FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Solver for the Steiner Tree Problem with few Terminals

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Softwareengineering und Internet Computing

eingereicht von

André Schidler, BSc
Matrikelnummer 1225113

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Stefan Woltran
Zweitbetreuung: Dr. Johannes Fichte
Mitwirkung: DI Markus Hecher

Wien, 11. Oktober 2018

André Schidler

Stefan Woltran

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Solver for the Steiner Tree Problem with few Terminals

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Softwareengineering and Internet Computing

by

André Schidler, BSc
Registration Number 1225113

to the Faculty of Informatics

at the TU Wien

Advisor:	Prof. Stefan Woltran
Second advisor:	Dr. Johannes Fichte
Assistance:	DI Markus Hecher

Vienna, 11th October, 2018

André Schidler

Stefan Woltran

Erklärung zur Verfassung der Arbeit

André Schidler, BSc
Murlingengasse 15/319, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Oktober 2018

André Schidler

Kurzfassung

Das graphentheoretische Steinerbaumproblem ist ein sehr bekanntes $\mathcal{NP}$-hartes Problem. Instanzen des Problems bestehen aus einem Graphen $G = (V, E)$, den Kosten für Kanten c und einer Menge aus Terminalen $R \subseteq V$. Die Lösung zu einer Instanz ist ein zusammenhängender Teilgraph minimalen Gewichts der alle Terminale enthält. Es ist lange bekannt, dass Instanzen mit einer Laufzeitbeschränkung von $\mathcal{O}(3^{|R|})$ gelöst werden können. Das Problem ist also effizient lösbar, wenn die Anzahl der Terminale beschränkt werden kann.

PACE ist ein alljährlicher Wettbewerb. Jedes Jahr wird ein anderes $\mathcal{NP}$-hartes Problem verwendet, von dem bekannt ist, dass es effizient lösbar ist, wenn ein Parameter beschränkt werden kann. Die TeilnehmerInnen geben für diesen Wettbewerb Programme ab, die Instanzen des Problems lösen können (Lösungsprogramme). Diese werden danach gereiht, wie viele Instanzen sie innherhalb einer gewissen Zeit- und Arbeitsspeichergrenze lösen können. Dieses Jahr war das ausgeschriebene Problem das graphentheoretische Steinerbaumproblem.

Wir haben ein Lösungsprogramm implementiert und damit den vierten Platz erzielt. In dieser Arbeit präsentieren wir den formaltheoretischen Hintergrund sowie Implementierungsdetails unseres Programms. Der Großteil der Implementierung folgt direkt aus der Theorie. Daher beschränken wir uns auf jene Implementierungsdetails die entweder spezifisch für unser Programm sind, oder nicht direkt aus der Theorie ableitbar sind.

Der Quellcode aller teilnehmenden Programme muss nach der Abgabefrist veröffentlicht werden. Dies hat uns erlaubt die drei besten Programme näher zu analysieren. Wir präsentieren hier die wesentlichen Unterschiede zwischen diesen Programmen und unserem. Wir haben außerdem versucht, das durch die Analysen gesammelte Wissen, in unser Programm einzubauen. Wie diese Adaptierungen unser Programm beeinflusst haben, ist ein weiteres Thema dieser Arbeit.

Wir haben unser Programm außerdem mit öffentlichen und bekannten Instanzen getestet. Wir präsentieren die Ergebnisse und vergleichen diese mit den von anderen aktuellen Lösungsprogrammen erzielten Resultaten.

Abstract

The Steiner tree problem on graphs is a well known $\mathcal{NP}$-hard problem. Instances of this problem define a graph $G = (V, E)$ with edge costs and a set of terminals $R \subseteq V$. The solution to an instance is a connected subgraph of minimal total cost, that contains all vertices in R. It is known that an instance can be solved in time $\mathcal{O}(3^{|R|})$. The problem is therefore (fixed parameter) tractable if we know that the number of terminals is smaller than some k.

PACE is an annual competition. Each year a different fixed parameter tractable problem is used. Participants submit programs that solve instances of the problem. These solvers are then ranked by the number of instances they could solve within a given time and memory limit. This year's topic was the Steiner tree problem on graphs.

We implemented and submitted a solver that placed 4th in the competition. In this thesis we discuss the formal theory used in our solver, as well as implementation details. Since most of the implementation follows directly from the theory, we focus on details that are particular to our implementation or not apparent from theory.

PACE requires the source code of all submitted solvers to be publicly available. This allowed us to analyze the top three submissions. We discuss how they differ from our implementation. We also tried to apply the insights from this analysis to our solver. We show if and how these extensions impact the performance of our implementation.

We benchmarked our solver against a collection of publicly available and well known instances. We present the results and discuss how these compare to those of other state-of-the-art solvers.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The *Steiner Tree Problem on Graphs (ST)* is named after the 19th century mathematician Jakob Steiner. It is well known that the problem is $\mathcal{NP}$-hard [Kar72] and therefore intractable. Since we cannot generally solve instances of $\mathcal{NP}$-hard problems efficiently, knowing which classes of instances are practically solvable is of great value.

One approach to identify such classes is *parameterized complexity*. The idea is to find parameters besides the input size to classify the complexity. We can then identify those classes of instances that become tractable, if the parameter is bounded.

The Parameterized Algorithms and Computational Experiments Challenge (PACE) is an annual competition that encourages the implementation of solvers for such problems. Often new approaches to problems are developed only theoretically and their practical value is unknown. PACE [BS18] tries to encourage participants to bridge this gap between theory and practice.

The main motivation was to put knowledge from the parameterized complexity domain into practice. The choice of ST was mandated by this year's topic of PACE.

1.2 Problem Statement and Aim

An instance of ST is defined by a graph, edge costs and a subset of its vertices called the terminals. The goal is to find a connected subgraph of minimum weight containing all terminals. Figure 1.1 shows an example graph and the solution to the associated ST. The graph has 9 terminals and the edges of the minimal subgraph have a total weight of 82. Although the problem is intractable, one of the classic algorithms for this problem can efficiently solve instances with a low number of terminals. The number of terminals is therefore a parameter that allows us to identify efficiently solvable instances.

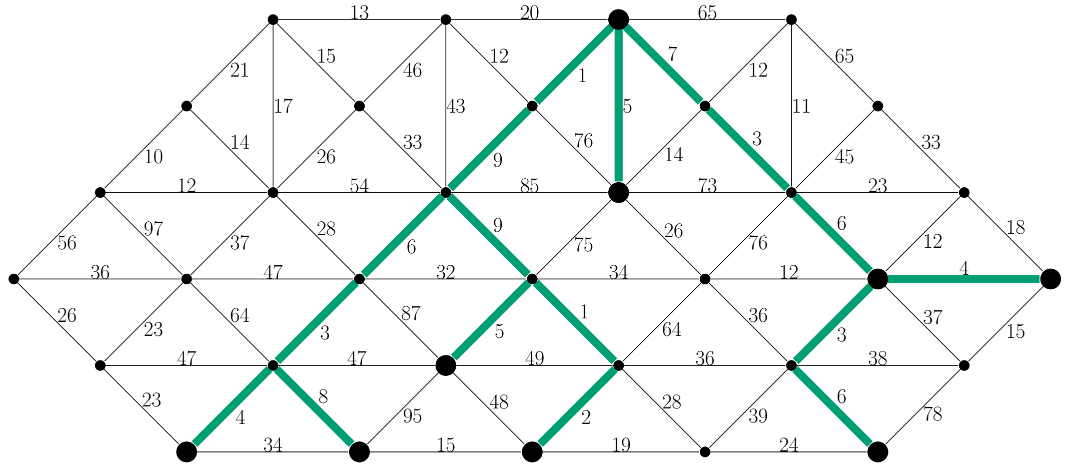


Figure 1.1: A graph with a minimal subgraph connecting the terminals. Terminals are drawn larger and edges of the subgraph are colored.

This year's PACE had several tracks:

1. Exact with few terminals: Instances contain few terminals.
2. Exact with low treewidth: Instances have low treewidth, a structural property that can be used by algorithms.
3. Heuristic: In contrast to the other two tracks, the solution found by the solver is not required to be optimal.

We participated in the *exact with few terminals* track. The rules for this track stated that every instance has to be solved optimally or not at all. It was therefore not allowed to output a heuristic solution, in the hope that it is correct. Furthermore, use of third party solvers was prohibited. It would have otherwise been possible to base the implementation on existing linear programming frameworks. In every track the submitted solvers had 30 minutes per instance to find a solution and were required to stay within a 6 GB memory limit.

The general objective was to create a solver that is specialized for instances with few terminals. For this end, we set the following goals:

- Combine the state-of-the-art body of theory into one program.
- Optimize the solver to perform well in the competition. This required the following considerations:
  - The runtime does not matter as long as it is within the limits. Solving hard instances in time is therefore more important than solving easier instances fast.
  - Memory has to be used wisely, as the limit is tight.
  - The instances used to rank the submissions will probably resemble the publicized instances. It is therefore important that our solver performs well for the public instances.

- Analyze the other submissions after the deadline.
- Compare our solver to existing solvers using well known benchmarks.
- Prove the correctness of our approach.

1.3 Contributions

Our two main contributions are the proof of correctness for our approach and the actual implementation.

Most of the theoretical foundation for our solver has been available and proven. We adapted the solving algorithm to allow for linear programming based, or more generally inconsistent, heuristics. This mandated a proof of correctness for the algorithm, given this new type of heuristics. This proof is presented in Chapter 3.7.

The implementation is a novel combination of techniques. Especially the solving algorithm has not been combined with other approaches before. Furthermore, we extended the implementation of the algorithm with details not yet discussed in literature. These extensions are discussed in Chapter 4.

Another contribution is the analysis and comparison of the PACE submissions as presented in Chapter 5.

1.4 Structure

We start by introducing some definitions and basic knowledge in Chapter 2. We then discuss our implementation in two ways. In Chapter 3 we present the theoretical body that went into our solver. The ideas presented here have been gathered from different sources and comprise the base our solver is built on. We also prove the correctness for our solving algorithm. In Chapter 4 we discuss implementation details. Here we present new ideas that went in our solver as well as insights we gathered. Afterwards we discuss the results of the PACE competition and take a close look at the other submissions in Chapter 5. We also discuss attempts to extend our solver with features from the other submissions. In Chapter 6 we present the results of benchmarking our solver using well known instances. We also compare it to the results in other papers.

1.5 State of the Art

One successful approach to solving the Steiner tree problem is dynamic programming. The *Dreyfus-Wagner algorithm* [DW72] is one of the classic algorithms for this problem. A refinement of this approach, the *Dijkstra-Steiner algorithm* [HSV16], is used in our solver. Both are discussed in Chapter 3.2.

Another dynamic programming algorithm with similar properties is a minimum-flow algorithm for networks [EMV87]. It is used by one of the other submissions and is discussed in Chapter 5.2.1.

A different approach used to solve the problem is *Branch and Cut (B&C)*. The idea is to use *linear programming* to model the problem and use a modified *Branch and Bound (B&B)* algorithm to find a solution. Solvers that use B&C distinguish themselves by the linear programming model they use [FLL$^+$17, KM96]. As one of the other submissions uses this approach, we discuss it in Chapter 5.4.

