\mathrm{rev}(\cdot)$ be the function that reverses darts, i.e. $\mathrm{rev}((e,i))=(e,-i)$. For a dart d of G, let $\mathrm{tail}(d)$ be the vertex, i.e. the permutation cycle of π, containing d. We define $\mathrm{head}(d)=\mathrm{tail}(\mathrm{rev}(d))$. For an edge e of G, let $\mathrm{ends}(e)=\{\mathrm{head}((e,1)),\mathrm{tail}((e,1))\}$. One can think of π as the function specifying for each dart, the next dart in clockwise order around its tail.

A *walk* of darts in G is a sequence $d_1,\dots,d_k$ of darts such that head(d_{i-1}) = tail(d_i), for $i = 2, \dots, k$. It is a *closed* walk if in addition head(d_k) = tail(d_1). It is a *simple* path/cycle if no vertex occurs twice as the head of a dart. A walk contains an edge if it contains a dart of it and it contains a vertex if the vertex occurs as the head or tail of some dart of it.

To define the *faces* of the graph we look at the *dual* of the graph. Define $\pi^\star = \pi \circ \text{rev}$ and $G^\star = (\pi^\star, E)$. $G^\star$ is said to be the dual of the graph G. We define the faces of G to be the vertices of $G^\star$. That is, the faces of G are the permutation cycles of $\pi^\star$. A face can be interpreted as a closed walk in G. Note that the faces of $G^\star$ are the vertices of G and that G and $G^\star$ are defined on the same set of edges. Since rev $\circ$ rev is identity, we get

Proposition 6.5. $G^{\star\star} = G$.

We say that an embedding π of a graph G is planar if it satisfies Euler's formula: $n - m + \phi = 2\kappa$, where n is the number of vertices, m is the number of edges, ϕ is the number of faces and κ is the number of connected components. In this case, we say that $G = (\pi, E)$ is a *planar embedded graph* or simply a *plane graph*. It can be shown that the dual of a connected embedded graph is connected. Hence, the connected components of $G^\star$ correspond one-to-one to the connected components of G and if G satisfies Euler's formula, so does $G^\star$. So, if G is a plane graph, so is $G^\star$. However, notice that for disconnected graphs, this definition of dual diverges from the geometric definition in that it assigns multiple vertices to the infinite face, one for every connected component. According to the geometric definition, the dual of a plane graph is always connected. But using that definition would mean giving up some nice properties like $G^{\star\star} = G$.

Now let us state some properties of plane graphs. Let T be a spanning tree in G. An edge $e \notin T$ builds a unique cycle with T if we add it to T. This cycle is called the *elementary cycle* of e with respect to T in G. Let $T^\star$ be the set of edges of G that are not in T. We refer to $T^\star$ as the tree dual to T, since we have

Proposition 6.6. *$T^\star$ is a spanning tree of $G^\star$.*

For a set $S \subset V$, we use $\Gamma(S)$ to denote the set of edges having one endpoint in S and one endpoint in $V \setminus S$. Such a set is called a *cut* in G. Note that $\Gamma(S) = \Gamma(V \setminus S)$ and that it separates any pair of vertices u, v with $u \in S, v \notin S$. If S is connected in G and $V \setminus S$ is connected in G, we call $\Gamma(S)$ a *simple cut*. The following results can be shown with little effort.

Proposition 6.7. *The edges of a simple cut in G form a simple cycle in $G^\star$ and vice versa.*

Proposition 6.8. *Let G be a connected plane graph, let T be a rooted spanning tree of G, let v be a non-root vertex of G, and let e be the parent edge* *of v. Then the elementary cycle of e in $G^\star$ with respect to $T^\star$ consists of the edges of Γ(descendents of v in T).*

Proposition 6.9. *An edge e is a self-loop of G iff it is a cut-edge of $G^\star$.*

Finally, we discuss two ways of removing edges from an embedded graph: deletion and compression. Both of these preserve planarity. *Deletion* is just the usual deletion of an edge, formally resulting in $E \setminus \{e\}$ as the new set of edges and a slight modification in π. For a set W of edges, we denote by $G - W$, the graph resulting from deleting the edges in W. The order of deletion does not affect the resulting graph. *Compression* is deletion in the dual. Compressing an edge e results in the graph $(G^\star - \{e\})^\star$ and we denote it by G/e. If e is not a self-loop then the effect of compressing e is the same as the effect of contracting e in the usual sense. However, if e is a self-loop, and so a cut-edge in $G^\star$, then the effect of compressing e is to duplicate its tail v and get two connected components: the part of the graph that was inside e and the part that was outside.

Now that we went through some basics about planar graphs, we state Klein's linear time algorithm to find a spanner in a planar graph [47]. It is summarized in Alg. 6.2. It produces a spanner with the same properties as the one by Althöfer et al. [2] but in linear time. Its proof of correctness uses similar ideas as the proofs of Althöfer et al. [2] together with some of the propositions above. Note that a minimum spanning tree of a planar graph can be found in linear time using the algorithm of Cheriton and Tarjan [22].

Algorithm 6.2: Klein's Planar Spanner [47]

```
Input  : A graph $G = (V, E)$ and an $\epsilon > 0$.
Output: A $(1+\epsilon)$–spanner $G'$ of $G$ with tree weight $< (1 + 2/\epsilon)$.
begin
    find a minimum spanning tree $T$ of $G$;
    let $T^\star$ be the dual tree, rooted arbitrarily;
    for each edge $e$ of $T^\star$, in order from leaves to root do
        let $f$ be the face of $G$ whose parent edge in $T^\star$ is $e$;
        let $e_1, \ldots, e_s$ be the child edges of $e$ in $T^\star$;
        $x_{\text{omit}} = w(D(f) \cap D(T)) + \sum_{i=1}^{s} x[e_i]$;
        if $x_{omit} \leq (1+\epsilon)w(e)$ then
            $x[e] = x_{\text{omit}}$;
        else $x[e] = w(e)$
    let $G'$ be the union of $T$ and the edges $\{e \in T^\star : x[e] = w(e)\}$;
    return $G'$;
end
```

6.3 Mehlhorn's Graph is Planar for Planar Graphs

In 1998, Arora et al. made the following conjecture in [5]:

Conjecture 6.10. *There exists a function $f(\cdot)$ such that: given $\epsilon > 0$, a weighted planar graph G, and a subset S of vertices, there exists an edge-induced subgraph G' which $(1+\epsilon)$–approximates all internode distances in S, and furthermore G' has total edge weight at most $f(\epsilon)$ times the minimum Steiner tree weight of S.*

The correctness of this conjecture would result in a PTAS for subset TSP. Also, we thought that this result might help in finding a PTAS for the planar Steiner tree problem. In May 2006, the proof of this conjecture was published by Klein [49]. We will discuss his proof in the next section. But before we knew of Klein's result, we attempted to prove it using Mehlhorn's graph [58]. Our proof did not succeed but we got a nice byproduct that we are going to present in this section.

A well known algorithm [24, 70] to get a 2–approximation for the SMT problem in graphs is to find the minimum spanning tree in the *complete distance graph* $G_D = (R, E_D)$ for an instance $G = (V, E)$ of the SMT problem with terminal set R. The graph G_D is the complete graph on the set of terminals and the weight of an edge equals to the weight of the shortest path between its endpoints in G. By computing an MST in G_D, replacing every edge by the corresponding shortest path in G, eliminating cycles and repeatedly deleting non-terminal leaves, one gets a tree in G whose weight is at most twice as much as the SMT of G. In [58], Mehlhorn observed that it is not necessary to consider the *complete* distance graph. Instead, it is sufficient to find the SMT of a sparser graph G_D^M, that we call *Mehlhorn's graph*, obtained as follows: we divide the input graph G into *Voronoi regions* around the terminals. The Voronoi region of a terminal v is the set of vertices whose closest terminal (in terms of the shortest path) is v. If a vertex happens to be at the same distance to two or more terminals, it should belong to the Voronoi region of the terminal with the smallest index. This gives us a complete partition of the graph G. The graph $G_D^M = (R, E_D^M)$ has an edge between two terminals u and v iff there exists an edge $\{x, y\}$ in G such that x belongs to the Voronoi region of u and y belongs to the Voronoi region of v. The weight of such an edge is the minimum of the weight of all such paths connecting u and v. Mehlhorn showed that every MST of G_D^M is also a MST of G_D and thus it is sufficient to consider G_D^M.

Using Dijkstra's algorithm [27] with a super-source connected to all terminals with weight 0, one can find the Voronoi regions of the input graph in time $O(n \log n + m)$. Mehlhorn argued that replacing the edges of a MST of G_D^M with the corresponding shortest paths of G will result in a tree and hence it is not necessary to look for cycles and non-terminal leaves. Thus, all the remaining parts of the algorithm can be implemented in linear time and the algorithm has a total runtime of $O(n \log n + m)$.

We proved the following proposition:

Proposition 6.11. *If the input graph G is planar, then Mehlhorn's graph G_D^M is also planar.*

Proof. We show explicitly how to draw the graph G_D^M in a planar way. To this end, consider a planar drawing of G on the plane. If we run Dijkstra's algorithm as described above from a super-source connected to all terminals, look at the resulting shortest path tree and remove the super-source from it, we get a shortest path tree around each terminal, corresponding to the Voronoi region of that terminal. These trees are disjoint and moreover, they do not intersect on the plane either. Hence, it is possible to draw a very thin polygon around each one of these trees in a way that these polygons do not intersect. Now consider one such polygon. Literally erase whatever is in its interior except for its single terminal vertex v. Consider all the edges of the graph that intersect with this polygon (these are edges that connect this Voronoi region to another one). We call the intersection point of such an edge with the boundary of the polygon a *portal*. Draw a curve from v to each one of these portals in a way that these curves do not intersect. This is possible by considering the portals in clockwise order around the boundary. Now perform this procedure on every polygon. Notice that the only edges of the original graph that remain are the ones interconnecting one such polygon with another. These edges do not intersect themselves, since they were part of the original planar drawing of G. By drawing the curves inside the polygons from the portals to the terminals, we have *extended* each such edge to connect the terminals of the Voronoi regions of its endpoints. If we now erase the polygons from our drawing, what remains is a graph on the set of terminals that has the Mehlhorn graph as its subgraph. This drawing is planar – and so is Mehlhorn's graph. □

In [41], Henzinger et al. show that it is possible to solve the single-source shortest path problem in planar graphs in linear time. Their algorithm assigns distance labels to every vertex and relaxes edges by a given scheme, so that in the end, the distance labels correspond to the lengths of the shortest paths from the source to every vertex and the actual shortest paths can then be calculated. Initially, the label of every vertex is infinity, except for the source that has label 0. One can modify their algorithm to calculate the Voronoi regions for a given set of terminals, simply by initializing the label of all terminals with 0. Intuitively, we believe that this modification can not possibly increase the runtime of their algorithm — in fact, shortest paths should be found faster when there are more sources and we only want to find the distance to the closest source. However, the analysis of the runtime is very complicated and involves a charging scheme that does not immediately adapt to this variation. A formal proof that the runtime will remain linear after this modification was beyond the scope and intent of this work. However, if this is true, then *we get that Mehlhorn's algorithm can be implemented in linear time on planar graphs.* This would be the first linear-time algorithm to derive a 2-approximation of the SMT problem on planar graphs.

Our original idea was to find spanners with constant tree weight using Mehlhorn's graph together with the greedy algorithm of Althöfer et al. [2] presented in the first section of this chapter. But exactly since Mehlhorn's graph does not contain all edges of the complete distance graph, the resulting "spanner" will not preserve approximate distances between all pairs of the terminals and thus, our attempt failed.

6.4 A Subset Spanner for Planar Graphs

Klein positively settled the conjecture of Arora et al. [5] (Conjecture 6.10) by proving the following theorem in [49]:

Theorem 6.12 ([49]). *There is an algorithm that given $\epsilon > 0$, a planar graph $G = (V, E)$ with edge-weights, and a set of terminals $R \subseteq V$, selects an edge-induced subgraph G' with the following properties:*

(i) G' has weight $O(\epsilon^{-4})$ times the weight of the Steiner minimum tree of R in G,

(ii) for every pair of terminals $u, v \in R$, we have $d_{G'}(u, v) \leq (1 + \epsilon)d_G(u, v)$.

The algorithm takes $O(\epsilon^{-1} \cdot n \log n)$ time.

The idea of the proof is to show how to construct a number of spanners that become gradually more and more complicated, each one based on the previous one, until in the end we get the desired subset spanner. First, a basic charging scheme is proven that is used to show the weight bound. Then we introduce a single-source spanner, a bipartite spanner, a boundary-to-boundary spanner and finally a tree spanner. We use the tree spanner on Mehlhorn's approximation to get our final result. Now we are going to shortly discuss each of the steps of this algorithm.

Basic Charging Technique

The charging technique is used to prove the weight bound on the spanner. The basic idea of the charging technique is similar in its flavor to that of Althöfer et al. [2] in the proof of Theorem 6.2 but somewhat more complicated. We start with some face f_0 and sequentially add paths $P_1, \ldots, P_k$ to it. Each path P_i splits the face f_{i-1} into two faces f_i and g_i. The weight of a path P_i is at most $(1+\epsilon)$ times the weight of that part of f_{i-1} that it "cuts out", so it can be "charged" to that part of it. Combining these inequalities leads to a bound on the total weight of the P_i's, namely a constant factor of the weight of the original face f_0. Formally, Klein proves the following lemma:

Lemma 6.13 ([49]). *Let G be a planar embedded graph with nonnegative edge-weights $\ell(\cdot)$. Let H_0 be a subgraph of G, let f_0 be a face of H_0, and let $P_1, \ldots, P_k$ be a set of edge-disjoint paths in G that are edge-disjoint from H_0. For $i = 1, 2, \ldots, k$, let H_i denote the subgraph of G formed by $H_0, P_1, P_2, \ldots, P_i$. We assume that, for $i = 1, \ldots, k$,*

- *the path P_i shares only its endpoints with H_{i-1},*
- *the edges of P_i are strictly enclosed by a face f_{i-1} of H_{i-1}, splitting f_{i-1} into two faces, g_i and f_i, in H_i, where*

$$g_i = f_i[start(P_i), end(P_i)] \circ rev(P_i)$$

  and

$$f_i = P_i \circ f_{i-1}[end(P_i), start(P_i)] \ ,$$

- $(1+\epsilon)\ell(P_i) < \ell(f_{i-1}[start(P_i), end(P_i)])$.

Then we have

$$\ell(P_1 \cup \cdots \cup P_k) < \epsilon^{-1}\ell(f_0) \ .$$

A Single-Source Spanner

As a first step, we show how to construct a certain single-source spanner, a spanner that preserves approximate distances from a given source to a certain set of vertices. More precisely we have the following theorem:

Theorem 6.14 ([49]). *Let G be a plane graph with nonnegative edge-weights $\ell(\cdot)$, let Q be a shortest path in G between its endpoints, and let R be a shortest path from some node x to Q. Then for any $\epsilon > 0$ there is a subgraph H of G of weight $< 4(\epsilon^{-1} + \epsilon^{-2})\ell(R)$ such that, for each node y of Q,*

$$d_{H \cup Q \cup R}(x, y) \leq (1+\epsilon) d_G(x, y)$$

and H consists of $O(\epsilon^{-1})$ shortest paths from x to some node of Q.

Algorithm 6.3: Single-Source Spanner [49]

```
Input/Output: See Theorem 6.14.
begin
    $H := \emptyset, j := 0, k := 0$;
    for $i := 1, 2, \ldots$ do
        repeat
            $j := j + 1$;
            if there is no node $y_j$ or $\ell(Q[y_0, y_j]) > \epsilon^{-1}\ell(R)$ then exit;
        until $(1+\epsilon)d_G(x, y_j) < d_G(x, y_k) + \ell(Q[y_k, y_j])$ ;
        $P_i :=$ shortest $x$-to-$y_j$ path;
        add $P_i$ to $H$;
        $k := j$;
    return $H$;
end
```

Let y_0 be end(R), that is the node of Q closest to x and number the nodes of Q as $\ldots, y_{-2}, y_{-1}, y_0, y_1, y_2, \ldots$. The spanner H is comprised of shortest paths from x to some y_i and is found in two passes, one considering $y_1, y_2, \ldots$ and the other one considering $y_{-1}, y_{-2}, \ldots$. Here we discuss only the first pass, the second pass is symmetric. The algorithm is summarized in Alg. 6.3. The algorithm proceeds in a simple greedy fashion, where j is the main counter and for each y_j it decides on-the-spot to include the path $[x, y_j]$ in H or not. The variable k stores the last j for which the path was included and the variable i stores the number of paths that were included so far. The decision to include a path or not is simply based on the given condition in the theorem that $d_{H\cup Q\cup R}(x, y_j) = d_G(x, y_k) + \ell(Q[y_k, y_j])$ must be $\leq (1+\epsilon)d_G(x, y_j)$. The algorithm stops once all y_j are processed or when $\ell(Q[y_0, y_j]) > \epsilon^{-1}\ell(R)$. For the latter case, Klein argues that the path $R \circ Q[y_0, y_j]$ is an x-to-y_j path whose length is at most $1 + 3\epsilon$ times $d_G(x, y_j)$[1].

The weight bound is shown fairly easily using the basic charging technique of Lemma 6.13. The analysis of the number of paths involves considering a number of inequalities, deriving a recurrence and solving it.

A Bipartite Spanner

The bipartite spanner is defined on plane graphs whose boundary is comprised of two paths Q_1 and Q_2, where Q_2 is a shortest path. The spanner preserves approximate distances between the vertices of Q_1 and Q_2.

[1]In the original paper, this argument was forgotten and Philip Klein sent me the correction after I asked about it. Indeed, that the factor here is $1 + 3\epsilon$ instead of $1 + \epsilon$ results in a slight increase in the constants of this theorem and the algorithm overall.

Theorem 6.15 ([49]). *Let G be a plane graph with nonnegative edge-weights $\ell(\cdot)$ whose boundary is a cycle formed by two paths, Q_1 and Q_2, where Q_2 is a shortest path. Then for any $\epsilon > 0$ there is a subgraph H of weight at most $c\epsilon^{-3} \cdot \ell(Q_1)$ such that, for each node x in Q_1 and each node y in Q_2,*

$$d_{H \cup Q_2}(x, y) \leq (1 + O(\epsilon))d_G(x, y)$$

where c is a constant.

Algorithm 6.4: Bipartite Spanner [49]

```
Input/Output: See Theorem 6.15.
begin
    for each node $x$ of $Q_1$ do
        $F[x]$ := a shortest $x$-to-$Q_2$ path;
    let $x_0, \ldots, x_s$ be the nodes of $Q_1$ in left-to-right order;
    $S := \emptyset, i := 0$;
    for $j := 1$ to $s - 1$ do
        if $\ell(Q_1[x_i, x_j]) > \epsilon \cdot \ell(F[x_j])$ then
            $S := S \cup \{x_j\}$;
            $i := j$;
    let $H$ be $Q_1$;
    for each node $x$ of $S$ do
        apply Alg. 6.3 with $Q = Q_2$ and $R = F[x]$ to obtain a
        single-source spanner $H_x$;
        add $H_x$ to $H$;
    return $H$;
end
```

The algorithm to find a bipartite spanner is given in Alg. 6.4. First, for each vertex x of Q_1, we find a shortest path $F[x]$ from x to Q_2. Then we greedily find a subset S of the vertices of Q_1 that are not too far from each other. For each one of these vertices, we find a single-source spanner to Q_2 and we let H be the union of these single-source spanners with Q_1. The weight bound and the proof of correctness are shown with little effort by adding up and combining a number of inequalities. Klein also discusses how to implement Alg. 6.4 in time $O(\epsilon^{-1} n \log n)$ using a dynamic-tree data structure (like self-adjusting top trees [78]) and multiple-source shortest-paths algorithms for planar graphs [48].

A Boundary-to-Boundary Spanner

The boundary-to-boundary spanner preserves all internode approximate distances of the nodes on the boundary of a plane graph. Its weight is bounded by a constant factor times the weight of the boundary cycle.

Theorem 6.16 ([49]). *Let G be a plane graph with nonnegative edge-weights $\ell(\cdot)$. Let C be the boundary of G. Then for any $\epsilon > 0$ there is a subgraph $\hat{H}$ of weight $O(\epsilon^{-4} \cdot \ell(C))$ such that, for any nodes x and y in C,*

$$d_{\hat{H}}(x, y) \leq (1 + \epsilon) d_G(x, y) \ .$$

Algorithm 6.5: Boundary-to-Boundary Spanner [49]

```
Input/Output: See Theorem 6.16.
begin
    let $C$ be the boundary of $G$;
    let $S$ be the set of pairs of nodes $(x, y)$ with
        $(1 + \epsilon)d_G(x, y) < \ell(C[x, y])$;
    if $S$ is empty then return $C$;
    let $(\hat{x}, \hat{y}) \in S$ such that there is no pair $(x, y) \in S$ lying on $C[\hat{x}, \hat{y}]$;
    let $B$ be the path $C[x, y]$ and let $P$ be the shortest $x$-to-$y$ path;
    let $L$ be the subgraph of $G$ enclosed by the cycle $B \circ \mathrm{rev}(P)$;
    let $G'$ be the subgraph of $G$ enclosed by the cycle $P \circ C[y, x]$;
    using Alg. 6.4 obtain a bipartite spanner $H$ of $L$ between the
    nodes of $B$ and $P$;
    recursively find a boundary-to-boundary spanner $H'$ of $G'$;
    return $H \cup H'$;
end
```

The boundary-to-boundary spanner is built recursively. One finds a pair of vertices $\hat{x}, \hat{y}$ for which the spanner-property is not fulfilled and such that there is no other pair of such vertices between them. Then the graph is split into two parts using the shortest path between $\hat{x}$ and $\hat{y}$: one "below" the shortest path and one "above" it. For the part below, one can use the bipartite spanner of Theorem 6.15 and for the part above, one finds a boundary-to-boundary spanner recursively. This construction fits perfectly with the basic charging scheme of Lemma 6.13 and indeed, this lemma is used to prove the weight bound on the spanner. The approximate distance preserving property is shown by combining the paths in the various subgraphs of the recursion and arguing over a number of cases that there is indeed a path in the spanner that is at most $(1 + \epsilon)$ times as long as the original shortest path between any pair of vertices on the boundary. Klein also presents an implementation of Alg. 6.5 that runs in $O(n \log n)$ time.

A Spanner on the Nodes of a Tree

Theorem 6.17 ([49]). *Let G be a plane graph with nonnegative edge-weights $\ell(\cdot)$. Let T be a tree in G. For any $\epsilon > 0$, there is a subgraph*

H of G of weight $O(\epsilon^{-4}\ell(T))$ such that, for every pair of nodes x, y in T,

$$d_H(x,y) \leq (1+\epsilon)d_G(x,y) \ .$$

Consider the tree T and "blow it up" this way: consider the planar Euler tour of T, that is, a tour that visits every edge of the tree twice and every node v some number $n(v)$ times. Duplicate every edge and make $n(v)$ copies of each node such that the tour turns into a simple cycle C, in fact, into a face of the graph. We have $\ell(C) = 2\ell(T)$. Redraw the graph in a way that C becomes the boundary face of the graph. Use Algorithm 6.5 to derive a boundary-to-boundary spanner H on the redrawn graph. This spanner satisfies the required properties.

Proof of Main Theorem (Theorem 6.12)

Given an n-node planar graph G and set of terminals R, find a Steiner tree whose weight is at most twice as much as the SMT of R in G, in $O(n \log n)$ time, using Mehlhorn's algorithm [58](also see the previous section). Apply Theorem 6.17 on G and this tree to derive a spanner that approximately preserves the internode distances among the vertices of T and whose weight is $O(\epsilon^{-4})$ times the weight of the Steiner minimum tree of R in G. The whole algorithm takes time $O(\epsilon^{-1} n \log n)$.

6.5 On Banyans

In [73], Rao and Smith introduced a generalization of spanners called *banyans.* A t-banyan for a set of terminals R is a graph that contains a t-approximation of the Steiner minimum tree of all subsets of R (not just for subsets of size 2 as in the case of spanners). Rao and Smith considered this concept on complete geometric graphs and indeed proved that if ϵ and d are constants, it is possible to find a $(1+\epsilon)$-banyan of n points in d-dimensional space in time $O(n \log n)$ and space $O(n)$ whose weight is at most a constant factor times the weight of the Steiner minimum tree of the given points. They use this theorem to derive a PTAS for the Euclidean and rectilinear SMT problem that runs in $O(n \log n)$ time and consumes $O(n)$ space. In fact, even a somewhat simpler notion of banyans would suffice for the SMT problem: we do not need to have approximations of the SMT of *all* subsets of the terminals. It is sufficient that an approximation of the SMT of the whole set is included in the banyan. Zhao [85] used this concept in his PTAS for the geometric Steiner tree problem, in which the set of possible Steiner points are given in advance.

We will show in Chapter 9 that the existence of a polynomial time algorithm to find a $(1 + \epsilon)$-banyan of a set of terminals R in planar graphs with weight at most a constant factor times the weight of the SMT of R would lead to a PTAS for the SMT problem on planar graphs. As mentioned above, even an algorithm for the simpler notion of banyans would suffice. We conjecture that such an algorithm exists.

Chapter 7

PTASs for Geometric Variants

In this chapter we are going to review some of the developments of the recent years in the design of PTASs for geometric SMT problems. First, we are going to discuss Arora's approach [3] whose framework has been used to derive PTASs for many geometric problems. Then we are going to consider node-weighted SMTs and discuss the technique of Remy and Steger [74], which is based on Arora's framework but includes a number of new ideas to overcome the difficulties of this variant of the problem. Afterwards, we will turn to Mitchell's approach [60] and an application thereof [57] to find a PTAS for the rectilinear SMT problem in the presence of a constant number of obstacles. At the end of this chapter, we will shortly analyze why these approaches are not easily adaptable to the rectilinear/octilinear SMT problem with an arbitrary number of obstacles.

7.1 Arora's Approach

Arora [3] first published his PTAS for the traveling salesman problem and other related geometric problems such as the geometric SMT problem in 1996. Somewhat later he improved the performance of his algorithm considerably. Here we shortly discuss his improved approach on the rectilinear SMT problem based on the presentation in [71]. We would also like to mention Arora's [4] excellent survey on this approach, its applications on a number of geometric problems and its current limitations. We would like to notice that the algorithm is randomized, it produces a $(1+\epsilon)$-approximation with probability at least $1/2$. However, it can be derandomized at the cost of some additional runtime.