Solvers usually try to reduce an instance before they solve it. Therefore, they try to remove irrelevant parts of the graph, before passing it to the solving algorithm. As most reductions have been known for some time, state of the art solvers use a similar set of reductions. We discuss these in Chapter 3.3.

Some algorithms with theoretically good bounds have been developed, that do not seem to perform well in practice. A subset convolution algorithm has the currently best known runtime bound [BHKK07]. Furthermore, algorithms with worse time bounds, but polynomial memory bounds have been developed [BHKK07, FGK08].

CHAPTER 2

Preliminaries

In this chapter, we establish the notations and background knowledge required throughout this thesis. We start by introducing some *complexity theory* basics. We then discuss the necessary *graph theory* definitions. This allows us to define the *Steiner Tree Problem on Graphs (ST).* Finally, we introduce *linear programming.*

2.1 Complexity Theory

We use $\mathcal{O}(f(n))$ to classify the runtime of algorithms. Given a measurement n for the size of the input, e.g. the number of vertices in a graph or the number of elements to sort, we define $\mathcal{O}$ as follows [Bac94, Lan09]:

$$\mathcal{O}(g(n)) = \{f(n) \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : 0 \leq f(n) \leq c \cdot g(n)\}$$

We can divide problems into two basic groups. *Tractable* and *intractable* problems. A tractable problem is a problem that can be solved by an algorithm in time $\mathcal{O}(f(n))$, where $f(n)$ is a polynomial [Pap95]. We also say the problem is solvable in *polynomial time.*

An important class of problems are $\mathcal{NP}$-complete problems. These are problems that can only be solved in polynomial time by non-deterministic algorithms. For an exact definition we refer the reader to the literature [Pap95]. We limit this discussion to some important properties. It is an open problem if $\mathcal{NP}$-complete problems are tractable. As of now no deterministic polynomial-time algorithms are known that solve such problems. $\mathcal{NP}$-hard problems are at least as hard to solve as $\mathcal{NP}$-complete problems.

As we cannot generally solve intractable problems efficiently, it is of interest to identify tractable classes, whose instances can be solved in polynomial time. *Parameterized complexity* is one approach for this. Given a parameter k of the problem, we can define the complexity of the problem under the assumption that the parameter is constant. This gives us a bound on those classes, where we know a bound for k. Of particular importance for this thesis is the class of problems that are *Fixed Parameter Tractable ($\mathcal{FPT}$)*. Let I denote an instance, k_I the value of the parameter k for the instance and $|I|$ the size of the instance. A problem with parameter k is in $\mathcal{FPT}$ if there exists an algorithm, that solves every instance I in time $\mathcal{O}(f(k_I)|I|^c)$ for a computable function f and a constant c [CFK+15].

2.2 Graphs

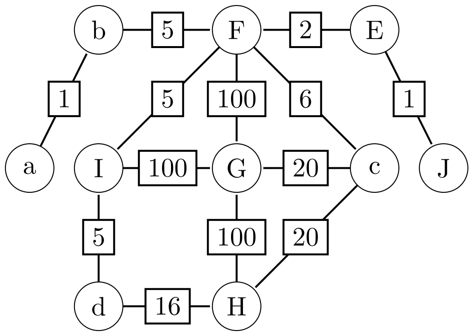


Figure 2.1: The basic graph of our running example

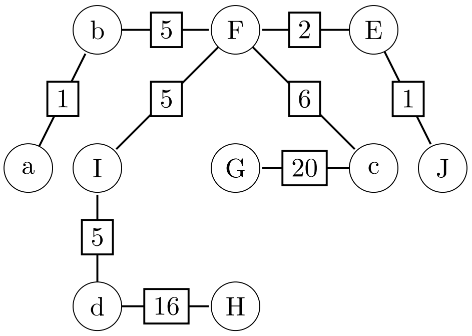


Figure 2.2: A minimum spanning tree for the graph in Figure 2.1

An *undirected graph* $G = (V, E)$ is defined by a set of vertices $V \neq \emptyset$ and edges $E \subseteq \{\{v_i, v_j\} \in V \times V \mid v_i \neq v_j\}$. We use v and v_i to denote vertices, where the subscript distinguishes different vertices. An edge $\{v_i, v_j\} \in E$, also denoted e_{ij}, connects vertices v_i and v_j. We use V and E whenever we refer to the current instance. In order to avoid ambiguity we use $V[G]$ and $E[G]$ to refer to the vertices and edges of a specific graph G. We also use $m = |E|$, $n = |V|$ as shorthand notations. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq (E \cap (V' \times V'))$.

A *network* is a graph with extra properties. For this thesis we use networks with edge costs $c : E \mapsto \mathbb{N}$ and denote them $G = (V, E, c)$. We also use the terms (undirected) graph and network synonymously. The weight of a graph is the sum of its edge weights denoted $|G| = \sum_{e \in E} c(e)$. Figure 2.1 shows the graph used throughout this thesis. Circles represent vertices, lines are edges and the edge costs are in rectangles.

We denote the set of edges incident to a vertex v_i as $\delta(v_i) = \{e_{ij} \mid e_{ij} \in E\}$. We use δ_G to refer to the incident edges in a specific graph. The degree of a vertex is $|\delta(v_i)|$. The neighborhood of a vertex v_i are the incident vertices or $\{v_j \mid e_{ij} \in \delta(v_i)\}$. A vertex induced subgraph for $V' \subset V$ is defined as $(V', E \cap (V' \times V'))$. An edge induced subgraph for $E' \subset E$ is defined as $(\bigcup_{e_{ij} \in E'} \{v_i, v_j\}, E')$. A complete graph is a graph $G = (V, \{e_{ij} \in V \times V \mid v_i \neq v_j\})$, i.e. a graph where every vertex is connected to all other vertices. The density of a graph is the ratio of edges to vertices. Sparse graphs have low density.

A *path* is a subgraph that can be written as an alternating sequence of vertices and edges $v_{i_0} e_{i_0 i_1} v_{i_1} e_{i_1 i_2} v_{i_2} ... e_{i_{\ell-1} i_\ell} v_{i_\ell}$, where each edge connects the adjacent vertices. The path *connects* v_{i_1} and v_{i_ℓ}. As a path S is a subgraph, we denote its vertices by $V[S]$ and its edges by $E[S]$. The length of a path is $\sum_{e_{ij} \in E[S]} c(e_{ij})$. The distance between two vertices is the length of the shortest path connecting the vertices and denoted by $d(v_i, v_j)$ [Reh15]. A *cycle* is a path connecting a vertex to itself using distinct edges. A *connected graph* is a graph where for every $v_i, v_j \in V$ such that $v_i \neq v_j$, there exists a path connecting v_i with v_j.

A connected component C of a graph G is a subgraph of G such that for every $v_i, v_j \in V[C], v_i \neq v_j$, there exists a path from v_i to v_j. Furthermore, there exists no $v_i \in V[C], v_j \in V[G] \setminus V[C]$ such that there is a path from v_i to v_j. A graph is therefore connected if it contains only one connected component.

A *tree* is a connected graph without cycles. A Minimum Spanning Tree (MST) of a graph G is a tree $M_G = (V, E', c)$ with $E' \subseteq E$, that minimizes $\sum_{e_{ij} \in E'} c(e_{ij})$. Figure 2.2 shows the MST for the graph in Figure 2.1.

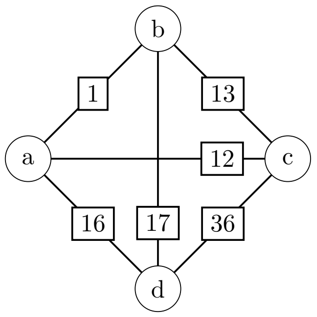


Figure 2.3: A distance network

The distance network for a graph G and a set W with $\emptyset \subset W \subseteq V$ is defined as $D_G(W) = (W, W \times W, d)$. It is therefore a complete graph for W with the shortest distances in G as edge weights. Figure 2.3 shows the distance network $D_G(\{a, b, c, d\})$ for our running example.

A *directed* graph $G = (V, A)$ is defined by its vertices and arcs $A \subseteq \{(v_i, v_j) \in V \times V \mid v_i \neq v_j\}$. We define the set of incoming edges for a vertex v_i as $\delta^-(v_i) = \{a_{ji} \in A\}$ and the set of outgoing edges as $\delta^+(v_i) = \{a_{ij} \in A\}$. Paths can be defined analogously.

2.3 Steiner Trees

We can now formally define the problem.

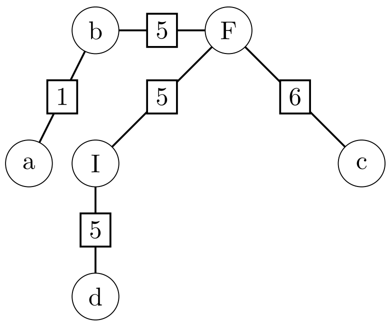


Figure 2.4: The Steiner minimal tree (SMT) of our running example

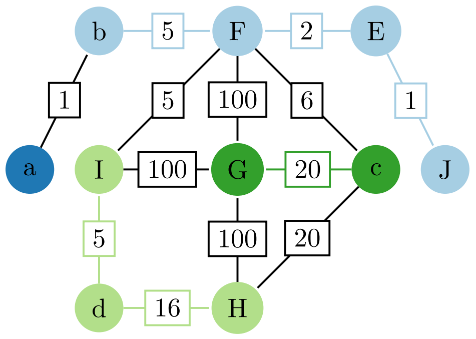


Figure 2.5: A Voronoi partitioning. Different colors indicate different regions

Problem 1 (Steiner Tree Problem on Graphs (ST))

Input *An instance $I = (G, R)$. With pairs (G, R) where $G = (V, E, c)$ is a graph and $R \subseteq V$ are terminals.*

Task *A Steiner tree is a connected subgraph $T = (V', E', c)$, with $R \subseteq V'$. Let $\mathcal{T}$ be the set of all possible Steiner trees for G. The task is to find $\arg\min_{T \in \mathcal{T}} |T|$.*

We refer to such a solution as a Steiner minimal tree (SMT) (sometimes the term minimal Steiner tree is used instead). Note that any SMT is a tree, as otherwise removing an edge from a cycle would reduce the total weight. We assume that every Steiner tree is a tree. The SMT for an instance is not necessarily unique. We define $smt_G(R)$ as the set of SMTs for an instance (G, R). $|smt_G(R)|$ is the (unique) weight of the SMTs. We use z_i to denote terminals, i.e. $z_i \in R$, and $k = |R|$. Steiner vertices are $v_i \in V \setminus R$. Our running example in Figure 2.1 uses lower case letters to denote terminals and upper case letters for Steiner vertices. Figure 2.4 shows the SMT for our running example.

An elementary path is a path that contains no intermediary terminals. Therefore, a path S from v_i to v_j is an elementary path if $\forall v_\ell \in V[S] : v_\ell \notin R \lor v_\ell = v_i \lor v_\ell = v_j$. The restricted distance $\underline{d}(v_i, v_j)$ is the length of the shortest elementary path connecting v_i and v_j [Reh15].