Let a set P of n points in the plane and a constant integer c be given. We describe a randomized algorithm that finds a rectilinear Steiner tree with weight at most $(1 + 1/c)$ times the weight of the minimum rectilinear Steiner tree of P with probability at least $1/2$. Arora's algorithm involves three major steps: perturbation, shifted quadtree construction and dynamic programming. But the centerpiece of the correctness and the polynomial runtime of the algorithm is a so-called structure theorem that usually involves a patching lemma. We are now going to discuss each of these steps and concepts, in turn.

In the *perturbation step*, one makes sure that the given set of terminals lies on odd integer coordinates inside a not-too-large bounding box. Specifically, it is shown that it suffices to consider sets of terminals that lie on the grid $\{1, 3, 5, \ldots, 4cn - 1\}^2$. An instance with this property is called a *well-rounded instance.* This is achieved by scaling the input set appropriately and then moving each terminal point to its closest grid-point (it might be that some terminals coincide on one grid point in which case they are merged into one point). One can argue that doing so will increase the cost of the optimum Steiner tree of the scaled problem by at most a factor of $(1 + 1/c)$, which is an accepted term for our purpose. From now on, we assume that our instance is well-rounded. Note that by using the Hanan grid [40], we know that there exists a rectilinear SMT that lies entirely on this grid of odd integers. Particularly, it is sufficient to only consider such grid points as possible Steiner points.

We introduce the notion of a *shifted quadtree* as in [74]. Let t and L be integers, such that $L = 2^t > 4cn \geq 2^{t-1}$. Put a bounding box of side-length L around the terminal set. Let a and b denote arbitrary integers from the set $\{0, 2, \ldots, L - 2\}$. Consider a vertical line with x-coordinate a and a horizontal line with y-coordinate b that divide the bounding box into four smaller rectangles. Enlarge those rectangles so that each one of them has side length L. Note that the resulting bounding box has now side length $L_0 = 2L$. Call this rectangle $\mathscr{R}_0$. Now we subdivide $\mathscr{R}_0$ into an ordinary quadtree: the root of the quadtree is $\mathscr{R}_0$ and each node $\mathscr{R}$ of the tree has 4 child nodes that are obtained by subdividing $\mathscr{R}$ into 4 equal sized rectangles. The leaves of the tree are rectangles of size 2×2. We denote this quadtree by $QT_{a,b}$ to emphasize its dependence on the integers a and b. Note that the depth of the tree is t and that each leaf contains at most one terminal or maybe one candidate Steiner point.

The basic idea of the *dynamic programming step* is simple: given a shifted quadtree $QT_{a,b}$, process it in a bottom-up fashion, starting from the leaves and working up to the root. For each node, store all possible "solutions" of the corresponding rectangle by combining them from the child nodes. At the root, the solution with the minimum weight will be the desired SMT.

However, we first have to define what a solution is. For a given non-root rectangle $\mathscr{R}$, a solution is a set of trees inside $\mathscr{R}$ such that every tree touches the border of $\mathscr{R}$ at least once and every terminal inside $\mathscr{R}$ is included in exactly one such tree. For the root node $\mathscr{R}_0$, a solution is a Steiner tree. But this algorithm is clearly not polynomial. In order to make it polynomial, we have to prove a *structure theorem* about rectilinear SMTs.

The structure theorem essentially tells that for every rectangle $\mathscr{R}$ in the tree, one can consider only a limited number of *portals* as possible crossing points for the trees inside $\mathscr{R}$ with the "world outside" and the number of such crossing may be bounded, either. Let m and r be parameters to be specified later. The portals are points on the boundaries of the rectangles that are determined by the choice of a, b and m in a way that every rectangle has at most $m-1$ of them on each of its four sides (we omit the details of the construction here). An (m, r)-light solution is defined as a solution that:

(i) shares no line segment with the boundary of any rectangle.

(ii) crosses/touches each side of the boundary of a rectangle at at most r points and only out of the set of at most $m-1$ given portals.

The structure theorem tells us that for a random choice of a and b, there exists with probability at least $1/2$ an (m, r)-light rectilinear Steiner tree with weight at most $(1 + 1/c)$ times the weight of the rectilinear Steiner minimum tree, where $m = O(c \log L)$ and $r = O(c)$. Hence, it suffices to look only at (m, r)-light solutions. In order to prove the structure theorem, one needs a so-called *patching lemma* that tells the following: if the optimal tree is not (m, r)-light, that is, it crosses the boundaries of rectangles too many times, the patching lemma shows how to modify the tree so as to reduce the number of crossings at a cost of no more than a factor of $1/c$ increase in the weight of the solution. This lemma is at the heart of the proof of why Arora's approach works.

Now one can make the dynamic programming work in polynomial time. For every node of the tree, we need to consider all possibilities of choosing at most r points out of $m-1$ portals on each side of the boundary and then partitioning these $4r$ points into at most $4r$ sets, each one building one tree. There are $O((mr)^{O(r)})$ possibilities to do this. Since $m = O(c \log n)$ and $r = O(c)$, this corresponds to $O((c \log n)^{O(c)})$ possibilities, which is polynomial in n. We call each such possibility a *configuration* of the node. Notice that the number of possible configurations for a node is the same for all nodes. For each node, we create a table that has an entry for each configuration. For every node and configuration, we store the length of the shortest (m, r)-light solution obeying the choices given by the configuration. If the considered node is a leaf, this value can be calculated in constant time by brute force. Otherwise, one considers all combinations of configurations of the four child nodes that are consistent with the given configuration and chooses the one that results in a tree with minimum length. Once the length of the minimum (m, r)-light Steiner tree is determined, it is possible to work out the tree in a top-down fashion to actually find the corresponding tree, as is standard in dynamic programming.

The runtime of this algorithm is $O(mn^2 \cdot (4mr)^{16r}) = O(n^2(\frac{1}{\epsilon}\log n)^{O(\frac{1}{\epsilon})})$ (recall that $m = O(c \log n)$, $r = O(c)$ and $c - 1 < \frac{1}{\epsilon} \leq c)$). But observe that it is in fact not necessary to consider the full shifted quadtree; we waste a lot of time on rectangles that contain no terminal at all. It is sufficient to stop the subdivision of a rectangle once it contains only one terminal. This way, the tree will have at most $2n$ leaves and so $O(n \log n)$ nodes. For a rectangle that contains only one node, it is possible to calculate the values of the configuration table in constant time. This modification reduces the runtime to $O(n(\frac{1}{\epsilon}\log n)^{O(\frac{1}{\epsilon})})$ and results in Arora's famous theorem:

Theorem 7.1 ([3]). *There exists a randomized algorithm that computes for every $\epsilon > 0$ and set of n terminals P in time and space $O(n(\frac{1}{\epsilon}\log n)^{O(\frac{1}{\epsilon})})$ with probability at least $1/2$ a rectilinear Steiner tree for P with weight at most $(1+\epsilon)$ times the rectilinear Steiner minimum tree of P.*

Note the important dependence of the algorithm on the random choice of the values a and b in the construction of the shifted quadtree. It is however possible to derandomize the algorithm by considering all possible choices of a and b, resulting in an increase of the runtime to $O(n^3(\frac{1}{\epsilon}\log n)^{O(\frac{1}{\epsilon})})$. Also, this algorithm can be extended to work for any constant dimension d in time and space polynomial in n.

In [73], Rao and Smith improved the running time of Arora's algorithm to $O((\frac{1}{\epsilon})^{O(\frac{1}{\epsilon})}n + n \log n)$ by making use of geometric banyans(see Section 6.5). They also show how to derandomize their algorithm preserving the running time of $O(n \log n)$. Their algorithm works for any constant dimension d with the same running time and it consumes $O(n)$ space (when ϵ and d are considered constants).

7.2 Node-Weighted Geometric Steiner Trees

In 2005, Remy and Steger [74] presented a PTAS for the node-weighted geometric Steiner tree problem. They follow Arora's basic framework but incorporate a number of significant new ideas to overcome the difficulties of this variant of the problem. The main difficulty is that there is no "natural" patching lemma for this problem and they had to come up with another kind of structure theorem. We are going to briefly discuss some of their ideas in this section.

In the node-weighted Steiner tree problem, there is a penalty $\pi(v)$ associated with every terminal v. For every terminal that is not included in the tree, its penalty is added to the cost of the tree. Furthermore, there is a penalty c_s given for Steiner points; every Steiner point added to the tree costs c_s units for the tree. We get the usual Steiner minimum tree problem as a special case of this problem by setting $\pi(v) = \infty$ for all terminals and $c_s = 0$. Another special case is the Steiner tree problem where the set of possible Steiner points is given in advance. This can be achieved by setting the penalty of terminals to ∞, adding the set of possible Steiner points to the set of terminals but with a penalty of 0 and setting $c_s = \infty$ (so no other Steiner points may be added). This case was also handled by a different approach (among other things, the use of spanners) by Zhao in [85].

This problem does not naturally admit a patching lemma, since requiring that the tree crosses the boundaries of the rectangles only at certain predefined portal points means forcing these portals to be Steiner points and Steiner points are not free of cost in this problem. Hence, a different approach has to be taken. Remy and Steger first make a perturbation step similar to that of Arora and then build a quadtree $QT_{a,b}$ with a random choice of a and b. But then, they subdivide each rectangle of the tree into $O(\log n)$ cells using (s,t)-maps. An (s,t)-map is a grid that contains $O(s^2 t)$ many cells so that the cells towards the inside are larger than those towards the boundary of the subdivided rectangle. The intuition behind (s,t)-maps is that long edges that reach deep inside a rectangle may be approximated more roughly than short edges whose endpoints are near the boundary. $t = O(\log n)$ is the depth of the quadtree and s is a parameter that may be chosen to be $O(1/\epsilon)$. The reason they use (s,t)-maps instead of an ordinary $m \times m$-grid is that they would have to choose $m = O(st)$ and thus having $O(\log^2 n)$ many cells and that would result in a super-polynomial algorithm. Their algorithm also takes another parameter r that specifies the maximum allowed number of connected components inside a rectangle and has to be chosen to be $O(1/\epsilon^2)$. They define an (r,s)-standardized solution with respect to a and b to be a solution that satisfies for every rectangle $\mathscr{R}$ of the quadtree $QT_{a,b}$:

(i) the number of connected components inside $\mathscr{R}$ is at most r,

(ii) for every cell in the (s,t)-map of $\mathscr{R}$, all vertices inside the cell that are incident to an external edge of $\mathscr{R}$ belong to the same connected component of $\mathscr{R}$.

An external edge of $\mathscr{R}$ is an edge that has one endpoint inside $\mathscr{R}$ and one outside. They show that an optimal (r,s)-standardized solution is also a nearly optimal solution overall. Property (i) is achieved by reconnecting the internal components of a rectangle by a Hamiltonian path. To bound the maximal overhead, they use the fact that the shortest salesman tour through k points inside a square of side length S has length at most $S\sqrt{k}$. To achieve property (ii) they apply reconnections inside a cell and redirections of external edges.

The configuration of a rectangle, used in the dynamic programming step, is quite different from that of Arora's algorithm. A configuration for a rectangle $\mathscr{R}$ is given by the following parameters:

(i) for every cell of the (s,t)-map of $\mathscr{R}$, a portal bit that indicates whether the cell is a portal cell;

(ii) for every cell of the (s,t)-map of $\mathscr{R}$, an anchor bit that tells whether the cell is an anchor cell; they require that every anchor cell is also a portal cell;

(iii) a partition of the portals of $\mathscr{R}$ into at most r sets.

A portal cell is a cell that has to be connected to some larger square. An anchor cell is a portal cell that has to be connected to the next larger square. The partition of the portals tells which portals have to build one connected component inside $\mathscr{R}$. The number of configurations for a rectangle is at most $2^{O(s^2 t \log r)}$. Note that if the number of cells inside a rectangle would be more than $O(\log n)$ the number of configurations would not be polynomial in n.

They use dynamic programming to find an almost optimal (r,s)-standardized solution in two passes. In the first pass, the structure of the solution is determined, not the actual tree. That is, the algorithm only decides about the inclusion of edges between the centers of the portals, not between specific vertices. This is done in a bottom-up fashion by calculating the best solution for every configuration of every node of the quadtree. For leaves, this can be done in constant time; for internal nodes, the best solution is determined by enumerating over all consistent configurations of the child nodes. If there is no solution for the given configuration of a given node, the cost of that configuration of that node is set to infinity. The second pass is a top-down traversal of the quadtree determining exactly which vertices inside the portal cells are to be connected. We leave out the exact details and refer the interested reader to the original paper [74]. All in all, we get the following theorem of Remy and Steger:

Theorem 7.2 ([74]). *For fixed $\epsilon > 0$, there is a polynomial time algorithm that computes a $(1+\epsilon)$-approximation of the node-weighted geometric Steiner tree problem on the plane.*

They also argue that their technique can be generalized to higher fixed dimensions but at the cost of losing polynomiality: they derive a QPTAS for fixed $\epsilon > 0$ and fixed $d \geq 3$ that calculates a $(1+\epsilon)$-approximation of the node-weighted geometric Steiner tree problem in d dimensions.

7.3 Mitchell's Approach

Mitchell introduced his idea of *guillotine subdivisions* to approximate polygonal subdivisions in 1996 [59], where he used it to obtain a constant factor approximation for the k-MST problem. Somewhat later, he generalized this notion to the notion of *m-guillotine subdivisions* that enabled him to present PTASs for the Steiner tree problem, Steiner k-MST problem, traveling salesman problem and similar problems on the 2-dimensional plane [60]. Here, we are first going to discuss the simpler concept from [59] and then the generalization in [60].

A *polygonal subdivision* S is a (not necessarily connected) planar straight-line graph drawn on the plane. Let E denote the set of edges of S, V the set of vertices and let the number of edges be n. Without loss of generality, we may assume that S is restricted to the unit square B. Then, each face of S is a bounded polygon, possibly with holes. The *length* of S is the sum of the lengths of its edges. If all edges are horizontal or vertical, we say that S is rectilinear. A closed axis-aligned rectangle W is a *window* if $W \subseteq B$. In the following, we focus attention on a given window W.

A line ℓ is a *cut* for E with respect to W if ℓ is horizontal or vertical and has a nonempty intersection with the interior of W. The intersection of a cut ℓ with the part of E inside W (that is, the intersection $\ell \cap E \cap \text{int}(W)$) consists of a discrete (possibly empty) set of subsegments of ℓ. The endpoints of these subsegments are called the *endpoints along* ℓ with respect to W. Note that the two points where ℓ crosses the boundary of W are not considered endpoints. Let ξ be the number of endpoints along ℓ and denote these points by $p_1, \ldots, p_\xi$ in order along ℓ. The *span of* ℓ is the convex hull of $E \cap \ell$ and is equal to the line segment $p_1 p_\xi$. We denote it by $\sigma(\ell)$. A cut ℓ is called *perfect* if $\sigma(\ell) \subseteq E$. This happens exactly if $\xi \leq 2$.

We say that a polygonal subdivision is a *guillotine subdivision* with respect to a window W if either

(i) $E \cap \text{int}(W) = \emptyset$; or

(ii) there exists a perfect cut ℓ with respect to W such that S is guillotine with respect to the windows $W \cap H^+$ and $W \cap H^-$, where H^+ and H^- are the closed half-planes induced by ℓ.

We say that S is a guillotine subdivision if it is guillotine with respect to the unit square B.

The main theorem of Mitchell is that for any given rectilinear subdivision S of the unit square B with edge set E and length L, there exists a guillotine subdivision S_G with edge set E_G that contains E and has length at most $2L$. Since a rectilinear k-MST of a set of points inside B is a rectilinear subdivision of B, it follows that there exists a guillotine subdivision of B that contains the optimal rectilinear k-MST and has length at most twice as much. Mitchell shows that this guillotine subdivision can be found in polynomial time using dynamic programming and hence, the 2-approximation algorithm follows. For the Euclidean case, the factor changes to $2\sqrt{2}$.

The proof of the main theorem is based on the idea of *dark points* and *favorable cuts*. Imagine that the two vertical boundary lines of a window W are light sources and project parallel horizontal light into W and imagine that the edge set E is a set of obstacles in W. Let ℓ be a vertical cut with respect to W. We say a point p of ℓ is *horizontally dark* if it receives no light from the sources, that is, some points of E block its view from the vertical boundary lines. Let the closest blocking point from the right be p^+ and the closest from the left be p^- (p^+ is obtained by projecting p to the right on E and p^- is obtained by projecting p to the left on E; note that if $p \in E$ then $p^+ = p^- = p$). Similarly one can define *vertically dark* points on a horizontal cut.

An interesting property of dark points is that if a subsegment pq of a vertical cut ℓ is entirely dark, one can "charge" its length to the left and right as follows: charge half of its length to $(pq)^+$ and the other half to $(pq)^-$. The same property holds of course for horizontal cuts, too. A *favorable cut* is a cut whose dark portion is at least as long as the span of ℓ. If one finds a favorable cut ℓ, one can cut the window W along it and charge the length of its span to the left and right (respectively, up and down) as we just described. Since every line segment of E is charged to at most once from every side, half of its length, it follows that the total length of the spans of favorable cuts and hence, the length of the resulting guillotine subdivision, is not more than the total length of E. Mitchell shows that whenever there is no perfect cut for a window W, there always exists a favorable cut for W (his method is not hard and involves only a little bit of elementary integration laws). Since the total length of favorable cuts is at most L and the total length of perfect cuts is also at most L, we get that the length of S_G will not exceed $2L$. So, the theorem follows.

In [60], Mitchell generalizes this method to obtain PTASs for a number of $\mathcal{NP}$-hard geometric problems. To this end, he introduces the notion of m-guillotine subdivisions. The *m-span*, $\sigma_m(\ell)$, of a cut ℓ is defined as follows: if the number of endpoints ξ along ℓ is $\leq 2(m-1)$ then $\sigma_m(\ell) = \emptyset$, otherwise we have $\sigma_m(\ell) = p_m p_{\xi-m+1}$, that is, the line segment connecting the mth endpoint along ℓ to the mth-from-the-last endpoint along ℓ. A cut is *m-perfect* if its m-span is contained in E. The definition of an *m-guillotine subdivision* is the same as a guillotine subdivision replacing perfect cuts with m-perfect cuts. The main theorem generalizes as follows:

Theorem 7.3 ([60]). *Let S be a rectilinear subdivision with edge set E of length L. Then, for any positive integer m, there exists an m-guillotine rectilinear subdivision S_G of length at most $(1 + 1/m)L$ whose edge set E_G contains E.*

The proof is exactly the same as the previous case only replacing the concept of dark points with *m-dark* points. A point p of a cut ℓ with respect to a window W is m-dark, if there exist at least m endpoints of $\ell^{\perp}$ with respect to E on each side of ℓ. That is, it must be possible to project p on each side of ℓ on at least m “layers” of E — not only on the closest one as before. A favorable cut is then a cut whose m-dark portion is at least as long as its m-span and again, Mitchell shows that if there is no m-perfect cut for a window, then there always exists a favorable cut and the theorem follows.

It remains to discuss the dynamic programming step. We do this for the rectilinear Steiner tree problem. A rectilinear Steiner tree of a set of n terminal points P inside the unit square B is a rectilinear subdivision of B and hence, for any positive integer m, there exists an m-guillotine subdivision of B that contains the optimal Steiner tree and which is longer by a factor of at most $(1+1/m)$. We need only consider the Hanan grid [40] of P and hence, it is sufficient to only consider the x-coordinates of P and one point in between any two consecutive x-coordinates of P as possible candidates for vertical cuts (respectively, y coordinates for horizontal cuts). So, there are only $O(n)$ of them. In the dynamic programming, one proceeds bottom-up and considers every one of the $O(n^4)$ relevant windows inside B from smallest to largest. For each such window W, a *configuration* is given by:

(i) a selection of at most $2m$ endpoints on each of the four sides of W;

(ii) a partition of these at most $4m$ boundary segments.

There are $O(n^{8m})$ choices for the endpoints and $O(m^m) = O(1)$ choices for the partition. The partition tells which boundary segments are to be connected inside this window. For every configuration of every window, we can now calculate the length of the shortest m-guillotine subdivision containing all the terminals of that window and satisfying the requirements of the given configuration. If the window contains no terminals, this value is either 0 or ∞ depending on weather the configuration requirements are satisfied or not. Otherwise, one can find the optimal value by considering the subproblems arising from the following choices and selecting the one with minimum total cost:

(i) $O(n)$ choices of a cut by a horizontal/vertical line;

(ii) $O(n^{2m})$ choices of endpoints along the cut;

(iii) $O(m^m) = O(1)$ choices to partition the new boundary segments on each side of the cut.

Overall, we get an algorithm that runs in time $O(n^{10m+5})$. For a given $\epsilon > 0$ we can choose $m \geq 1/\epsilon > m - 1$ and obtain an $O((1/\epsilon)^{1/\epsilon} \cdot n^{10/\epsilon+5})$-time algorithm that calculates a $(1 + \epsilon)$-approximation for the rectilinear Steiner tree problem.

7.4 The Problem with Obstacles

In this section, we will shortly discuss how much these ideas are or are not applicable to the rectilinear Steiner tree problem *with obstacles*. In [57], Liu et al. observe that one can extend Mitchell's algorithm [60] to work with rectilinear obstacles by a simple modification but at the cost of increasing the runtime. If the number of obstacles is k, the runtime of Mitchell's algorithm will become $O((1/\epsilon + k)^{1/\epsilon+k} \cdot (n + k)^{10/\epsilon+5})$. This is polynomial in n if $k = O(1)$. Hence, they obtain a PTAS for the case of a constant number of obstacles. Furthermore, one sees that if $k = O(\text{polylog}(n))$, then this algorithm runs in quasi-polynomial time and one obtains a QPTAS (as far as we know, this latter observation has not been stated in the literature yet).

Liu et al. [57] argue that there exists an obstacle-avoiding m-guillotine subdivision that contains the optimal obstacle-avoiding rectilinear Steiner tree. They find the minimal obstacle-avoiding connected m-guillotine subdivision that contains all terminals using Mitchell's dynamic programming method with one modification: when considering a new cut, one chooses the at most $2m$ endpoints along the cut, first ignoring the obstacles, but then breaks down each of the at most m line segments into smaller obstacle-avoiding segments. So, one might end up with at most $m + k$ line segments on each side of a window and has to consider all partitions of these $O(m+k)$ line segments on the boundaries of the windows. There are $O((m + k)^{m+k})$ such partitions and hence, the increase in the running time.

When considering Arora's approach [3], we see that the patching lemma is not satisfied anymore, if we allow obstacles. For the patching lemma or for similar structure theorems like the one from Remy and Steger [74] to work, it seems to be necessary to be able to reconnect terminals inside a rectangle in order to reduce the number of crossings or the number of connected components. But when dealing with obstacles, this is sometimes not possible at all. Even when possible, the added overhead can not be easily estimated since one lacks "nice" geometric properties (like the bound on the length of a salesman tour inside a rectangle) when obstacles are present.

Both Mitchell's and Arora's algorithms are based on the observation that for the rectilinear Steiner tree problem, one can restrict the number of connected components inside a given rectangle to some constant at the cost of a small increase in the total length of the minimal tree. But with obstacles, one might be forced to have as many as $O(\min\{n, k\})$ connected components inside a rectangle and we do not know yet how to overcome this issue to make their algorithms applicable to this problem.

Chapter 8

PTASs for Planar Graphs

In this chapter, we are going to review a number of techniques for designing PTASs for various $\mathcal{NP}$-hard optimization problems on planar graphs. The first PTAS on planar graphs was given in 1980 by Lipton and Tarjan [56] for the maximum independent set problem using their planar separator theorem that they had proven a year before [55]. We are not going to discuss their method directly but we will focus on planar separators and weighted planar separators with regard to the traveling salesman problem (TSP) in Sections 8.2–8.3. Before we turn to that, we review Baker's approach [7], which was a milestone in this area of research and led to PTASs for a number of problems on planar graphs. In the last section of this chapter, we will introduce Klein's framework [47] to derive a linear time PTAS for the weighted planar TSP. This framework will also be the focus of the next chapter, where we propose a potential method to derive a PTAS for the Steiner tree problem on planar graphs.

Though we are not going to discuss it here, we would also like to mention the *bidimensionality theory* introduced by Demaine and Hajiaghayi [26] that characterizes a number of problems — including problems on planar graphs — for which there exists a PTAS. Unfortunately, "subset problems" such as the Steiner tree problem do not comply with their theory. See also Hajiaghayi's PhD thesis [39]. An early attempt to syntactically characterize PTAS was given by Khanna and Motwani [46].

8.1 Baker's Decomposition

Baker's approach [7] is based on decomposing the graph into so-called k-*outerplanar* graphs. We will define this class of graphs in a moment. Many $\mathcal{NP}$-hard optimization problems can be solved on k-outerplanar graphs in time polynomial in the number of vertices n. Roughly speaking, Baker decomposes a planar graph into a number of k-outerplanar components, solves the problem optimally in each one of them and aggregates the solution to one solution for the main problem. She considers k possibilities to do this and argues that at least one of them is within a factor of $1 + \frac{1}{k}$ of the optimum (or $1 - \frac{1}{k}$ for maximization problems). Her method works for a number of optimization problems on planar graphs, such as maximum independent set, minimum vertex cover, maximum tile salvage, partition into triangles, maximum H-matching, minimum dominating set and minimum edge dominating set. Here, we are going to consider maximum independent set as an example.

An *outerplanar embedding* of a graph G, is a planar embedding of G such that all vertices lie on the infinite face. A graph G is called *outerplanar* if it has an outerplanar embedding. For an embedded planar graph G, we define *levels* for the vertices as follows: the vertices on the infinite face have level 1; the vertices that would lie on the infinite face if we removed the vertices with level 1 to i are on level $i + 1$. We say G is *k-outerplanar* if the maximum level of its vertices is k. We say a graph is k-outerplanar if it is has a k-outerplanar embedding. Every planar graph is k-outerplanar for some k. For a given planar graph G, it is possible to find a k-outerplanar embedding with minimum k in polynomial time [11]. One can imagine a k-outerplanar graph as consisting of k "rings" or "layers". The optimization problems mentioned in the last paragraph can all be solved on k-outerplanar graphs in polynomial time using dynamic programming. The dynamic programming is somewhat complicated and we refer the interested reader to the original paper of Baker [7]. For the maximum independent set problem, the algorithm takes time $O(8^k \cdot n)$, which is linear in n.