A Voronoi partitioning separates the vertices into k disjoint sets. For each terminal $z_i \in R$ we define the neighborhood $N(z_i)$ as the set of vertices that are closer to z_i than to any other terminal [Pol03]. Ties are assumed to be broken randomly. More precisely

$$V = \dot{\bigcup}_{z_i \in R} N(z_i)$$

$$v_i \in N(z_i) \Rightarrow d(v_j, z_i) \leq d(v_j, z_\ell) \text{ for all } z_\ell \in R$$

If $v_i \in N(z_i)$ then z_i is the *base* of v_i, therefore $base(v_i) = z_i$. $N(z_i)$ is called the Voronoi region of z_i. Voronoi regions can be computed in $\mathcal{O}(m + n \log n)$ [Meh88]. Figure 2.5 shows the Voronoi partitioning for our running example.

The Steiner tree problem is also defined for directed graphs:

Problem 2 (Steiner arborescence problem)
Input *An instance $I = (G, R, r)$. With triples (G, R, r), where $G = (V, A, c)$ is a directed graph, $R \subseteq V$ are terminals and $r \in R$ is a root.*
Task *A graph $T = (V', A')$ is a Steiner arborescence if $R \subseteq V' \subseteq V$, $A' \subseteq (A \cap (V' \times V'))$ and for every $v_i \in V' \setminus \{r\}$ there exists exactly one path from r to v_i. Let $\mathcal{T}^+$ be the set of all possible Steiner arborescences for (G, R). The task is to find* $\arg\min_{T \in \mathcal{T}^+} |T|$.

Any ST instance can be transformed into the directed version, with the same optimal value [HRW92a]. To transfer the problem from an undirected graph $G = (V, E, c)$ into a directed graph $\overrightarrow{G} = (V, A, \overrightarrow{c})$, we set A such that for each edge $e_{ij} \in E$, A contains arcs $(v_i, v_j), (v_j, v_i)$. Similarly $\overrightarrow{c}((v_i, v_j)) = c(\{v_i, v_j\})$. As a root we use a random terminal $r \in R$. The solution is a graph where every terminal is reachable from the root. We use R' to denote $R \setminus \{r\}$.

2.4 Linear Programming

Linear programming is a way to model optimization problems. A linear programming model consists of a linear *objective function*, that is either minimized or maximized, a set of variables $x = \{x_1, ..., x_\ell\}$ and a set of linear inequalities or *constraints*. The goal is to find a variable assignment that optimizes the objective function and complies with all constraints [Dan98]. We refer to such a model as a *Linear Program (LP)*.

We can choose a domain for the variables. If the domain is limited to integer values, we have an *integer linear programming* model. We refer to such a model as an *Integer Linear Program (ILP)*. Any ILP can be converted to a general LP by allowing real values instead of integers. We refer to this as the LP-relaxation of the ILP. Given a minimization problem, an optimal solution for the LP-relaxation is a lower bound for the ILP. This property is often used to compute bounds for ILPs [Dan98, Pap95].

We can formulate ST as an ILP. There are many different ways to formulate the problem [Pol03] we use the *cut formulation* stated below [Ane80, Reh15].

$$\begin{aligned}
\text{minimize:} \quad & \sum_{e \in E} c(e) x_e & \\
\text{subject to:} \quad & \sum_{\substack{e_{ij} \in E \\ v_i \in W, v_j \notin W}} x_{e_{ij}} \geq 1, & \text{for all } W \subset V, 0 < |W \cap R| < |R| \\
& x_e \in \{0, 1\} & \text{for all } e \in E
\end{aligned}$$

The variables are used to determine if an edge is in the SMT or not. A value of 1 means that the corresponding edge is in the SMT. Note that the number of constraints is exponential in n, as there are 2^n different subsets of V. The optimal solution does not depend on the model used, but the quality of the bounds derived from its LP-relaxation changes depending on the model [Pol03, KM96].

CHAPTER 3

Theory on Steiner Trees

In order to ensure the correctness of our solver, we built our implementation on a base of theoretically proven techniques. In this chapter, we present the theory used for our solver. The goal is to give an understanding of the methods. The proofs have been omitted for brevity and can be found in the respective sources.

Our solver is split into two parts. One tries to simplify the instance and the other solves it. We call these two parts reducing module and solving module respectively. Both parts work with approximations, which is therefore the first topic we present. Afterwards we continue with the discussion of our two modules and finish the chapter with a presentation of local search methods. These methods are used to enhance non-optimal solutions or upper bounds.

3.1 Approximations

Upper and lower bounds are an important part of our solver. Approximations are used to calculate them. A recurring concept in this thesis, is that, whenever a lower bound exceeds an upper bound, the element tested cannot be optimal. This works for Steiner trees, sub-solutions, vertices and edges, as we discuss subsequently.

In this chapter, we discuss several methods for computing such bounds. Every approximation we present here is used in our solver. As approximations are often a trade off between tightness and computation time, we use more than one method.

A general concept is that any Steiner tree is an upper bound for an SMT on the same instance.

3.1.1 Repeated Shortest Path Heuristic (RSPH)

The Shortest Path Heuristic (SPH) is a method to calculate an upper bound. It is similar to Prim's algorithm [Pri57], used for calculating MSTs. SPH incrementally builds a graph T. Initially T consists of a single vertex called the root. In each iteration, T is extended by one terminal. The algorithm chooses the terminal closest to but not in T. It then adds to T the shortest path from T to this terminal. After a maximum of k iterations, this method yields a Steiner tree [TM80].

Example 3.1.1. Figure 3.1 shows an application of the SPH to our running example. Here c is chosen as the root. The algorithm establishes a Steiner tree in three iterations.■

The choice of the root is an important factor determining the quality of the approximation. By quality we refer to how close $|T|$ is to the optimal weight. The quality of the approximation can be enhanced by running the algorithm repeatedly for different choices as root. This root may be a terminal or a Steiner vertex. This extension is called RSPH [HRW92c][1].

Further gains can be obtained by computing an MST for the graph induced by $V[T]$ on G. The resulting tree is then optimized by repeatedly removing non-terminal leafs [RC86].

Algorithm 3.1 implements the ideas presented above. It uses a modified version of Dijkstra's algorithm (see Chapter 3.2.1) [Dij59] to speed up computation. The original algorithm has runtime $\mathcal{O}(kn^2)$ [TM80] and while the algorithm presented here has the same worst-case runtime, the average case is faster [dW02].

3.1.2 MSTs and 1-Trees

Next, we establish a method to compute lower bounds. For this we use insights gained from research on another $\mathcal{NP}$-hard problem, the *Traveling Salesman Problem (TSP)*. It is well researched and relates to ST, as we discuss subsequently. First, we have to introduce some definitions specific to this topic.

A *tour* is a cycle with no repeating vertices. Given a graph $G = (V, E)$ and a distance function $d : E \to \mathbb{N}$, we call the resulting network (V, E, d) metric if

$$d(v_i, v_\ell) \leq d(v_i, v_j) + d(v_j, v_\ell) \quad \forall v_i, v_j, v_\ell \in V$$

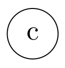
Note that the distance function d used in this thesis is metric.

The *TSP* is the problem of finding the shortest tour containing all vertices. Relaxing this condition to the shortest cycle containing all vertices yields the *metric-TSP*. This name is due to the fact, that it is equal to solving the TSP on the *metric completion*. Using the definitions from this thesis, the metric completion for (V, E, c) is $(V, V \times V, d)$. From here on we refer to the metric-TSP simply as TSP.

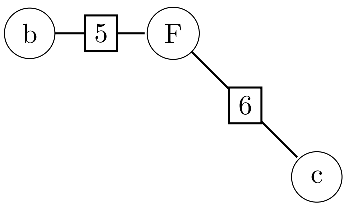
[1]The original source was not available, [HRW92c] refers to [WMS92]

Algorithm 3.1: RSPH [dW02, Chapter 5]

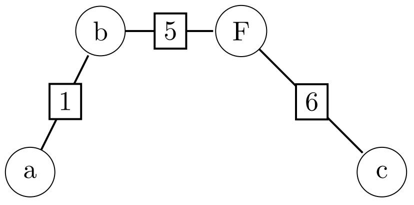
```
Data: An ST instance $(G, R)$ and a set of roots $B \subseteq V$
Result: A Steiner tree
return $\min_{r \in B} Compute(G, R, r)$
Function Compute(G, R, r) is
    $V' \leftarrow \{r\}, E' \leftarrow \emptyset$
    $Q \leftarrow \{r\}$
    $l(v) \leftarrow \infty, b(v) \leftarrow \emptyset$ for each $v \in V$
    $l(r) \leftarrow 0$
    while $R \nsubseteq V'$ do
        Choose $v_i \in Q$ minimizing $l(v_i)$
        $Q = Q \setminus \{v_i\}$
        if $v_i \notin R$ then  // As in Dijkstra's algorithm, see Chapter 3.2.1
            foreach $e_{ij} \in \delta_G(v_i)$ do
                if $l(v_j) > l(v_i) + c(e_{ij})$ then
                    $l(v_j) \leftarrow l(v_i) + c(e_{ij})$
                    $b(v_j) \leftarrow v_i$
                    $Q \leftarrow Q \cup \{v_j\}$
                end
            end
        else  // Add path to terminal and reset added vertices
            for $u \leftarrow v_i$ to $u = \emptyset$ do
                $V' \leftarrow V' \cup \{u\}$
                $Q \leftarrow Q \cup \{u\}$
                if $b(u) \neq \emptyset$ then
                    $E' \leftarrow E' \cup \{\{u, b(u)\}\}$
                $u = b(u)$
            end
            $l(v_j) = 0, b(v_j) = \emptyset$ for all $v_j \in V'$
        end
    end
    $T \leftarrow MST(V', \{e_{ij} \in E[G] \mid v_i, v_j \in V'\}, c)$
    do // Optimize by removing non-terminal leafs
        $O = V[T]$
        foreach $v_i \in V[T] \setminus R$ and $|\delta_G(v_i) \cap E[T]| = 1$ do
            $E[T] \leftarrow E[T] \setminus \delta_G(v_i)$
            $V[T] \leftarrow V[T] \setminus \{v_i\}$
        end
    while $O \neq V[T]$
    return $T$
end
```



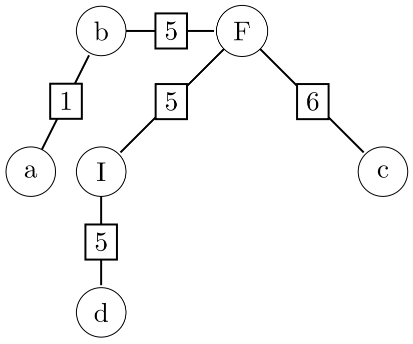
(a) Step 1: Our partial solution contains only the root (c).



(b) Step 2: The terminal closest to c has been added (b).



(c) Step 3: The terminal closest to our partial solution has been added.



(d) Step 4: The last terminal has been added.

Figure 3.1: SPH applied to our running example.