Let a planar embedded graph G be given. For $0 \leq i \leq k - 1$, consider the graphs G_i induced by the vertices at levels $jk + i$ for $j = 0, 1, 2, \ldots$. For any such i, the graph $G - G_i$ has a number of connected components, each of which is k-outerplanar. If one calculates the maximum independent set in each of these k-outerplanar components, then the union of those is an independent set for G. Now, if we do this for every $0 \leq i \leq k - 1$ and take a solution with maximum size, we have an algorithm that calculates an independent set for G in time $O(8^k k \cdot n)$. Indeed, the size of this independent set is at least $(1 - 1/k)$ times the size of a maximum independent set in G. To see this, let S be a maximum independent set in G. Since the G_i build a partition of G into k sets, there is at least one t for which the graph G_t contains at most a $1/k$ fraction of the vertices in S. That is, $|S \cap V(G - G_t)| \geq (1 - \frac{1}{k})|S|$. With the algorithm above, we calculate the *maximum* independent set in the graph $G - G_t$ and this is at least as large as $|S \cap V(G - G_t)|$ (which is *some* independent set in $G - G_t$). Hence, the approximation ratio follows and we are done.

8.2 Planar Separators and Planar TSP

In 1995, Grigni et al. [37] presented the first PTAS for unweighted planar graph TSP. They proved their own planar separator theorem and used it to decompose the graph and then use dynamic programming to find an approximate solution to the problem. They conjectured that the Euclidean TSP also admits a PTAS, since they believed that any hardness reduction to Euclidean TSP involves planar graph TSP. Indeed, less than two years later their conjecture was shown to be true by Arora [3] (see Chapter 7). Here we are going to briefly review their main ideas. We will have a closer look in the next section, where we talk about weighted planar TSP.

Their algorithm is based on the following version of the planar separator theorem that they prove in their paper:

Theorem 8.1 ([37]). *Given a connected embedded planar graph H with weights on its vertices and a parameter f such that $1 \leq f \leq \sqrt{|H|}$, there is a simple planar cycle C through $O(|H|/f)$ vertices of H such that at most f of the arcs of C are face-edges, the remaining arcs are ordinary edges, and the interior and exterior of C both have at most $2/3$ of the total weight. Such a C may be found in polynomial time (in fact nearly linear time).*

Face-edges are imaginary edges that run through the middle of a face. The cycle C from the theorem above uses at most f face-edges and hence, is comprised of at most f "real" paths. One can use this cycle to split a graph H the following way: first, we contract each of these at most f paths into single vertices (remove loops and double edges to keep the graph simple). Call these vertices, *constraint points.* Then the cycle C is reduced to a cycle consisting only of face-edges. Let H_1 be the subgraph in the interior of this cycle together with the constraint points and let H_2 be the subgraph in the exterior of this cycle together with the constraint points. They show that C can be chosen so that H_1 and H_2 are connected planar graphs. Note that any vertex of H_1 or H_2 is either an original vertex from H or a constraint point. Let G be the given planar embedded graph. One uses Theorem 8.1 to decompose G and build a binary decomposition-tree as follows: the root of the tree is G. Every node in the tree that represents a subgraph H with more than $S = \theta(f^2)$ (to be specified later) vertices, is split into two child nodes H_1 and H_2 as described above. In the splitting process, we assign weight 1 to every original vertex of the graph G and weight $W(H)/6f$ to all constraint points — both new ones and the ones inherited from ancestors in the tree. This way, one can show that every child node H_i $(1 \leq i \leq 2)$ of a subgraph H has weight at most $\frac{5}{6}W(H)$. An immediate consequence is that no subgraph appearing in the decomposition can have more than $5f$ constraint points. Another consequence is that every non-leaf of the tree contains at least one original vertex of G (because $S > 5f$) and hence, the depth of the tree is at most $D = \log_{\frac{6}{5}} n < 4 \log n$. We choose $f = O(\log n/\epsilon)$ and $S = 10Dcf/\epsilon$ for some large enough constant c. One can argue that the total number of constraint points in all the leaves of the tree is at most $\frac{n\epsilon}{c}$ and hence, the total number of vertices in all the leaves is no more than $(1 + \frac{\epsilon}{c})n$. Note that the decomposition tree can be built in polynomial time — in fact, in nearly linear time, independent of the parameter ϵ.

In the dynamic programming step, the tree is processed in a bottom-up manner. To this end, they consider a slightly more general optimization problem. Let H be a connected planar graph and X a subset of its vertices with $|X|$ even. An (H, X)-solution is a collection of paths that visits every vertex of H at least once and such that the endpoints of the paths are exactly X (so, it is comprised of $|X|/2$ paths). For the special case where X is empty, the (H, X)-solution should contain a single tour that covers H. Let $c^\star(H, X)$ be the minimum cost of any (H, X)-solution.

At the leaves of the tree, they use an approximation algorithm that solves the given problem for the subgraph H and every even subset of its vertices X up to a relative error of $\epsilon/4$. This base case algorithm is similar to the main algorithm but somewhat simpler. The base cases in there have size $O(\log n)$ and are solved exhaustively. The total time for handling the leaves is $O((n + 2^{1/\epsilon})^{1/\epsilon})$. For inner nodes of the tree, the optimal (H, X)-solution can be found by considering all consistent combinations of solutions of the child nodes, aggregating them into an (H, X)-solution and taking the one with minimum cost. We leave out the details of this process at this place. The running time of the dynamic programming step is $O(n^{1/\epsilon})$. In the analysis, they argue that the total leaf error, patching error and contraction error does not exceed an additive error of ϵn and hence, their algorithm delivers the promised approximation guarantee.

8.3 Weighted Planar Separators and Weighted Planar TSP

The first PTAS for weighted planar graph TSP was given by Arora et al. [5] in 1998. They used Althöfer et al.'s planar spanner theorem [2], Theorem 6.2, to filter out edges and then they use their own weighted planar separator theorem to decompose the graph. Afterwards, dynamic programming can be applied to find an approximate solution. Now to some more detail.

Let the input graph be $G_0 = (V, E_0)$. Using Althöfer et al.'s spanner [2], they obtain a subgraph $G = (V, E)$ of G_0 with the following properties:

(i) for all vertices $x, y \in V$, $d_G(x,y) \leq (1+\frac{\epsilon}{2})d_{G_0}(x,y)$;

(ii) the total weight of G is at most $O(1/\epsilon)$ times the weight of the minimum spanning tree of G_0.

Let the cost of the optimal tour in G_0 be denoted by OPT. From property (i), we see that the cost of the optimal tour in G is at most $(1+\frac{\epsilon}{2})$ times OPT, since one can simply replace every edge of the optimal tour in G_0 with a path in G that is longer by at most a factor of $(1+\frac{\epsilon}{2})$. From property (ii), we get that the total cost of G is at most $O(1/\epsilon)$ times OPT, since the cost of any tour in G_0 is $\geq$ the cost of the minimum spanning tree of G_0.

Arora et al. [5] prove the following weighted planar separator theorem:

Theorem 8.2 ([5]). *Given a parameter k and an n-vertex planar embedded graph G with edge-costs, vertex-weights and face-weights, there is a poly(n)-time algorithm that finds a Jordan curve C, comprised of ordinary edges and face-edges of G, such that*

balance condition: *the interior and exterior of C have weight at most* $2/3$ *of the total weight;*

face-edge condition: *C uses at most k face-edges;*

ordinary-edge condition: *C uses ordinary edges of total cost at most $O(1/k)$ times the total cost of all the ordinary edges.*

This theorem can be used to decompose the graph G similarly to what was done in the previous section. One finds the cycle C and contracts its ordinary edges. The ordinary-edge condition ensures that not much weight is contracted in this step. Let the contracted version of C be C'. By the face-edge condition, C' consists only of at most k face-edges and hence, has at most k vertices. Call these vertices *boundary vertices*. Let the interior of C' together with the boundary vertices be the interior piece of G and the exterior of C' together with the boundary vertices be the exterior piece. Both for the interior and the exterior piece, we have the property that the boundary vertices lie on the boundary of a single face. We call this face a *hole*, since it results from the removal of some part of the graph. The connected components of the interior piece and the exterior piece are the *children of* G. From the balance condition, we get that each child of G has weight at most a constant fraction of G. Moreover, we still have for every child that the boundary vertices lie around a single face, a hole, of the child.

Now we can build a decomposition tree. Fix the parameter $k = c\log n/\epsilon^2$, where c is a constant to be determined. Let G be the root of the tree and for each node in the tree, decompose it as described above and add its children to the tree until each leaf represents a graph with a small number of vertices.

Since the weight of every child is at most a constant fraction of its parent, we get that the depth of the tree is $O(\log n)$. Let G' be the graph obtained from G by contracting all the edges appearing in some separator in the tree decomposition above. We would like to bound the total weight of contracted edges. Consider all the graphs appearing at a particular level of the tree. These graphs are edge-disjoint and hence, the sum of the weights of the edges appearing in all of them is at most the weight of G. For each graph, the cost of the separating cycle is at most $O(1/k)$ times the cost of the graph and summing up these values gives us a bound of $O(1/k) \cdot w(G) = O(OPT/k\epsilon)$. Now summing over all levels of the tree, we can bound the total weight of contracted edges by $O(OPT \log n/k\epsilon)$. By appropriate choice of c in the definition of k, we can ensure that this bound is at most $\epsilon/4$ times OPT. We will describe below, how to find the *optimal tour* in G' using dynamic programming. Once we have that, we can uncontract the separator edges and add them to the tour of G' using the classic double-MST heuristic. The cost of the tour increases by an additive value of at most twice the weight of the contracted edges, i.e. $\epsilon/2 \cdot OPT$. Hence, we find a tour in G that exceeds the cost of the optimal tour of G by at most $\epsilon/2 \cdot OPT$. The other half of the allowed error is taken by the spanner subgraph approximation in the first step of the algorithm and so, we get an algorithm that delivers a $(1 + \epsilon)$-approximation of the optimal TSP tour in G_0.

We use dynamic programming to find the optimal tour in G'. For every subgraph appearing in the tree, we have to consider its boundary vertices. A simple *patching lemma* shows that there is an optimal tour that crosses each boundary vertex of a subgraph at most twice. Hence, it is sufficient to consider each subgraph with each possible configuration of such crossings. Suppose the number of boundary vertices of a subgraph is p. We can specify the boundary-crossings by a list of $2p$ numbers between 1 and p where each number appears at most twice. There are $p^{O(p)}$ of such lists. For every list of every subgraph of the decomposition tree we can find the optimal solution: if the subgraph is a leaf, we use exhaustive search; if it is an internal node, we can find the optimal solution by enumerating over all consistent possibilities of the children.

In order to bound the running time of the algorithm we have to find a bound on p, the number of boundary vertices of a subgraph, and its possible crossing-lists. We know that every cycle separator generates at most k boundary vertices and so, a subgraph at level d hast at most $dk = O(\log^2 n/\epsilon^2)$ boundary vertices. But using this bound, would only result in a quasi-polynomial-time algorithm. Indeed, by choosing the weight of vertices, boundary vertices and holes appropriately, one can achieve that any subgraph appearing in the tree, regardless of its level in the tree, has at most $10k$ boundary vertices. Furthermore, the same weighing scheme achieves that every subgraph has at most 10 holes. Using an analysis technique similar to the analysis of Catalan bounds (the number of balanced arrangements of pairs of parentheses), Arora et al. show that the number of crossing arrangements is bound by $2^{O(k)}$ and so, the overall runtime of the algorithm becomes $O(n^{1/\epsilon})$ — which is polynomial in n.

8.4 Klein's Approach and Weighted Planar TSP

In 2005, Klein [47] presented a *linear time* approximation scheme for weighted planar graph TSP. He proposed a framework for obtaining such PTASs that could potentially be applied to other problems, too — such as the Steiner tree problem. His framework consists of the following steps:

Filtering step: Delete some edges of the input graph while approximately preserving the optimal value. This can be done using a spanner — or in the case of Steiner trees possibly with banyans.

Thinning step: Apply thinning to the planar dual, effectively contracting some edges in the primal. This step should not increase the optimal value.

Dynamic-programming step: Use dynamic programming to find the optimal solution in the thinned graph.

Lifting step: Convert the optimal solution found in the previous step to a solution for the pre-thinned graph by incorporating some of the edges contracted during thinning.

For the weighted planar traveling salesman problem, he shows how to perform each of these steps in time linear in the number of vertices. In 2006, he shows how to construct a subset spanner for planar graphs [49](see Section 6.4) and uses it as the filtering step to obtain an $O(n \log n)$-time algorithm for subset TSP. In the next chapter, we are going to look at each of the steps of this framework — applied to the planar Steiner tree problem — in thorough detail. As we will see, the only missing part is an appropriate filtering step.

Chapter 9

Our Conjecture: The Planar SMT Problem Admits a PTAS

Problem 9.1 (Planar SMT Problem). *Given an undirected connected $n-$vertex planar graph $G_0 = (V_0, E_0)$ with edge weights $w_e \geq 0$ and a subset $R \subseteq V$ of terminal vertices, find a subtree of G_0 that includes all the terminals and has minimum total edge weight.*

Conjecture 9.2. *Problem 9.1 admits a PTAS.*

The Steiner minimum tree problem is one of the few classic problems where it is not known whether it admits a PTAS on planar graphs or not. Our conjecture is that it does admit a PTAS and we suggest that Klein's framework [47](see Section 8.4) together with a notion of banyans [73](see Section 6.5) for planar graphs could result in such a polynomial time approximation scheme. As discussed in Section 5.4, if this conjecture is true, then both the rectilinear and octilinear [63] Steiner tree problems with obstacles and maybe even some more Hanan-grid type problems [40, 84] will be shown to admit a PTAS.