As mentioned before, a tour is a cycle. Removing any edge from the solution of a TSP results in a spanning tree. We can therefore calculate a lower bound for the TSP by computing the MST. Conversely, the minimum value of a tour containing all vertices halved is a lower bound for an MST. This is because given an MST, we can create a tour of at most double the cost. First, we double all edges. Then we can construct a cycle of double the cost: We traverse the graph and always choose an unused edge, preferring those connecting unvisited vertices (depth first traversal). We can convert this cycle into a tour in the original graph. We follow the cycle but skip any vertex that we have already visited. This is possible since the network is metric and yields a tour of at most double the cost of the MST. Therefore, the optimal value of a TSP halved is a lower bound for the value of an MST.

We can tighten the aforementioned bound. Given any vertex $v_i \in V$, a tour containing all vertices is a spanning tree for $V \setminus \{v_i\}$ with v_i connected to this tree by two edges. This is the definition of a *1-tree*. A minimal 1-tree for vertex v_i can be calculated by connecting v_i with its two cheapest edges to an MST of $V \setminus \{v_i\}$ [HK70]. Note that if a minimal 1-tree is a tour, it is the solution to the associated TSP.

By a similar argument as for the TSP, the weight of a minimal 1-tree halved is a lower bound for the value of an MST.

The connection to ST is as follows. Given an ST instance (G, R) and an SMT T. We construct a tour in $D_G(V[T])$. Since T is an MST for $D_G(V[T])$, we can use the same method as before to construct a tour with cost at most $2 * |T|$. We therefore have a 1-tree. Every edge in this 1-tree is a path in G. We can therefore create a Steiner tree of cost at most $2 * |T|$ from the 1-tree. This shows that the 1-tree is a valid lower bound for the SMT in G.

We now argue, that we do not need $D_G(V[T])$ to calculate this bound, but can use $D_G(R)$. Since the 1-tree we computed above is a tour, we can map each path between two terminals to an edge in $D_G(R)$. This yields a 1-tree in $D_G(R)$. The following lemma formalizes this [HSV16].

Lemma 1. *[HSV16, Lemma 8] Given an ST instance (G, R). Let $r \in R$ be any element and $R' = R \setminus \{r\}$. Furthermore, let w_{mst} be the weight of an MST of $D_G(R')$.*

$$\frac{w_{mst}}{2} + \min_{z_i, z_j \in R' : z_i \neq z_j \vee |R'| = 1} \frac{d(r, z_i) + d(r, z_j)}{2}$$

is lower bound for any SMT.

The distance network can be constructed in $\mathcal{O}(n \log n + m)$ and the MST for this network in $\mathcal{O}(k^2)$. Given the value for the MST, the calculation of the bound can be performed in time $\mathcal{O}(|R|)$ [HSV16].

3.1.3 Dual Ascent

Dual ascent is another approach to computing lower bounds. It uses the idea of *linear programming* discussed in Chapter 2.4. A feasible solution is a, not necessarily optimal, variable assignment for the ILP, complying with all constraints. Therefore, a feasible solution for ST formulation is a Steiner tree. Any feasible solution to an ILP is an upper bound.

The ILP can also be formulated as its dual. This is achieved by applying a specific transformation to the ILP. We do not discuss this transformation further, as it is not important for this chapter. The reader is referred to the literature for more information [Dan98]. The original model is called the primal. The two models relate to each other as follows: If the primal is a minimization problem, then the dual is a maximization problem. Any feasible solution for the dual is a lower bound for both the primal and the dual.

For the dual ascent algorithm we use the conversion to the Steiner arborescence problem discussed in Chapter 2.3. We need to also adapt the ILP from Chapter 2.4 to the directed version. Given a set $W \subset V$ with $r \notin W$ and containing at least one terminal. This set induces two subgraphs, one containing all the vertices in W and one containing all other vertices, including the root. For any such subset, a Steiner tree must contain an arc connecting the root subgraph with the one induced by W. This is the idea behind this formulation [Won84]:

$$\begin{aligned}
\text{minimize:} \quad & \sum_{a \in A} c(a) x_a \\
\text{subject to:} \quad & \sum_{\substack{a_{ij} \in A \\ v_i \notin W, v_j \in W}} x_{a_{ij}} \geq 1, && \text{for all } W \subset V, r \notin W, W \cap R \neq \emptyset \\
& x_a \in \{0,1\} && \text{for all } a \in A
\end{aligned}$$

The dual formulation of the problem uses a variable u_W for each $W \subseteq R, r \notin W, W \cap R \neq \emptyset$ [Won84, Pol03]:

$$\begin{aligned}
\text{maximize:} \quad & \sum u_W \\
\text{subject to:} \quad & \sum_{W, v_i \notin W, v_j \in W} u_W \leq c(a_{ij}), && \text{for all } a_{ij} \in A \\
& u_W \geq 0 && \text{for all } u_W
\end{aligned}$$

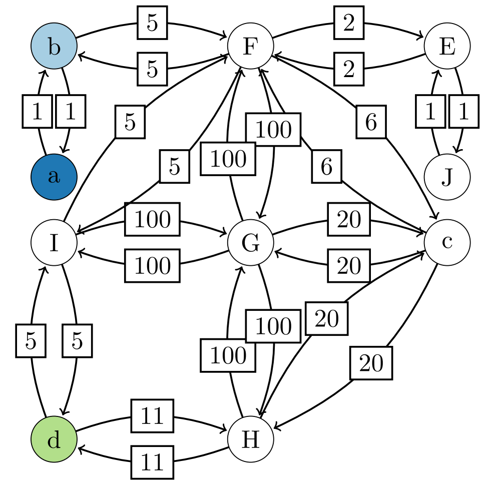
Note that in this case each cut is represented by a variable. Each such variable is assigned an integer value of at least 0. The algorithm starts with one component per terminal, consisting of exactly this terminal. It then iteratively adds arcs to these components, until the root is connected to all terminals.

Algorithm 3.2 shows the whole algorithm. It maintains a set Q of active terminals, the bound $\tilde{w}$ and reduced costs $\tilde{c}$. Initially all terminals except the root are in Q, the bound is 0 and $\tilde{c} = c$. Any arc such that $\tilde{c}(a_{ij}) = 0$ is called *saturated*. Let $cut : R \mapsto 2^V$ be a function that maps a terminal to all vertices that can reach it using only saturated arcs.

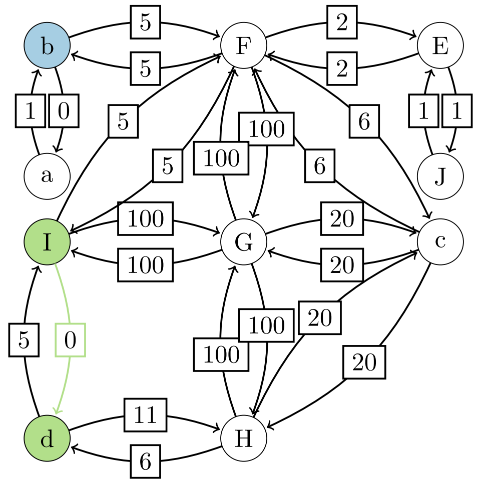
In each iteration a terminal from Q is chosen. If the cut contains any other active vertex or the root, the terminal is removed from Q, as it is already connected. Let F be the set of all arcs that go from a vertex not in the cut to a vertex in the cut. The lowest weight among all arcs in F is computed and $\tilde{c}$ is updated, such that the lowest weight is subtracted from all arcs in F. This yields at least one new saturated arc. The weight is furthermore added to the bound. As there is one new saturated arc in each iteration, the algorithm terminates. The result is a lower bound $\tilde{w}$ and the reduced costs $\tilde{c}$. The algorithm is illustrated in Figure 3.2 and runs in time $\mathcal{O}(m * \min\{n * k, m\})$ [Dui93]. We subsequently refer to $(V, E, \tilde{c})$ as a *dual ascent* graph.

Similar to the SPH, the choice of root impacts the quality of the bound. We can apply the same idea as for the RSPH and call the algorithm with different roots [Pol03].

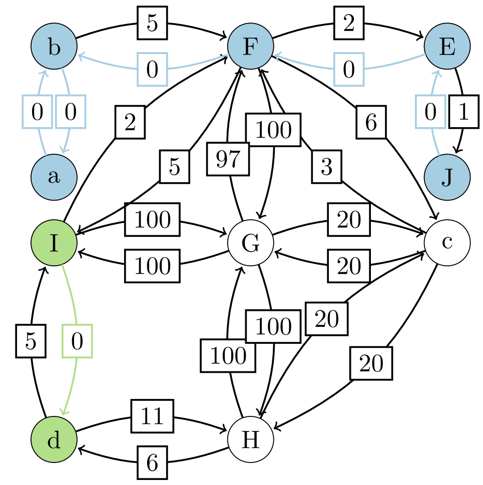
The choice of active vertex in each iteration is another important factor. In Line 5 some strategy for choosing an element from Q is applied. This choice affects the computed



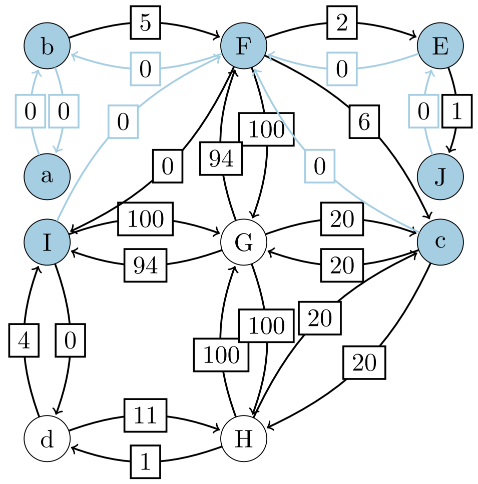
(a) Initial state, $A = \{a, b, d\}$



(b) After choosing a then d, $\tilde{w} = 6$, $A = \{a, d\}$



(c) After choosing b thrice, $\tilde{w} = 14$, $A = \{a, d\}$



(d) State after choosing d and b twice. The algorithm will terminate upon choosing b, $\tilde{w} = 22$

Figure 3.2: Dual ascent run with c as root. Colors indicate active components.

Algorithm 3.2: The dual ascent algorithm [PUW18, Chapter 3.1]

```
Data: A directed graph $G = (V, A, c)$, terminals $R \subseteq V$ and a root $r \in R$
$\tilde{c} \leftarrow c$
$\tilde{w} \leftarrow 0$
$Q \leftarrow R'$
while $|Q| > 0$ do
    Choose $z'$ in $Q$
    $C \leftarrow cut(z')$
    if $|(Q \cup \{r\}) \cap C| > 1$ then
        $Q \leftarrow Q \setminus \{z'\}$
    else
        $F \leftarrow \{a_{ij} \in A \mid v_i \notin C, v_j \in C\}$
        $c' \leftarrow \min_{a_{ij} \in F} \tilde{c}(a_{ij})$
        $\tilde{w} \leftarrow \tilde{w} + c'$
        foreach $a_{ij} \in F$ do
            $\tilde{c}(a_{ij}) \leftarrow \tilde{c}(a_{ij}) - c'$
        end
    end
end
return $\tilde{w}$
```

bound [Pol03]. Usually the algorithm tries to minimize some function, for example the size of the cut or the number of incoming arcs [Pol03, PUW18].

This concludes our discussion about approximations. We look at two areas where we use them next. These are solving algorithms and reductions.

3.2 Solver

In this chapter, we discuss different algorithms relating to our solving module. We first discuss *Dijkstra's algorithm*, a classic graph searching algorithm for shortest paths [Dij59]. We then present one of the classic solving algorithms for ST: The *Dreyfus-Wagner algorithm* [DW72]. Although its runtime of $\mathcal{O}(3^k)$ implies $\mathcal{FPT}$, its best-case runtime is not much better than the worst-case. It is therefore not efficient enough to compute SMTs for larger instances.

In the last part we discuss how we can combine the previous two algorithms. The new algorithm, which we refer to as the *Dijkstra-Steiner algorithm*, does not necessarily enumerate all possible sub-solutions. While not improving on the worst-case runtime, the best-case runtime is considerably better [HSV16].

In the following discussions the algorithms only compute the optimal value. For brevity we omitted the backtracking required to compute the actual paths/Steiner trees. We discuss it at the end of the chapter when presenting the final algorithm.

3.2.1 Dijkstra's algorithm

Dijkstra's algorithm is a classic graph search algorithm. It is a recurring topic in this thesis and we use several different implementations in our solver. In general it is used to find shortest paths in a graph. Algorithm 3.3 shows the version that calculates the minimal distance between two vertices.

The algorithm maintains a queue of vertices it has seen or scanned (Q) and stores the length of the shortest known path from the start vertex (l). In every iteration the vertex from the queue is picked that minimizes l. Then it is *expanded*: We check if we can reach any of the neighbors cheaper than previously thought. If so we have found a new shortest path and can add the neighbor to the queue. Once the target vertex is expanded, we have found a shortest path from start to target and stop [Dij59].

This idea is not limited to finding a shortest path from one vertex to another. If no target is used, the algorithm calculates the minimum paths from the start vertex to every other vertex. Also sets of vertices can be used as start and target.

The runtime of the algorithm depends on the distance between the vertices. If the vertices have maximum distance, every other option is explored first. The worst-case runtime is in $\mathcal{O}(n^2)$ [FT87].

Example 3.2.1. Figure 3.3 shows a run of the algorithm to find a path from b to H in our running example. In every iteration the dotted vertex with the shortest known distance is expanded. Note that in the final iteration it would be possible that G would have been expanded before H. Furthermore, the shortest path found uses c, because d has been expanded after c. ■

3.2.2 The Dreyfus-Wagner algorithm

This long known algorithm exploits the structure of Steiner trees. The idea is to calculate sub-solutions and incrementally build larger and larger sub-solutions until finally obtaining an SMT. The algorithm uses an important property of any SMT with three or more terminals.

Lemma 2. *[DW72, Optimal Decomposition Theorem]*
Let $T = (V', E')$ be an SMT for (G, R), such that $|R| \geq 3$. Then, for any $z_i \in R$ there exists:

- *A vertex $v_i \in V[G]$*
- *A subset I: $\varnothing \subset I \subset (R \setminus \{z_i\})$*
- *Disjoint sets E_1, E_2, E_3: $E' = E_1 \cup E_2 \cup E_3$, each inducing a connected subgraph.*

Algorithm 3.3: Dijkstra's algorithm [Dij59]

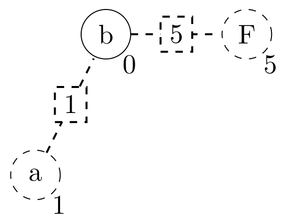
```
Data: A network $G = (V, E, c)$ and vertices $v_{start}$ and $v_{end}$
Result: The length of a shortest path between $v_{start}$ and $v_{end}$
$l(v_i) \leftarrow \infty$ for all $v_i \in V$
$l(v_{start}) \leftarrow 0$
$P \leftarrow \emptyset$
$Q \leftarrow \{v_{start}\}$
while $v_{end} \notin P$ do
    Choose $v_i \in Q$ minimizing $l(v_i)$
    $Q \leftarrow Q \setminus \{v_i\}$
    $P \leftarrow P \cup \{v_i\}$
    foreach $e_{ij} \in \delta(v_i)$ do
        if $l(v_j) > l(v_i) + c(e_{ij})$ then
            $l(v_j) \leftarrow l(v_i) + c(e_{ij})$
            $Q \leftarrow Q \cup \{v_j\}$
        end
    end
end

return $l(v_{end})$
```

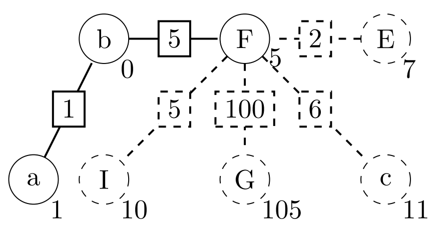
- *There exists* $T' \in smt_G(\{z_i, v_i\})$ *such that* $E[T'] = E_1$
- *There exists* $T' \in smt_G(I \cup \{v_i\})$ *such that* $E[T'] = E_2$
- *There exists* $T' \in smt_G((R \setminus (I \setminus \{z_i\})) \cup \{v_i\})$ *such that* $E[T'] = E_3$

Therefore, any SMT contains vertices v_i that connect sub-solutions. These vertices are called *junctions*. It suffices to compute these sub-solutions and join them at junctions, creating ever larger sub-solutions until finally creating a graph in $smt_G(R)$ [DW72].

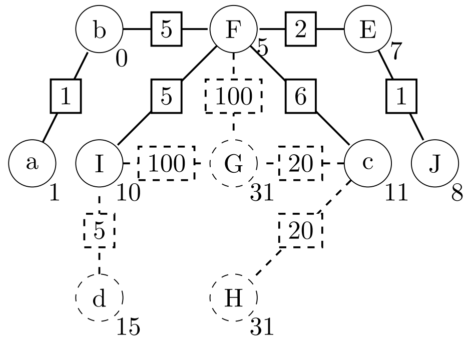
Algorithm 3.4 picks up this idea. It starts computing the smallest possible sub-solutions. These are the pairwise shortest paths between vertices and terminals. From there it starts processing terminal subsets by cardinality. For each cardinality t it creates optimal sub-solutions for each set $I \subseteq R'$ of cardinality t. This is done in two steps. First, all sub-solutions from disjoint sets composing I are merged. The minimum combination at every vertex is retained as tentative solution $l'(v_i, I)$. Now these tentative solutions are propagated to the other vertices. The combination corresponds to assuming that v_i is a junction for the two subtrees. The propagation connects a vertex to a junction. Therefore, the second step checks if it is cheaper to connect a vertex to a junction than joining the subtrees at that vertex. After the whole algorithm finishes, $l(r, R')$ contains the weight of any SMT for the current instance.



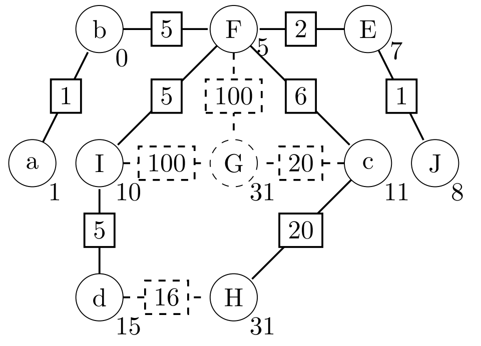
(a) After expanding the start vertex b.



(b) After expanding a then F.



(c) After expanding E, J, I and c



(d) Final state after choosing d then H, distance=31

Figure 3.3: *Dijkstra's algorithm* run from b to H. The number next to the vertex is the shortest known distance. Dotted vertices have been scanned and solid ones have been expanded. Dotted edges have been used by the algorithm and solid edges are part of a shortest path.

The whole algorithm runs in time $\mathcal{O}(3^k)$ [DW72] and requires memory in $\mathcal{O}(2^k n)$ [HSV16]. Although there are only $2^k n$ entries in l, there are 3^{k-1} possible combinations of disjoint subsets.

Example 3.2.2. Figure 2.4 shows the SMT of our running example. Here F is a junction. It joins the sub-solutions for $\{a, b\}$, $\{c\}$ and $\{d\}$.

Table 3.1 shows the result of full *Dreyfus-Wagner* run for our running example. c is used as a root. On the left side are the calculations for all terminals sets with cardinality 1. The entries are the distances from every vertex to the terminal in the set. Next are the calculations for all sets of size two, these are in the middle of the table. For each set the l' values are calculated by combining the single element sets on the left side. Next the l values are calculated using the l' values and the distances between the vertices. The l' values in the table's right are calculated by using the l values from the two and single element sets. The last l value needs only be calculated for the root and contains the solution. ■

Algorithm 3.4: The Dreyfus-Wagner algorithm [DW72, Chapter 2]

```
   Data: Instance $(G, R)$ of ST and a root $r \in R$
   Result: The weight of an SMT
1  $R' \leftarrow R \setminus \{r\}$
2  $l(v_i, \{z_j\}) \leftarrow d(v_i, z_j)$ for all $z_j \in R$, $v_i \in V$
3
4  for $t = 2$ to $|R'|$ do
5      foreach $I \subseteq R'$ such that $|I| = t$ do
6          foreach $v_i \in V[G]$ do  $l'(v_i, I) \leftarrow \min_{\emptyset \subset J \subset I}(l(v_i, J) + l(v_i, I \setminus J))$
7
8          foreach $v_i \in V[G]$ do  $l(v_i, I) \leftarrow \min_{v_j \in V[G]}(d(v_i, v_j) + l'(v_j, I))$
9
10     end
11 end
12 return $l(r, R')$
```

	$\{a\}$	$\{b\}$	$\{d\}$	$\{a,b\}$		$\{a,d\}$		$\{b,d\}$		$\{a,b,d\}$	
Vertex	l	l	l	l'	l	l'	l	l'	l	l'	l
a	0	1	16	1	1	16	16	17	16	16	
b	1	0	15	1	1	16	16	15	15	16	
c	12	11	16	23	17	28	22	27	26	33	*22*
d	16	15	16	31	16	32	21	31	25	36	
E	8	7	12	15	8	20	18	19	17	25	
F	6	5	10	11	6	16	16	15	15	16	
G	32	31	36	63	32	68	42	67	41	68	
H	32	31	16	63	32	48	42	47	41	48	
I	11	10	5	21	11	16	16	15	15	16	
J	9	8	13	17	9	22	19	21	18	22	

Table 3.1: The full *Dreyfus-Wagner* calculation for our running example.

3.2.3 Applying Dijkstra's algorithm

Although the *Dreyfus-Wagner algorithm* is $\mathcal{FPT}$ given k, it quickly becomes time and memory consuming even for double digit k. In an effort to improve the best-case runtime, a method similar to *Dijkstra's algorithm* can be added.

The idea presented in Algorithm 3.5 is to define a neighborhood for sub-solutions. The neighborhood is based on both graph neighborhood and disjoint sets of terminals. Entries in the queue are prioritized by the weight of the underlying sub-solution. Whenever an entry is processed all neighboring sub-solutions are created and added to the queue. This process is continued until an optimal solution is found. Since the queue is ordered by weight, all partial solutions that are more expensive than the optimum are disregarded.