In this chapter, we are going to discuss each of the steps of Klein's framework that were introduced in Section 8.4 in detail and show that each one of them can be applied to the Steiner tree problem, except the first step, namely, the filtering step. For some restricted inputs, the filtering step is automatically given and for those inputs, we derive a PTAS. However for most interesting cases, the filtering step is missing and we show that a polynomial time algorithm to find a $(1 + \epsilon)$-banyan (actually even a somewhat simpler construct) on planar graphs would result in a polynomial time algorithm to find a $(1+\epsilon)$-approximation of the Steiner minimum tree problem on planar graphs.

Let $OPT(G, R)$ denote the weight of a Steiner minimum tree of a set of terminals R in a graph G. In the context of Problem 9.1, we denote the value of $OPT(G_0, R)$ simply by OPT. We are looking for a solution with a value of at most $(1+\epsilon) \cdot OPT$. In the first section of this chapter, we discuss edge compression in relation to the Steiner tree problem. In the next sections, we will describe each of the steps of Klein's approach in detail and in the last section, we will see how these steps come together to potentially deliver a PTAS for Problem 9.1.

9.1 Edge Compression and Steiner Trees

In this chapter, we rely on the combinatorial definition of planar graphs, see Section 6.2. Remember that according to this definition, a planar embedded graph is defined in terms of its edge-set and the dual of a planar embedded graph has the same edge-set as the primal. Also, recall that *compression* is deletion in the dual and has the effect of contraction, except for loops: when compressing a loop of a vertex x, the vertex x is split into two vertices x_1 and x_2 and the graph falls apart into two connected components, namely, one containing the interior of the loop together with x_1 and one containing the exterior of the loop together with x_2. Compression is denoted by the / operator. In order to deal with compression in the context of Steiner trees, we have to first state some definitions and lemmas. The problem is that we have to specify what happens to the terminal set, when we compress or uncompress edges.

For a not-necessarily-connected planar embedded graph G and a set of terminals R, define a *Steiner multitree* to be a set of trees — at most one in every connected component of G — such that every terminal from the set R is included in the tree of its connected component. Denote the weight of a Steiner minimum multitree of R in G by $OPT(G, R)$. Let $e = \{x, y\}$ be an edge of G. When compressing the edge e, we would like to specify the new terminal set R/e in G/e. Assume that e is not a loop, i.e. $x \neq y$ and let z be the vertex obtained from compressing e. If any of x or y are terminals in G, we let $R/e = (R \setminus \{x, y\}) \cup \{z\}$, otherwise we let $R/e = R$. Now assume e is a loop, i.e. $x = y$. Let x_1 and x_2 be the vertices obtained from splitting x and belonging to connected components H_1 and H_2, respectively. Let $X = \{x_i : H_i \text{ contains a vertex from } R \setminus \{x\}\}$. If x is a terminal in G, then we let $R/e = (R \setminus \{x\}) \cup X$. But even if x is not a terminal in G but both H_1 and H_2 contain terminals of G, we let $R/e = R \cup X$. Otherwise, we let $R/e = R$. Notice that with this definition of X, we avoid creating connected components that contain only one terminal. Let $S = \{e_1, \dots, e_s\}$ be a set of edges of G to be compressed. Define $R/S = (\dots(R/e_1)/e_2)/\dots)/e_s$. Note that the order of compression does not affect the final value of G/S and R/S. For a multitree T of G, let T/S denote the multitree of G/S obtained from compressing S in T in G. Note that T/S does depend on G, since compressing a loop that is not in T might still cause T/S to become disconnected. We have the following two lemmas:

Lemma 9.3. *Let G be an edge-weighted planar embedded graph, let $R \subseteq V(G)$ be a set of terminals, let T be a Steiner multitree of R in G, and let S be a set of edges of G. Then T/S is a Steiner multitree of R/S in G/S and hence, $OPT(G/S, R/S) \leq OPT(G, R)$.*

Proof. The proof is by induction on $|S|$. If $|S| = 0$, the lemma is trivially true. Otherwise let $e = \{x, y\}$ be a an edge of S and let $S' := S \setminus \{e\}$. By the induction hypothesis, we know that $T' := T/S'$ is a Steiner multitree of $R' := R/S'$ in $G' := G/S'$. If e is not a loop, then one can easily check that T'/e is a Steiner multitree that contains all the vertices of R'/e and we are done. If e is a loop, i.e. $x = y$, let H be the connected component of G' containing e. Let x_1 and x_2 be the two new vertices created by the compression and H_1 and H_2 be the two new connected components of H/e including x_1 and x_2, respectively. If either H_1 or H_2 have no edges, then the case becomes trivial. Assume that each one has at least one edge. It follows that x is a cut vertex of H. We have the following cases:

(i) $x \in T'$: then $x_1, x_2 \in T'/e$ and so, all the vertices of R'/e are included in T'/e.

(ii) $x \notin T'$: since x is a cut vertex of H, it follows that T' can have edges in at most one of H_1 and H_2. Also it follows that $x \notin R'$ and so $x_1, x_2 \notin R'/e$. So, T'/e still contains all the vertices of R'/e.

Hence, the first part of the lemma is proven. Furthermore, we have that

$$OPT(G/S, R/S) \leq w(T/S) \leq w(T) = OPT(G, R) \ .$$

□

Lemma 9.4. *Let G be an edge-weighted planar embedded graph, let $R \subseteq V(G)$ be a set of terminals, and let S be a set of edges each having weight zero. Then $OPT(G, R) = OPT(G/S, R/S)$.*

Proof. By Lemma 9.3, we have that $OPT(G/S, R/S) \leq OPT(G, R)$. Conversely, let T'' be any Steiner multitree of R/S in G/S. We show by induction on $|S|$, how to lift the multitree T'' to obtain a Steiner multitree

T of R in G with $w(T) \leq w(T'')$. The case $|S| = 0$ is trivial. Let e be an edge of S and let $S' := S \setminus \{e\}$. So, $G' := G/S'$ is the graph obtained from uncompressing e. Let e have the vertices $x, y \in G'$ as its endpoints. We show how to obtain a Steiner multitree T' of $R' := R/S'$ in G' with $w(T') \leq w(T'')$. Then, the induction hypothesis delivers us the desired tree T and we are done.

First assume $x \neq y$ and let $z \in G'/e$ be the vertex obtained from compressing e. We set $T' = T''$ and include e in T' iff $z \in T''$. Since the weight of e is zero, we have that $w(T') = w(T'')$ and one can easily check that T' will be a Steiner multitree of R' in G'.

Now assume that e is a loop, i.e. $x = y$. We set $T' = T''$. We have to argue that T' will be a Steiner multitree of R' in G'. Let H be the connected component of G' containing e and define x_1, x_2, H_1, and H_2 as in the proof of the previous lemma. The case when either H_1 or H_2 contains no edge is trivial. Assume H_1 and H_2 both have at least one edge and hence, x is a cut vertex of H. Let T''_1 be the tree of T'' inside H_1 and T''_2 be the tree of T'' inside H_2. If H_1 contains no terminal of R', we remove T''_1 from T' and if H_2 contains no terminal of R', we remove T''_2 from T'. In these cases, we have $w(T') \leq w(T'')$ and T' is clearly a Steiner multitree of R' in G'. The only remaining case is when both H_1 and H_2 contain a terminal of R'. But according to our definition of R'/e, we have in this case that both x_1 and x_2 are terminals of R'/e and are thus included in T'', i.e. T''_1 includes x_1 and T''_2 includes x_2. So, x is included in T' and thus, $T' \cap H$ is a (connected) Steiner tree in H. ☐

9.2 Step 1 (The Missing Step): Filtering

Now we turn to the first step of Klein's approach, the filtering. The goal of the filtering step is to find a subgraph $G = (V, E)$ of $G_0 = (V_0, E_0)$ such that

(i) the cost of the optimal solution in G is at most $(1 + \frac{\epsilon}{2})$ times OPT;

(ii) and the total weight of edges in G is no more than some ρ_ϵ times OPT.

If $\rho_\epsilon = O(1/\epsilon)$ then one can often hope to achieve a linear time approximation scheme in the next steps of this framework. If $\rho_\epsilon = O(\log n)$, then it might still be possible to achieve a polynomial time approximation scheme. For the case of the traveling salesman problem, one can simply use the planar spanner given by Althöfer et al. [2] (see Section 6.1) or even better, the linear time algorithm given by Klein [47] that finds a spanner with the same properties (see Section 6.2). For both of these algorithms, we have that

$\rho_\epsilon = (1 + 2/\epsilon) = O(1/\epsilon)$ and hence, one derives a linear time approximation scheme for TSP.

Unfortunately, these spanner results can not be used for the Steiner tree problem. The resulting spanner would satisfy property (i), since one can replace every edge of the optimal Steiner tree of G_0 with an approximate path in G, but it would not necessarily satisfy property (ii): the cost of the minimum Steiner tree can be arbitrarily smaller than the cost of the minimum spanning tree (whereas in TSP it is the other way round: the cost of a TSP is larger than that of the MST). *However, if we know in advance that the cost of the SMT is $O(1)$ times the cost of the MST, then these spanner results can indeed be used.* This is for example the case for unweighted graphs with a set of terminals that has $O(n)$ vertices. For these cases, we get a linear time approximation scheme for the Steiner tree problem.

In [49], Klein presents an algorithm to find a subset spanner with weight at most $O(1/\epsilon)$ times the weight of the minimum Steiner tree of a given terminal set that preserves the distances between any two terminals up to a factor of $(1 + \frac{\epsilon}{2})$ (see Section 6.4). The algorithm runs in time $O(\frac{1}{\epsilon} \cdot n \log n)$ and can be used as the filtering step for a PTAS for subset TSP. Indeed, Klein derives a PTAS for subset TSP that runs in time $O(n \log n)$ [49]. At first glance, one might think that this subset spanner can also be used as a filter for the SMT problem. Property (ii) of the filtering step is now fulfilled. But unfortunately, this subset spanner violates property (i). The vertices appearing in the SMT of G_0 might not even be part of the spanner. Or even if they are, the distances between them are not necessarily preserved in the spanner. Unlike the subset TSP, where one can replace subpaths between consecutive terminals of the optimal tour of G_0 with approximate shortest paths of G, it is not sufficient for the SMT problem to preserve shortest paths. One needs to preserve an approximate Steiner tree for the whole set of terminals.

This brings us to the notion of *banyans* [73, 85](see Section 6.5). For graphs, one can define banyans as follows: given a terminal set R, a t-banyan is a subgraph of G_0 that contains a t-approximation of the Steiner minimum tree for all subsets of R and its weight is at most ρ_t times the weight of the Steiner minimum tree of R. Rao and Smith [73] define geometric banyans in terms of *all* subsets of R. However, for SMT problems it is sufficient that a banyan includes only an approximation of the SMT of the whole set R, not necessarily for every subset of R. This notion was used in [85]. An algorithm that could find a $(1+\frac{\epsilon}{2})$-banyan of G_0 and R (even with this more restricted notion of banyans) in time polynomial in n would immediately result in a PTAS for the SMT problem in planar graphs — as we will see in the following sections. The banyan algorithm of Rao and Smith for complete geometric graphs exploits many geometric properties of Steiner trees on the plane and is not easily amenable to planar graphs — not even to incomplete geometric graphs (as would be sufficient for the rectilinear and octilinear SMT problems with obstacles).

One train of thought to derive a banyan would be to try to find a subset of the vertices $V_0' \subseteq V_0$, such that the subgraph of G_0 induced by $V_0' \cup R$ includes a $(1+\epsilon)$-approximation of the SMT of R and the weight of its MST is at most some constant times the weight of the SMT of R. That is, one would filter out vertices first. Then one could apply Althöfer et al.'s spanner [2] (or alternatively, Klein's planar spanner [47]) to derive the desired filtering. A similar idea was used for geometric banyans by Zhao [85].

In the following, we assume that the subgraph G with properties (i) and (ii) above is somehow given and we show how to derive a PTAS for the SMT problem in planar graphs based on this assumption.

9.3 Step 2: Thinning

In the thinning step, one tries to eliminate some edges of G of total weight no more than $\frac{\epsilon}{2} \cdot OPT$, such that it is possible to find the optimal solution in the thinned graph in polynomial time. Specifically, we compress a number of edges of G to achieve this effect. Klein [47] proved the following lemma and applied it to $G^\star$ for his thinning algorithm. It is similar in nature to Baker's decomposition [7](see Section 8.1). Remember that the *radius* of an undirected graph is the minimum height of any rooted spanning tree of the graph.

Lemma 9.5 ([47]). *There is a linear time algorithm that, for any edge-weighted embedded planar graph X and integer k, returns an edge-set S of weight at most $\frac{1}{k} \cdot w(X)$ and an embedded planar graph Y of radius at most k such that $Y - S = X - S$ and $V(Y) = V(X) \cup \{s\}$, where s is a new auxiliary vertex.*

We can not use this lemma as it is for the Steiner tree problem. The problem is that we have to deal with terminal sets. We prove the following variation of the lemma that overcomes this difficulty.

Lemma 9.6. *Let an integer k, a connected edge-weighted embedded planar graph G, and a set of terminals $R \subseteq V(G)$ be given. There is a linear-time algorithm that returns an edge-set S of weight at most $\frac{1}{k} \cdot w(G)$, an embedded planar graph H, and a set $R_H \subseteq V(H)$ such that:*

(i) $H/S = G/S$;

(ii) $R_H/S = R/S$;

(iii) every connected component of $H^\star$ has radius at most k.

Proof. Since G is connected, we know that $G^\star$ is connected. Also, since G and $G^\star$ share the same set of edges, we know that $w(G) = w(G^\star)$. We apply the following procedure to $G^\star$:

Carry out breadth-first-search from some node r of the graph and label each vertex according to its distance from r (in breadth-first-search, distance is measured in the number of edges, not the weight). For $i = 0, 1, \dots, k-1$, let S_i denote the set of edges with one endpoint labeled $d_1 \equiv i(\mod k)$ and $d_2 = d_1 - 1$. This gives us a partition of all the edges of $G^\star$ into k sets. So, there exists at least one index i, such that the weight of S_i is at most $(1/k)w(G^\star)$. Let t be such an index. We have that $w(S_t) \leq (1/k)w(G)$.

We can regard S_t as a set of cuts in $G^\star$, separating vertices with labels $jk + t - 1$ from vertices with labels $jk + t$ for $j = 0, 1, 2, \dots$. Now, if we remove S_t from $G^\star$, the graph falls apart into a number of connected components $\hat{G}_0^\star, \hat{G}_1^\star, \dots, \hat{G}_z^\star$. The graph $\hat{G}_j^\star$ contains the vertices of $G^\star$ with labels $\geq (j-1)k + t$ and $< jk + t$ for $j = 0, 1, \dots, z$. Let $\hat{Y}_j$ be the graph obtained from $\hat{G}_j^\star$ by adding a supernode s_j to it and edges as follows: for every edge e in S_t that connects some vertex labeled $(j-1)k + t - 1$ to a vertex v of $\hat{G}_j^\star$ with label $(j-1)k + t$, add an edge e' from s_j to v and identify e' with e. Notice that the breadth-first-search tree of Y_j rooted at s_j has depth at most k and so, Y_j has radius at most k for all $j = 0, 1, \dots, z$. Let Y be the union of these $\hat{Y}_j$ for $j = 0, 1, \dots, z$. We define $H := Y^\star$ and let $S := S_t$.

Notice that since we identified the edges e and e' above, we have that G and H are defined on the same set of edges. We have that $H^\star - S = G^\star - S$ and hence, $H/S = G/S$. Also, since every Y_j has radius at most k, we get that property (iii) of the lemma is also fulfilled. It remains to specify a set $R_H \subseteq V(H)$ such that $R_H/S = R/S$.

Let $\hat{G}_j$ be the dual of some connected component $\hat{G}_j^\star$ of $G^\star - S$. So, $\hat{G}_j$ is a connected component of G/S. Let v be the vertex of $\hat{G}_j$ that corresponds to the face of $\hat{G}_j^\star$ in which s_j will be added. Adding s_j to $\hat{G}_j^\star$ and connecting it to a number of vertices of $\hat{G}_j^\star$ using edges of S to obtain the graph $\hat{Y}_j$ has the effect of first adding a loop to v and then splitting it into a number of vertices $v_1, \dots, v_q$ that are connected by a cycle of zero-weight edges. Hence $V(\hat{Y}_j^\star) = (V(\hat{G}_j) \setminus \{v\}) \cup \{v_1, \dots, v_q\}$. Initialize R_j with $R \cap V(\hat{Y}_j^\star)$. Add the vertices $\{v_1, \dots, v_q\}$ to R_j if and only if $v \in R$. We obtain R_H as the disjoint union of the R_js over all $j = 0, 1, \dots, z$. Now one can check that $R_j/S = (R/S) \cap V(\hat{G}_j)$. Hence, $R_H/S = R/S$.

Note that the only step of this algorithm where it is not clear if it can be done in linear time is the construction of G/S and R/S, since when we compress a loop, we have to check whether both of the new connected components include terminals or not. But because of the special structure of S, this can be checked using breadth-first-search once at the beginning of the compression process. We leave the details of this linear-time implementation to the reader. $\square$

9.4 Step 3: Dynamic Programming

Theorem 9.7. *There is an algorithm that, given a connected edge-weighted n-node planar embedded graph H such that $H^\star$ has radius k and a set of terminals $R \subseteq V(H)$, finds the Steiner minimum tree of R in H in time $O(n \cdot k^{O(k)})$.*

Our proof closely follows the ideas of the proof given in [47] for the traveling salesman problem. We adapt some of those ideas and add some new concepts to derive the proof of our theorem about the Steiner tree problem.

As a first step of our algorithm, we show that we can reduce the problem to the case in which the degree of the input graph is bounded by three. Every vertex of H that has degree $d > 3$, can be split into two new vertices that each have degree $< d$ and are connected by a new edge of weight zero. If the vertex that was split was a terminal, let both of its split copies be terminals. Let L be the graph obtained from H by applying this procedure until every vertex has degree at most 3, let A be the set of artificial zero-weight edges added, and let R_L be the resulting terminal set. Note that $H = L/A$ and $R = R_L/A$. Let T_L be an optimal Steiner tree of R_L in L.

Lemma 9.8. *T_L/A is an optimal Steiner tree of R in H.*

Proof. We have that $H = L/A$ and $R = R_L/A$. By Lemma 9.3, we get that T_L/A is a Steiner tree of R in H. Furthermore,

$$w(T/A) = w(T) = OPT(L, R_L) = OPT(L/A, R_L/A) = OPT(H, R) \ .$$

The first equality results from the fact that A is zero-weight, the third one is given by Lemma 9.4. $\square$

The outline of the dynamic programming is as follows. Let $T^\star$ be a spanning tree of $H^\star$ of height at most k. Since $L^\star$ is obtained from $H^\star$ by adding a number of edges (the splitting of vertices above corresponds to adding edges to the dual), we have that $T^\star$ is also a spanning tree of $L^\star$. Let T be the set of edges of $L^\star$ not in $T^\star$. By Proposition 6.6, we know that T is a spanning tree of L. Let $r \in R_L$ be an arbitrary terminal vertex. Add an artificial vertex r_0 to T and connect it with an edge to r (if the degree of r becomes more than 3, split r; note that these modifications do not affect $T^\star$). Root T at r_0. The dynamic programming will process T in a bottom-up manner and construct a table for each edge $e \in T$. The value of $OPT(L, R_L)$ will be computed from the table of the edge connecting r_0 to r. Once the value of the optimal solution is known, the Steiner tree itself can be constructed in a top-down fashion as is common in dynamic programming. This post-processing is straightforward and we do not describe it here.

For an edge $e \in T$, let v_e denote the child-endpoint of e and let D_e denote the descendents of v_e (including v_e). By Proposition 6.8, the elementary cycle of e in $L^\star$ with respect to $T^\star$ consists of the edges $\Gamma_L(D_e)$, i.e. the cut that disconnects D_e from the rest of the graph. Call this cut/cycle C_e. Define the *boundary vertices of* D_e to be those vertices of D_e that are adjacent to some edge of C_e and denote them by B_e. Since $T^\star$ has height k, it follows that any path in $T^\star$ has length at most $2k$ and hence, any elementary cycle of $T^\star$ has length at most $2k+1$. So, we have that $|B_e| \leq |C_e| \leq 2k+1$.

For an edge $e \in T$, let a *configuration* K of e be given by

(i) a selection of the boundary vertices, $S_K \subseteq B_e$, such that every boundary vertex that is also a terminal is included in S_K;

(ii) a partitioning P_K of S_K into at most $|S_K|$ sets.

For a vertex $v \in S_K$, let $p_K(v) \in \{1, \ldots, |S_K|\}$ denote the partition it belongs to. A *solution* for an edge e and a configuration K is defined as a set of trees in the graph spanned by D_e, such that

(i) every terminal in D_e is included in exactly one tree;

(ii) every tree includes at least one vertex of S_K;

(iii) all vertices of S_K that belong to the same partition in P_K are part of the same tree;

(iv) no two vertices of S_K that belong to different partitions in P_K are part of the same tree.

Let $\min(e, K)$ denote the weight of a solution for e and K with minimum total edge-weight. For the special case when S_K is empty, let $\min(e, K)$ be 0 if D_e contains no terminal and ∞ otherwise. We would like to find the value of $\min(e, K)$ for every possible choice of e and K. Note that there are $O(n \cdot k^{O(k)})$ such possibilites. Once we have all these values, $OPT(L, R_L)$ can be easily computed: let $\hat{e}$ be the edge connecting r_0 to r. Let $\hat{K}$ be the configuration with $S_K = \{r\}$ and $P_K = \{S_K\}$. Then $OPT(L, R_L) = \min(\hat{e}, \hat{K})$ (remember that r has to be included in the tree, since it is a terminal).

Now to the computation of the table. If v_e is a leaf, then $\min(e, K)$ is ∞ if v_e is a terminal and is not included in S_K; otherwise it is 0. Let v_e be some internal node of T and K a given configuration of it. Denote the set $\{v_e\}$ by D_0. Let K_0 be a configuration for D_0 (there are exactly two possibilites for K_0: including v_e in S_{K_0} or not). Let $C_0 = \{e_1, \ldots, e_s\}$ be the child edges of e ($s \leq 2$), $D_i = D_{e_i}$ and $C_i = C_{e_i}$ for $i = 1, \ldots, s$. Let K_1 be a configuration of e_1 and if $s = 2$, let K_2 be a configuration of e_2, otherwise let K_2 be empty. We say the configurations K_0, K_1, and K_2 are *consistent* with K if

(i) $(S_{K_0} \cup S_{K_1} \cup S_{K_2}) \cap B_e = S_K$;

(ii) for $i = 1, \ldots, s$ and $u, v \in S_{K_i} \cap S_K$, we have that $p_K(u) = p_K(v)$ if and only if $p_{K_i}(u) = p_{K_i}(v)$, i.e. u and v belong to the same partition in P_K if and only if they also belong to the same partition in P_{K_i};

(iii) for $i = 1, \ldots, s$, any set in P_{K_i} includes at least one vertex of S_K.

By condition (iii), we know that for any vertex $v \in S_{K_0} \cup S_{K_1} \cup S_{K_2}$, there is a vertex $u \in S_K$, such that v and u are in the same partition of P_K and by condition (ii), this partition does not depend on the choice of u. Let $\hat{p}(v) := p_K(u)$. Let $\hat{E}$ be the union of $C_0, \ldots, C_s$ restricted to $D_0 \cup \cdots \cup D_s$, i.e. $\hat{E}$ is the set of edges that run between D_0, D_1 and D_2 (if existent). Note that $|\hat{E}| = O(k)$. We say that a selection of edges $\hat{E}'$ is consistent with the consistent configurations K, K_0, K_1, and K_2 if for all edges $e = \{u, v\} \in \hat{E}'$ we have

(i) $u, v \in S_{K_0} \cup S_{K_1} \cup S_{K_s}$;

(ii) $\hat{p}(u) = \hat{p}(v)$;

(iii) if for some $i, j \in \{0, 1, 2\}$, there are vertices $u \in D_i$ and $v \in D_j$, such that $\hat{p}(u) = \hat{p}(v)$, then there exists an edge $\{x, y\} \in \hat{E}'$, such that $\hat{p}(x) = \hat{p}(y) = \hat{p}(u)$.

Property (iii) ensures that all the vertices that belong to the same partition of P_K will end up in the same tree. Now, one can compute $\min(e, K)$ by enumerating over all consistent configurations $K_0, \ldots, K_s$ and consistens edge selections $\hat{E}'$ and take the one with minimum total edge weight. Our algorithm runs in time $O(n \cdot k^{O(k)})$.

9.5 Summary of the Algorithm

All in all, we have the following algorithm to find a $(1+\epsilon)$-approximation of the Steiner minimum tree of a set of terminals R in an embedded planar graph G_0:

Step 1 (Filtering): Apply a (not-yet-existing) polynomial-time filtering algorithm to find a subgraph G of G_0 that has total weight at most ρ_ϵ times OPT and that contains a $(1+\frac{\epsilon}{2})$-approximation of the SMT of R in G_0.

Step 2 (Thinning): Let $k = \lceil \frac{2}{\epsilon} \cdot \rho_\epsilon \rceil$ and apply the thinning algorithm of Lemma 9.6 to G to obtain an edge set S of weight at most $(1/k) \cdot w(G)$, a graph H and a set of terminals $R_H \subseteq V(H)$ with the given properties of the lemma.

Step 3 (Dynamic Programming): Let H' be the graph obtained from H by setting the weight of all edges in S to zero. For every connected component H'_j of H', apply the dynamic programming of Theorem 9.7 to H'_j and $R'_j := R_H \cap V(H'_j)$ to obtain the optimal Steiner multitree $T_{H'}$ of H'. By Lemma 9.4, we know that $T_{H'}/S$ is the optimal Steiner multitree of $H'/S = H/S = G/S$.

Step 4 (Lifting): Interprete $T_{H'}/S$ as a Steiner multitree of R/S in G/S and obtain a Steiner tree T of R in G by uncompressing the edges of S as described in the proof of Lemma 9.4.