Algorithm 3.5: The preliminary *Dijkstra-Steiner algorithm* [HSV16]

```
Data: An ST instance $(G, R)$ and a root $r \in R$
Result: The weight of an SMT
$R' \leftarrow R \setminus \{r\}$
$l(v_i, I) \leftarrow \infty$ for all $(v_i, I) \in V \times 2^{R'}$
$l(z_i, \{z_i\}) \leftarrow 0$ for all $z_i \in R'$
$l(v_i, \varnothing) \leftarrow 0$ for all $v_i \in V$
$Q \leftarrow \{(z_i, \{z_i\}) \mid z_i \in R'\}$
while $(r, R') \notin P$ do
    Choose $(v_i, I) \in Q$ minimizing $l(v_i, I)$
    $Q \leftarrow Q \setminus \{(v_i, I)\}$
    $P \leftarrow P \cup \{(v_i, I)\}$
    foreach $e_{ij} \in \delta(v_i)$ do
        if $l(v_j, I) > l(v_i, I) + c(e_{ij})$ then
            $l(v_j, I) \leftarrow l(v_i, I) + c(e_{ij})$
            $Q \leftarrow Q \cup \{(v_j, I)\}$
        end
    end
    forall $\varnothing \subset J \subseteq (R' \setminus I)$ with $(v_i, J) \in P$ do
        if $l(v_i, I \cup J) > l(v_i, I) + l(v_i, J)$ then
            $l(v_i, I \cup J) \leftarrow l(v_i, I) + l(v_j, J)$
            $Q \leftarrow Q \cup \{l(v_i, I \cup J)\}$
        end
    end
end

return $l(r, R')$
```

The currently best known solution is $l(v_i, I)$. The value is optimal once the tuple is expanded, i.e. $(v_i, I) \in P$. In every iteration the cheapest sub-solution is chosen from the queue. The result is propagated to neighboring vertices. Furthermore, if there are known optimal disjoint solutions at the current vertex, they are recombined into larger sub-solutions. Created entries are added to the queue for later processing. As soon as we have an optimal entry for the root and the set of all terminals, we have a solution.

Note that this isn't entirely the same principle as Dijkstra's algorithm. The merging of solutions depends on the vertices expanded before, which is different to the original algorithm. A straightforward application of Dijkstra's algorithm would immediately create all larger sub-solutions, as they are in the neighborhood. This is not possible, because, in contrast to edge costs, we do not know how much transitioning from sub-

l				Q	
v	E	d	P	v	E
a	$\{a\}$	0	x	b	{a}
	$\{b\}$	1		a	{b}
b	$\{b\}$	0	x	F	{d}
	$\{a\}$	1		I	{d}
d	$\{d\}$	0	x	H	{d}
F	$\{d\}$	5			
H	$\{d\}$	16			
I	$\{d\}$	5			

$\Rightarrow$

l				Q	
v	E	d	P	v	E
a	$\{a\}$	0	x	a	{b}
	$\{b\}$	1		b	{a,b}
b	$\{b\}$	0	x	F	{d}
	$\{a\}$	1	x	I	{d}
	$\{a, b\}$	1		F	{a}
d	$\{d\}$	0	x	H	{d}
F	$\{b\}$	5			
	$\{a\}$	6			
H	$\{d\}$	16			
I	$\{d\}$	5			

Table 3.2: Data structure changes in one *Dijkstra-Steiner* iteration.

solution to sub-solution costs. Therefore, we cannot automatically transfer insights about Dijkstra's algorithm to this one.

Example 3.2.3. Listing all the steps in the calculation of our running example would take too long. Instead Table 3.2 shows one iteration of the algorithm, from iteration 3 to 4. The first element in the queue, $b, \{a\}$ is chosen in the iteration. The entry is marked in P and propagated to its neighbors. For a the new sub-solution is no improvement over the known one. F does not know any sub-solution for $\{a\}$ yet, so the entry is created. Furthermore, the entries for $\{a\}$ and $\{b\}$ are combined to a larger sub-solution. Finally the corresponding queue entries are created. ■

3.2.4 Adding a guiding heuristics

Algorithm 3.5 expands all sub-solutions that have lower weight than $|smt_G(R)|$. This can be a large amount, especially if the total cost is high. We can apply another idea used in graph search to avoid this problem. A^* applies a guiding heuristic to speed up the search in Dijkstra's algorithm [HNR68]. Instead of prioritizing the entries in Q by distance alone, a heuristic lower bound is added. Ideally entries closer to the target have a lower heuristic value and are prioritized.

Let $h : V \times 2^R \to \mathbb{N}$ be the heuristic and l^* be the optimal weight. The heuristic is required to be admissible: $l^*(v_i, I) + h(v_i, R \setminus I) \leq l^*(v_i, R)$ for all $v_i \in V, I \subset R$. Therefore $h(v_i, I)$ is a lower bound for $l^*(v_i, I)$.

In the preliminary algorithm, elements from Q are prioritized by $l(v_i, I)$. We now add the heuristic value and use $l(v_i, I) + h(v_i, R \setminus I)$ for prioritization.

The use of a heuristic does not guarantee lower runtime. In the worst case the number of enumerated sub-solutions is the same and the added computation time for the heuristic worsens runtime. Besides potential runtime gains, the use of a heuristic can also drastically reduce memory consumption.

Example 3.2.4. For our running example, the algorithm requires a minimum of ten iterations. The necessary operations can be tracked using the SMT in Figure 2.4. In the beginning the initial entries $(a, \{a\}, 0), (b, \{b\}, 0)$ and $(d, \{d\}, 0)$ are expanded. As a results $(I, \{d\}, 5)$ and $(b, \{a\}, 1)$ are expanded. After expanding $(b, \{a, b\}, 1)$, all necessary sub-solutions are present at the junction F. After expanding $(F, \{d\}, 10)$ and $(F, \{a, b\}, 6)$, we have the combined sub-solutions. The only thing remaining is to propagate it towards the root. This is achieved by expanding $(F, \{a, b, d\}, 16)$ and finally $(c, \{a, b, d\}, 22)$ yielding the solution. This run creates 20 entries in Q.

If we have a perfect heuristic, i.e. $h(v_i, I) = l^*(v_i, I)$, our algorithm takes exactly this path. Without the heuristic, we have to expand every sub-solution with weight smaller than 22 before. This takes about 55 iterations and about 90 entries in Q. Note that this is still considerably less than the worst case of 160 iterations. ■

3.2.5 Pruning

The guiding heuristic allows us a navigate the search space more efficiently. Pruning reduces the search space. Before we add any entry to Q, we validate if it can be part of any SMT. If not it is not added to Q and subsequently not to P. This avoids unnecessary iterations and can lower the number of combinations for subsets significantly. We use upper and lower bounds to identify non-optimal sub-solutions.

A very fast test allows us to discard some entries. Since the heuristic is a lower bound, whenever $l(v_i, I) + h(v_i, R \setminus I)$ exceeds an upper bound, we can discard the entry. This method merely saves memory, as the entries would never have been expanded.

The following lemma allows for a test that provides a potentially tighter bound.

Lemma 3. *[HSV16, Lemma 15] Let (G, R) be an ST instance, $v_i \in V[G]$, $I \subset R$ and $\emptyset \subset U \subseteq (R \setminus I)$. Furthermore, let T_1 be a Steiner tree for $\{v_i\} \cup I$ and $H = (V', E', c)$ be a, not necessarily connected, subgraph of G such that:*

1. *$(I \cup U) \subseteq V'$*
2. *Each connected component of H contains a terminal in U.*
3. *$|H| < |T_1|$*

If such a graph H exists, there is no SMT for R that contains T_1 as a subgraph.

The definition of H seems a little obscure. T_1 is a Steiner tree containing $I \subset R$. Now assume we have another Steiner tree T_2 containing $R \setminus I$. It especially contains all the vertices in U and is connected. If we add all the vertices and edges from H to T_2 we have a (connected) Steiner tree for R. The lemma therefore states that, if connecting I to a Steiner tree for $R \setminus I$ is cheaper than T_1, the current sub-solution is not part of an SMT.

We can easily construct such a graph H using Algorithm 3.6. Given a sub-solution $T' \in smt_G(I \cup \{v_i\})$. We now take a terminal $z_j \in R \setminus (I \cup \{v_i\})$ and add it via its shortest path to T'. If we set $U = \{z_j\}$, the new graph meets the first two required criteria.

To lower the weight of the graph, we choose z_j to be of minimum distance to T'. We keep the weight of the final graph as an upper bound. It can be used to test all future sub-solutions using I.

Algorithm 3.6: Prune method [HSV16, Chapter 5]

```
Data: An ST instance $(G, R)$, a root $r \in R$ and a function $l : V \times 2^R \to \mathbb{N}$ as used
      in Algorithm 3.3
Result: true if the current sub-solution can be pruned and false otherwise.
$ub(I) = \infty$ for all $I \subseteq R \setminus \{r\}$
$U_{ub}(I) = \emptyset$ for all $I \subseteq R \setminus \{r\}$
Function $prune(v_i, I)$ is
    if $l(v_i, I) > ub(I)$ then
        return true
    $z_i, z_j \leftarrow \underset{z_i \in I \cup \{v_i\}, z_j \in R \setminus I}{\arg\min} d(z_i, z_j)$
    if $l(v_i, I) + d(z_i, z_j) < ub(I)$ then
        $ub(I) \leftarrow l(v_i, I) + d(z_i, z_j)$
        $U_{ub}(I) \leftarrow \{z_j\}$
    end
    return false
end
Function $prune\_combine(v_i, I, J)$ is
    if $U_{ub}(I) \cap J = \emptyset$ or $U_{ub}(J) \cap I = \emptyset$ then
        $u \leftarrow U_{ub}(I) + ub(J)$
        $X \leftarrow I \cup J$
        if $ub(X) > u$ then
            $ub(X) \leftarrow u$
            $U_{ub}(X) \leftarrow (U_{ub}(I) \cup U_{ub}(J)) \setminus X$
        end
    end
    return $prune(v_i, I \cup J)$
end
```

Example 3.2.5. We apply pruning to our running example without using a heuristic. For a and b the closest other terminal has distance 1 and for d it has distance 10. Therefore, after expanding the initial entries we have $ub(\{a\}) = ub(\{b\}) = 1$ and $ub(\{d\}) = 15$. As discussed in the previous example, the sub-solutions for $\{a\}$ and $\{b\}$ are joined at b which has distance 11 to terminal c. This results in $ub(\{a, b\}) = 12$. When sub-solutions $\{a, b\}$ (weight 6) and $\{d\}$ (weight 10) are merged at F it is established that $ub(\{a, b, d\}) = 22$. In later iterations it is also established that $ub(\{a, d\}) = 16$ and $ub(\{b, d\}) = 15$.

Running the whole algorithm with pruning requires 25 iterations and creates 27 entries in the queue. This is a considerable improvement. In practice, the effectiveness of pruning depends on the instance [HSV16]. ■

3.2.6 Final algorithm

In Algorithm 3.7 everything discussed in this chapter is put together. The guiding heuristic is used in Line 9. Note that the heuristic requires R and not R' as we need a bound for all remaining terminals. In Lines 16 and 24 are the checks for pruning. The information for backtracking the SMT is gathered in Lines 5, 15 and 23. The actual backtracking is performed in Line 31.