By Lemma 9.4 we know that if we consider S as having zero weight, then T is indeed an optimal Steiner tree of R in G. Adding the real weight of S back to the graph, adds at most

$$\frac{1}{k} \cdot w(G) \leq \frac{\rho_\epsilon}{k} \cdot OPT \leq \frac{\epsilon}{2} \cdot OPT$$

to the total weight. Hence, the algorithm delivers the promised approximation ratio in time $O(n \cdot k^{O(k)})$ plus the time needed for filtering. If $\rho_\epsilon = O(1/\epsilon)$ and the filtering algorithm also runs in linear time, this is linear in n. If $\rho_\epsilon = O(\log n)$ then we would at least get a QPTAS for this problem. We think that investigating the existance of a filtering algorithm for this problem is a promising direction for future research.

Bibliography

[1] R. K. Ahuja, O. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1–3):75–102, 2002.

[2] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Computational Geometry*, 9(1):81–100, 1993.

[3] S. Arora. Polynomial time approximation schemes for the Euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45:753–782, 1998.

[4] S. Arora. Approximation schemes for $\mathcal{NP}$-hard geometric optimization problems: a survey. *Mathematical Programming*, 97(1–2):43–69, 2003.

[5] S. Arora, M. Grigni, D. Karger, P. Klein, and A. Woloszyn. A polynomial-time approximation scheme for weighted planar graph TSP. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 33–41, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[6] M. Ayob, P. Cowling, and G. Kendall. Optimisation of surface mount placement machines. In *Proceedings of IEEE International Conference on Industrial Technology*, pages 486–491, 2002.

[7] B. S. Baker. Approximation algorithms for $\mathcal{NP}$-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994.

[8] H.-J. Bandelt and A. Dress. Reconstructing the shape of a tree from observed dissimilarity data. *Adv. Appl. Math.*, 7(3):309–343, 1986.

[9] S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *ICALP '03: Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 384–396, Berlin, Germany, 2003. Springer-Verlag.

[10] M. Bern and P. Plassmann. The Steiner problem with edge lengths 1 and 2. *Inormation Processing Letters*, 32(4):171–176, 1989.

[11] D. Bienstock and C. L. Monma. On the complexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990.

[12] U. Brandes and D. Handke. $\mathcal{NP}$-completeness results for minimum planar spanners. *Discrete Mathematics & Theoretical Computer Science*, 3(1):1–10, 1998.

[13] M. Brazil. Steiner minimum trees in uniform orientation metrics. In I. D.-Z. Du and X. Cheng, editors, *Steiner trees in Industries*, pages 1–27. Kluwer Academic Publishers, 2001.

[14] M. Brazil, D. Thomas, and P. Winter. Minimum networks in uniform orientation metrics. *SIAM Journal on Computing*, 30(5):1579–1593, 2000.

[15] M. Brazil, D. A. Thomas, J. F. Weng, and M. Zachariasen. Canonical forms and algorithms for Steiner trees in uniform orientation metrics. *Algorithmica*, 44:281–300, 2006.

[16] M. Brazil, P. Winter, and M. Zachariasen. Flexibility of Steiner trees in uniform orientation metrics. *Networks*, 46:142–153, 2005.

[17] P. Brucker. *Scheduling Algorithms.* Springer-Verlag, 2004.

[18] L. Cai. $\mathcal{NP}$--completeness of minimum spanner problems. *Discrete Appl. Math.*, 48(2):187–194, 1994.

[19] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. In *SCG '92: Proceedings of the eighth annual symposium on Computational geometry*, pages 192–201, New York, NY, USA, 1992. ACM Press.

[20] H. Chen, C.-K. Cheng, A. B. Kahng, I. I. Mandoiu, Q. Wang, and B. Yao. The Y architecture for on-chip interconnect: analysis and methodology. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(4):588–599, 2005.

[21] H. Chen, C.-K. Cheng, A. B. Kahng, I. Măndoiu, and Q. Wang. Estimation of wirelength reduction for λ-geometry vs. Manhattan placement and routing. In *Proceedings of SLIP'03*, pages 71–76. ACM Press, 2003.

[22] D. R. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5(4):724–742, 1976.

[23] P. Chew. There is a planar graph almost as good as the complete graph. In *SCG '86: Proceedings of the second annual symposium on computational geometry*, pages 169–177, New York, NY, USA, 1986. ACM Press.

[24] E.-A. Choukhmane. Une heuristique pour le problème de l'arbre de Steiner. *Rech. Opèr.*, 12:207–212, 1978.

[25] Y. Crama, J. Klundert, and F. C. R. Spieksma. Production planning problems in printed circuit board assembly. *Discrete Applied Mathematics*, 123:339–361, 2002.

[26] E. D. Demaine and M. T. Hajiaghayi. Bidimensionality: new connections between FPT algorithms and PTASs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 590–601, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[27] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, pages 269–271, 1959.

[28] D. P. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete Comput. Geom.*, 5(4):399–407, 1990.

[29] M. Elkin and D. Peleg. The hardness of approximating spanner problems. In H. Reichel and S. Tison, editors, *STACS '00: Proceedings of the 17th annual Symposium on Theoretical Aspects of Computer Science*, volume 1770 of *Lecture Notes in Computer Science*, pages 370–381. Springer, 2000.

[30] D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 9, pages 425–461. Elsevier, 2000.

[31] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

[32] A. Frangioni, E. Necciari, and M. G. Scutellà. A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. *J. Comb. Optim.*, 8(2):195–220, 2004.

[33] J. L. Ganley and J. P. Cohoon. Routing a multi-terminal critical net: Steiner tree construction in the presence of obstacles. In *Proceedings of the International Symposium on Circuits and Systems*, pages 113–116, 1994.

[34] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32:835–859, 1977.

[35] M. Garey and D. Johnson. The rectilinear Steiner tree problem is $\mathcal{NP}$-complete. *SIAM Journal on Applied Mathematics*, 32:826–834, 1977.

[36] R. Gaudlitz. Optimization algorithms for complex mounting machines in PC board manufacturing. Diploma thesis, Department of Computer Science, Technische Universität Darmstadt, 2004.

[37] M. Grigni, E. Koutsoupias, and C. Papadimitriou. An approximation scheme for planar graph TSP. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 640–645, Washington, DC, USA, 1995. IEEE Computer Society.

[38] M. Grunow, H.-O. Günther, and M. Schleusener. Component allocation for printed circuit board assembly using modular placement machines. *International Journal of Production Research*, 41:1311–1331, 2003.

[39] M. T. Hajiaghayi. The bidimensionality theory and its algorithmic applications. *PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA*, 2005.

[40] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14:255–265, 1966.

[41] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *STOC '94: Proceedings of the twenty-sixth annual ACM Symposium on Theory of Computing*, pages 27–37, New York, NY, USA, 1994. ACM Press.

[42] F. Hwang. On Steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics*, 30:104–114, 1976.

[43] F. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53. Annals of Discrete Mathematics, North-Holland, 1992.

[44] A. Kahng, I. Măndoiu, and A. Zelikovsky. Highly scalable algorithms for rectilinear and octilinear Steiner trees. *Proceedings 2003 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 827–833, 2003.

[45] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[46] S. Khanna and R. Motwani. Towards a syntactic characterization of PTAS. In *STOC '96: Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing*, pages 329–337, New York, NY, USA, 1996. ACM Press.

[47] P. N. Klein. A linear-time approximation scheme for TSP for planar weighted graphs. In *Proceedings, 46th IEEE Symposium on Foundations of Computer Science*, pages 146–155, 2005.

[48] P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[49] P. N. Klein. A subset spanner for planar graphs, with application to subset TSP. In *Proceedings, 38th ACM Symposium on Theory of Computing*, pages 749–756, 2006.

[50] C. Koh. Steiner problem in octilinear routing model. *Master thesis, National University of Singapore*, 1995.

[51] C.-K. Koh and P. H. Madden. Manhattan or non-Manhattan?: a study of alternative VLSI routing architectures. In M. Sarrafzadeh, P. Banerjee, and K. Roy, editors, *ACM Great Lakes Symposium on VLSI*, pages 47–52. ACM, 2000.

[52] D. Lee and C.-F. Shen. The Steiner minimal tree problem in the λ-geometry plane. In *Proceedings 7th International Symposium on Algorithms and Computations (ISAAC 1996)*, volume 1178 of *Lecture Notes in Computer Science*, pages 247–255. Springer, 1996.

[53] C. Levcopoulos and A. Lingas. There are planar graphs almost as good as the complete graphs and almost as cheap as minimum spanning trees. *Algorithmica*, 8(3):251–256, 1992.

[54] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[55] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[56] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[57] J. Liu, Y. Zhao, E. Shragowitz, and G. Karypis. A polynomial time approximation scheme for rectilinear Steiner minimum tree construction in the presence of obstacles. In *9th IEEE International Conference on Electronics, Circuits and Systems*, volume 2, pages 781–784, 2002.

[58] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[59] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: a simple new method for the geometric k-MST problem. In *SODA '96: Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithms*, pages 402–408, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[60] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on Computing*, 28(4):1298–1309, 1999.

[61] B. Mohar and C. Thomassen. *Graphs on Surfaces.* The John Hopkins University Press, 2001.

[62] M. Müller-Hannemann and S. Peyer. Approximation of rectilinear Steiner trees with length restrictions on obstacles. In *8th Workshop on Algorithms and Data Structures (WADS 2003)*, volume 2748 of *Lecture Notes in Computer Science*, pages 207–218. Springer, 2003.

[63] M. Müller-Hannemann and A. Schulze. Hardness and approximation of octilinear Steiner trees. In *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC 2005), Hainan, China*, volume 3827 of *Lecture Notes in Computer Science*, pages 256–265. Springer, 2005.

[64] M. Müller-Hannemann and A. Schulze. Approximation of octilinear Steiner trees constrained by hard and soft obstacles. In L. Arge and R. Freivalds, editors, *SWAT*, volume 4059 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2006.

[65] M. Müller-Hannemann and K. Weihe. Moving policies in cyclic assembly-line scheduling. *Theoretical Computer Science*, 351:425–436, 2006. An extended abstract appeared in Proceedings of the International Workshop on Parameterized and Exact Computation (IWPEC'04) Lecture Notes in Computer Science 3162, pp. 149–161, Springer-Verlag.

[66] B. Nielsen, P. Winter, and M. Zachariasen. An exact algorithm for the uniformly-oriented Steiner tree problem. In *10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 760–772. Springer, 2002.

[67] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[68] D. Peleg and A. Schaeffer. Graph spanners. *Journal on Graph Theory*, 13:99–116, 1989.

[69] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 77–85, New York, NY, USA, 1987. ACM Press.

[70] J. Plesnik. A bound for the Steiner problem in graphs. *Math. Slovaca*, 31:155–163, 1981.

[71] H. Prömel and A. Steger. *The Steiner Tree Problem: A Tour through Graphs, Algorithms, and Complexity.* Advanced Lectures in Mathematics, Vieweg, 2002.

[72] J. S. Provan. An approximation scheme for finding steiner trees with obstacles. *SIAM Journal on Computing*, 17(5):920–934, 1988.

[73] S. B. Rao and W. D. Smith. Approximating geometrical graphs via "spanners" and "banyans". In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 540–550, New York, NY, USA, 1998. ACM Press.

[74] J. Remy and A. Steger. Approximation schemes for node-weighted geometric steiner tree problems. In C. Chekuri, K. Jansen, J. D. P. Rolim, and L. Trevisan, editors, *APPROX-RANDOM*, volume 3624 of *Lecture Notes in Computer Science*, pages 221–232. Springer, 2005.

[75] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 770–779, 2000.

[76] L. Roditty, M. Thorup, and U. Zwick. Deterministic constructions of approximate distance oracles and spanners. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 2005.

[77] A. Scholl. *Balancing and Sequencing of Assembly Lines.* Physica-Verlag, Heidelberg, 2nd edition, 1999.

[78] R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 813–822, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[79] S. Tazari, M. Müller-Hannemann, and K. Weihe. Workload balancing in multi-stage production processes. In C. Àlvarez and M. J. Serna, editors, *WEA*, volume 4007 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2006.

[80] S. L. Teig. The X architecture: not your father's diagonal wiring. In *SLIP '02: Proceedings of the 2002 international workshop on System-level interconnect prediction*, pages 33–37. ACM Press, 2002.

[81] M. Thimm. On the approximability of the Steiner tree problem. *Theoretical Computer Science*, 1-3:387–402, 2003.

[82] G. Venkatesan, U. Rotics, M. S. Madanlal, J. A. Makowsky, and C. P. Rangan. Restrictions of minimum spanner problems. *Information and Computation*, 136(2):143–164, 1997.

[83] W. Wang, P. C. Nelson, and T. M. Tirpak. Optimization of high-speed multistation SMT placement machines using evolutionary algorithms. *IEEE Transactions on Electronic Packaging and Manufacturing*, 22:137–146, 1999.

[84] M. Zachariasen. A catalog of Hanan grid problems. *Networks*, 38:76–83, 2001.

[85] H. Zhao. Algorithms and complexity analyses for some combinatorial optimization problems. *PhD Thesis, New Jersey Institute of Technology, NJ, USA*, 2005.

[86] Q. Zhu, H. Zhou, T. Jing, X. Hong, and Y. Yang. Efficient octilinear Steiner tree construction based on spanning graphs. *Proceedings 2004 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 687–690, 2004.