The original algorithm had a stronger requirement for the heuristic called consistency [HSV16]. A heuristic is consistent if

$$h(v_i, I) \leq h(v_j, I') + l^*(v_j, I \setminus I' \cup \{v_i\}) \qquad \text{for all } v_i, v_j \in V \text{ and } \{r\} \subseteq I' \subseteq I \subseteq R$$

As we use heuristics that do not have this property, the algorithm has been adapted to accommodate for this fact. If the heuristic is consistent, we can skip adding entries to Q if they are already in P. This extra check is performed in the conditionals in lines 13 and 21. Another consequence of using inconsistent heuristics is, that even if an entry is in P, it is not guaranteed, that weight is optimal, except for the full terminal set at the root.

The original correctness proof does not work with inconsistent heuristics [HSV16]. The important invariant $(v_i, I) \in P \Rightarrow l(v_i, I) = |smt_G(\{v_i\} \cup I)|$ is lost. If the heuristic is inconsistent, it may happen that a cheaper sub-tree is found at a later point, violating the invariant. We therefore prove that correctness still holds.

3.2.7 Correctness

We now prove that the adapted algorithm presented above is still correct, if the heuristic is not consistent. We show this without pruning to simplify the proof. The proof for pruning follows from showing that the invariants still hold. Intuitively, pruning uses $l(v_i, I)$ to calculate an upper bound, if the value is non-optimal, this results in a higher upper bound, decreasing the effectiveness, but preserving validity.

We use the following results from the original proof, that did not depend on the consistency of the heuristic [HSV16]:

- R1 For each $(v_i, I) \in P \cup Q$ holds that:
  - R1.1 $b(v, I) \subseteq P$ and $backtrack(v, I)$ returns the edge set of a connected graph T such that all vertices $\{v\} \cup I$ are in T and $|T| \leq l(v, I)$.
  - R1.2 $I \cup \{v_i\} = \{v_i\} \cup \bigcup_{(v_j, J) \in b(v_i, I)} J$.
- R2 For each $(v_i, I) \in (V \times 2^{R'}) \setminus P$ holds:
  - R2.1 $l(v_i, I) \geq |smt_G(\{v_i\} \cup I)|$.
  - R2.2 if $l(v_i, I) = |smt_G(\{v_i\} \cup I)|$, then $(v_i, I) \in Q$.

Algorithm 3.7: The final Dijkstra-Steiner algorithm [HSV16]

```
Data: An ST instance $(G, R)$, a root $r \in R$, an admissible heuristic $h$ and an
      upper bound $ub'$
Result: An SMT and its weight
$R' = R \setminus \{r\}$
$l(v_i, I) \leftarrow \infty$ for all $(v_i, I) \in V \times 2^{R'}$
$l(z_i, \{z_i\}) \leftarrow 0$ for all $z_i \in R'$
$l(v_i, \varnothing) \leftarrow 0$ for all $v_i \in V$
$b(v_i, I) \leftarrow \varnothing$ for all $(v, I) \in V \times 2^{R'}$
$Q \leftarrow \{(z_i, \{z_i\}) \mid z_i \in R'\}$

while $(r, R') \notin P$ do
    Choose $(v_i, I) \in Q$ minimizing $l(v_i, I) + h(v_i, R \setminus I)$
    $Q \leftarrow Q \setminus \{(v_i, I)\}$
    $P \leftarrow P \cup \{(v_i, I)\}$
    foreach $e_{ij} \in \delta(v_i)$ do
        if $l(v_i, I) + c(e_{ij}) < l(v_j, I)$ then
            $l(v_j, I) \leftarrow l(v_i, I) + c(e_{ij})$
            $b(v_j, I) \leftarrow \{(v_i, I)\}$
            if $l(v_i, I) \leq ub'$ and not $prune(v_i, I)$ then
                $Q \leftarrow Q \cup \{(v_j, I)\}$
        end
    end
    forall $\varnothing \subset J \subseteq R' \setminus I$ with $(v_i, J) \in P$ do
        if $l(v_i, I) + l(v_i, J) < l(v_i, I \cup J)$ then
            $l(v_i, I \cup J) \leftarrow l(v_i, I) + l(v_j, J)$
            $b(v_i, I \cup J) \leftarrow \{(v_i, I), (v_i, J)\}$
            if $l(v_i, I \cup J) \leq ub'$ and not $prune\_combine(v_i, I, J)$ then
                $Q \leftarrow Q \cup \{l(v_i, I \cup J)\}$
        end
    end
end
return $backtrack(r, R'), l(r, R')$

Function $backtrack(v_i, I)$ is
    if $b(v_i, I) = \{(v_j, I)\}$ then
        return $\{e_{ij}\} \cup backtrack(v_j, I)$
    else
        return $\bigcup_{(v_j, I') \in b(v_i, I)} backtrack(v_j, I')$
end
```

We also introduce a few definitions. We denote by $T(v_i, I)$ the graph defined by $backtrack(v_i, I)$. Furthermore we define a a tuple (v_i, I) to be optimal, if it holds that $l(v_i, I) = |smt_G(I \cup \{v_i\})|$ and $T(v_i, I)$ is a subgraph of at least one $T \in smt_G(R)$. Note that an optimal entry is never overwritten, since any new entry cannot have lower weight by *R1.1* and *R2.1*.

We define the transformation $tr(v_i, T_1, T_2)$ for two trees T_1, T_2, where T_1 is a subgraph of T_2 and $v_i \in V[T_1]$. Let $V' = V[T_2] \setminus (V[T_1] \setminus \{v_i\})$ and $E' = (E[T_2] \setminus E[T_1])$. The result of the transformation is the graph $T' = (V', E')$. Therefore, the result of tr is the remaining graph, after removing the edges in T_1.

Furthermore we define $tr_r(v_i, T_1, T_2)$ analogously to tr, adding a root vertex $r \in V[T_1]$. Let $T' = tr(v_i, T_1, T_2)$. Since we have a tree, there exists a unique path $S_{v_i,r}$ from v_i to r. Let $T'' = tr(v_i, S_{v_i,r}, T')$ and C_i be the connected component in T'' containing v_i. The result of the transformation is the graph $(C_i, E[T''] \cap (C_i \times C_i))$. After the transformation, only the subtree rooted at v_i, without T_1, remains of T_2.

Lemma 4 (Loop Invariant). *The following invariants hold at the start of every iteration of the main loop in Algorithm 3.7, Line 8: There exists a partition $\mathcal{W}$ of R' such that:*

- *H1 There exists a $T \in smt_G(R)$, such that*
  - *H1.1 For each $W \in \mathcal{W}$, there exists a $(v_i, W) \in P \cup Q$ such that $T(v_i, W)$ is a subgraph of T and (v_i, W) is optimal.*
  - *H1.2 For each $(v_i, W) \in P \cup Q$, such that $T(v_i, W)$ is a subgraph of T, there exists no $W' \in \mathcal{W}$ such that $(v_i, W \cup W')$ is optimal, $T(v_i, W')$ is a subgraph of T and $(v_i, W \cup W') \in P \cup Q$.*
- *H2 For each $W \in \mathcal{W}$ it holds that either*
  - *H2.1 There exists a tuple $(v_i, W) \in Q$ such that (v_i, W) is optimal.*
  - *H2.2 There exists a tuple $(v_i, W) \in P$ such that (v_i, W) is optimal. Furthermore, there exists $T \in smt_G(R)$ such that $T(v_i, W)$ is a subgraph of T and $|R \cap V[tr_r(v_i, T(v_i, W), T)]| > 0$.*

Before we prove the lemma, we prove some results of this invariant. In the following lemmas and proofs we refer to the invariant conditions by the label.

Lemma 5. *Let $\mathcal{W}$ be a partitioning of R' as described in Lemma 4. Let (v_i, I) be the element chosen in the current iteration from Q. If (v_i, I) is optimal and $I \in \mathcal{W}$, then invariant* H2 *holds after the current iteration.*

Proof. We assert that after the iteration there exists an element (v_o, I) such that *H2* holds. Let $T \in smt_G(R)$ be any SMT containing $T(v_i, I)$ as a subgraph. Due to optimality such a graph exists. Let $S_{v_i,r}$ be the unique path from v_i to r in T. Furthermore, we use $\phi(v_j) := V[R \cap tr_r(v_j, T(v_j, I), T)]$ (see *H2.2*). We prove the lemma by contradiction. We therefore assume that no tuple $(v_o, I) \in P$ exists, such that *H2* holds. Therefore for all optimal (v_j, I) holds that, $(v_j, I) \notin Q$ and $|\phi(v_j)| = 0$.

We now show the following result by induction on ℓ, where x_ℓ denotes the ℓ-th vertex in $S_{v_i,r}$. Note that the property defined by $\phi(v_j)$ implies, that v_j has degree at maximum two in T. If it had degree 3 or higher, one of the subtrees would contain no terminal, contradicting that T is minimal.

- *Induction Hypothesis*: For every vertex $v_j \in V[S_{v_i,r}]$ it holds that, (v_j, I) is optimal, $(v_j, I) \in P, (v_j, I) \neq Q$ and $|\phi(v_j)| = 0$.
- *Base Case (x_1)*: (v_i, I) is by definition optimal. In this iteration it is removed from Q and added to P. Furthermore, by assumption $|\phi(v_i) = 0|$.
- *Induction Step ($x_{\ell+1}$)*: For vertex $x_{\ell+1}$ it holds that by hypothesis $(x_\ell, I) \in P$ and (x_ℓ, I) is optimal. Together with $(x_\ell, I) \notin Q$ this implies, that (x_ℓ, I) has been expanded with optimal weight. Due to the optimality of (x_ℓ, I) and because x_ℓ has by assumption degree 1 or 2, the optimal weight for $(x_{\ell+1}, I)$ is $l(x_\ell, I) + c(\{x_\ell, x_{\ell+1}\})$. Therefore, after the expansion of (x_ℓ, I), the tuple $(x_{\ell+1}, I)$ was optimal. By *R2.2* it is therefore either in Q or P. Therefore, by assumption $(x_{\ell+1}, I) \in P$ and also by assumption $|\phi(x_{\ell+1})| = 0$.

Since $r \in S_{v_i,r}$ and $\phi(r) > 0$ we have a contradiction. □

Lemma 6. *Whenever an element from Q is chosen in the loop in Line 8 there exists a $(v_i, I) \in Q$ that is optimal.*

Proof. Let $\mathcal{W}$ be a partition as described in Lemma 4. We first show that given T as defined in *H1* and $\phi(v_j) \coloneqq V[R \cap tr_r(v_j, T(v_j, I), T)]$, it holds that for every $W \in \mathcal{W}$ exists an optimal entry $(v_i, W) \in P$ such that $|\phi(v_i)| > 0$ and $T(v_i, W)$ is a subtree of T. Note that this is different from *H2.2* as T is a specific SMT.

Let $W \in \mathcal{W}$ be an arbitrary element and (v_i, W) an optimal entry such that $T(v_i, W)$ is a subgraph of T. We know from *H1* that such an entry exists. Now we can show the desired property by using the same proof as in Lemma 5.

We can now prove the lemma. From the previous discussion, we know that for every $W \in \mathcal{W}$ exists an entry satisfying *H2.2* in T. As each of the entries defines a subtree in T, we have $|\mathcal{W}|$ subtrees. There can be at maximum $|\mathcal{W}| - 1$ junctions joining these subtrees. Every junction reduces the number of subtrees by one and the last subtree remaining is the tree itself. We define $A_i = \{W \in \mathcal{W} \mid (v_i, W) \text{ is optimal}\}$, the set of optimal entries in $\mathcal{W}$ for a vertex. Since we have $|\mathcal{W}|$ subtrees connected by $|\mathcal{W}| - 1$ junctions, there exists a $v_i \in V$ such that $|A_i| > 1$. Since all entries $(v_i, W), W \in A_i$ are optimal, the entry $(v_i, \bigcup_i A_i)$ would have been added to Q with optimal value in Line 25. This contradicts *H1.2*. □

We now have all the tools to prove Lemma 4:

Proof. The invariant clearly holds at the start of the first iteration. Every entry in Q is of the form $(z_i, \{z_i\})$. This is clearly a partition and every entry is an optimal subtree of any SMT.

We assume that the conditions hold and we are at the beginning of an arbitrary iteration. We remove an element (v_i, I) from Q. We proceed with a case distinction on this tuple:

Case 1: (v_i, I) is not optimal. In this iteration no optimal value can be overwritten in l, as the weight of any tuple processed in this iteration cannot be lower than an optimal value. The invariant still holds at the end of the iteration.

Case 2: (v_i, I) is optimal, but there exists no partitioning $\mathcal{W}$, such that $I \in \mathcal{W}$ and *H1* and *H2* hold for $\mathcal{W}$. By the invariant such a partition exists. The loop in Line 12 cannot overwrite any entries defined in the invariant, as no optimal entry can be overwritten, and no changes to l and b are performed in case of equality.

In case the loop in Line 20 creates an optimal entry $(v_i, I \cup J)$. This can invalidate the invariant only, if there exist $I', J' \in \mathcal{W}$ such that $I \cup J = I' \cup J'$. Let T be the SMT as defined in *H1*. It holds that $I \notin \mathcal{W}$ and therefore, either $I \nsubseteq I'$ or $J \nsubseteq J'$. This implies that either $T(v_i, I)$ is not a subtree of T or $T(v_i, J)$ is not. Therefore, *H1.2* still holds, as well as the other invariants.

Case 3: (v_i, I) is optimal and there exists a partitioning $\mathcal{W}$ as described in the lemma. We only consider optimal entries that are created during the iteration. All other entries do not affect any of the invariant rules.

If no new optimal entries in Q are created, then by Lemma 5 *H2* is preserved. As (v_i, I) is added to P, *H1.1* holds. And as no new entries are created, *H1.2* holds as well.

For every optimal entry processed in the loop at Line 12 (propagation), *H2.1* and therefore *H2* is preserved. Since (v_i, I) is added to P, *H1.1* is also preserved. Finally, since all entries are created with I, *H1.2* holds as well.

For every optimal entry processed in the loop at Line 20 (merging). Entries created here use $I \cup J$ as the set. We proceed with a case distinction on J:

> *Case 3a*: $J \notin \mathcal{W}$; *H1* holds after the iteration as $(v_i, I) \in P$. *H1.1* holds since $(v_i, I) \in P$ and *H1.2* holds because $J \notin \mathcal{W}$. Since $(v_i, I \cup J)$ is optimal, there exists a $T \in smt_G(R)$ such that $T(v_i, I \cup J)$ is a subtree. Since $T(v_i, I)$ and $T(v_i, J)$ are the subtrees of $T(v_i, I \cup J)$ and each contains at least one terminal, *H2.2* holds.
>
> *Case 3b*: $J \in \mathcal{W}$; We argue that $\mathcal{W}' = (\mathcal{W} \setminus \{I, J\}) \cup \{I \cup J\}$ is a partition preserving the invariant. Let T be the SMT as described in *H1* for $\mathcal{W}$. *H1.1* holds since either $T(v_i, I \cup J)$ is a subgraph of T, or one of $T(v_i, I)$, $T(v_i, J)$ is no subgraph of T. *H1.2* holds since it held for the smaller sets and therefore also for the larger set. Since at the beginning of the iteration $(v_i, I \cup J) \notin P \cup Q$ by *H1.2*, $(v_i, I \cup J)$ is added to Q and therefore *2.1* and consequently *H2* holds. □

Next, we show the desired result.

Theorem 7. *Algorithm 3.7 terminates and when $(r, R') \in P$ then $l(r, R')$ is the weight of an SMT for the instance (G, R).*

Proof. For correctness we have to show that whenever $(r, R') \in P$ then $l(r, R') = |smt_G(R)|$. Proof by contradiction, we assume $(r, R') \in P$ and $l(r, R')$ is not equal to the optimal weight. Remember that heuristic h is admissible and therefore $l^*(v_i, I) + h(v_i, R \setminus I) \leq l^*(v_i, R)$ for all $v_i \in V, I \subset R$.

Let $w = |smt_G(R)|$. Due to the admissibility of the heuristic it holds that $w' = l(r, R') + h(r, \{r\}) = l(r, R')$. And since the solution is not optimal by the property defined above $w' > w$. Since we expanded the entry, it holds that at this iteration, no entry $(v_i, I) \in Q$ such that $l(v_i, I) + h(v_i, R \setminus I) \leq w$ existed. Any optimal entry (v_j, J) would have $l(v_j, J) + h(v_j, R \setminus J) \leq w$ due to optimality and the admissibility of the heuristic. Therefore, there existed no optimal entry in Q, contradicting Lemma 6.

For termination consider that any value of l that is set (not ∞) is clearly bounded above by $|G|$. The maximum number of values in l is $2^n n$. Since every update of l requires a lower value than the existing one, l is updated at most $2^n n * |G|$ times. Any update of l can produce one entry in Q. Including the initial values, there are at most $2^n n * |G| + k$ entries added to Q. Since this number is finite, the algorithm terminates. □

We do not know how the use of an inconsistent heuristic impacts the runtime. The termination proof gives us a bound on the number of iterations, but not a very good one. Consistency guarantees us, that the estimates decrease if we get closer to the target. This is true for many everyday estimation methods, like *Euclidian distance*. Admissibility on the other hand has a comparatively weak guarantee. It states, that we never overestimate the distance. This allows for the possibility that we find a shorter path to an already expanded entry.

As stated before, the *Dijkstra-Steiner algorithm* is similar to but different than *Dijkstra's algorithm* and A^*. It is therefore not possible to directly use knowledge from this domain. It nonetheless shows possible impacts. If an inconsistent heuristic is used, the runtime of A^* deteriorates from $\mathcal{O}(n^2)$ to $\mathcal{O}(2^n)$ [Mar77]. The actual worst-case runtime depends on the heuristic and may well be $\mathcal{O}(n^2)$ [ZSH+09]. On the one hand this suggests, that the worst case runtime of our new algorithm, is considerably worse. On the other hand, the impact may well be negligible as long as heuristic produces mostly consistent results.

This concludes our discussion of solving algorithms. We established the method we use to compute SMTs and present methods to simplify an instance to reduce computation time next.

3.3 Reductions

The runtime for solvers usually depends on the size of the graph and the number of terminals. One of the ways to speed up computation is therefore a reduction in the size of the instance. Therefore, removing components of a graph: vertices, terminals or edges. Reductions use computationally fast tests to find areas of the graph that can be transformed into a simpler graph. Given that these tests can be run quick enough, the overall runtime of the solver can be improved considerably. Therefore, most reductions we present in this chapter run in $\mathcal{O}(m + n \log n)$.

Each reduction consists of a test and a transformation. Whenever the test succeeds, the transformation is run. The transformation maps the instance to a, preferably easier solvable, instance. It must hold that any SMT for the reduced instance can be mapped to an SMT of the original instance. Although reduced instances are usually easier to solve, this property is not guaranteed.

Reductions can be applied multiple times and in arbitrary order. The resulting SMT is then mapped back in the reverse order. It is common to apply reductions either until the graph cannot be reduced further, or the amount of reducible graph components fall below a defined threshold.

In this chapter, we present the theory behind the reductions used in our solver. We start by introducing the concept of Steiner distance, which is central for subsequent reductions. Afterwards, we explore the reductions grouped by type. We distinguish between three types of reductions:

1. Exclusions: These tests identify graph components that can be removed.
2. Inclusions: Inclusion reductions identify components that are guaranteed to be in an SMT.
3. Bound based: The idea of bounds has been discussed in the previous chapter. Whenever the lower bound for a component exceeds the upper bound for the instance, the component can be removed.

It is common that a reduction introduces new edges. Whenever an edge that already exists is added, we retain the minimum of the existing edge cost and the new edge cost. For the aforementioned reverse transformations of an SMT it is important to identify edges. For this chapter we assume implicitly that an edge is identified by its endpoints and costs. For example: A reduction adds edge e with cost 5 to G. An SMT T is computed for the reduced instance. The reduction checks if e is T. When doing so it also validates, that $c(e) = 5$, otherwise $e \notin E[T]$. This is important as reductions may change costs multiple times.

3.3.1 Bottleneck Steiner distance

Throughout this thesis we use the distance measure d. While very useful, it is not specific to the ST. In contrast the *bottleneck Steiner distance* utilizes instance specific properties. It has proven very useful for reductions.

In order to define *bottleneck Steiner distance* we first have to define *Steiner distance*. Given a path P between two vertices v_i and v_j. The path can be transformed into one or more elementary paths by splitting it at intermediate terminals. The maximum length among these elementary paths is the *Steiner distance*. The *bottleneck Steiner distance* between vertices v_i and v_j, denoted $s(v_i, v_j)$, is the minimum *Steiner distance* among all paths connecting v_i and v_j. The *restricted bottleneck Steiner distance* $\overline{s}(v_i, v_j)$ is the minimum Steiner distance among all paths connecting v_i and v_j without using the edge e_{ij}. Note that if $e_{ij} \notin E$ then $\overline{s}(v_i, v_j) = s(v_i, v_j)$ [DV89].

To give an intuition for this measurement: Assuming that the graph is a road network and that terminals are gas stations. Any car driving from v_i to v_j needs to be able to drive distance $s(v_i, v_j)$ without refueling [KM96]. Given that the road from v_i to v_j is blocked, the car has to be able to drive distance $\overline{s}(v_i, v_j)$.

Algorithm 3.8 shows the classical way to calculate s. If run for all vertices it yields the exact values for all possible vertex pairs. The algorithm runs in $\mathcal{O}(nk)$ per vertex [HRW92b]. Note that it requires d to be calculated for all vertex pairs. Given that the measurement is central to many reductions and becomes outdated if the graph changes, the runtime is infeasible for larger instances.

Algorithm 3.8: Algorithm to find $s(v_i, v_j)$ [HRW92b, Chapter 2.3.1]

```
   Data: A graph $G = (V, E, c)$ and a vertex $v_i \in V$
   Result: $s(v_i, v_j)$ for all $v_j \in V$
1  $s(v_i, v_j) \leftarrow d(v_i, v_j)$ for all $v_j \in V$
2  $L \leftarrow \{v_i\}$
3  while $R \nsubseteq L$ do
4      Choose $z_k \in R \setminus L$ minimizing $s(v_i, z_k)$
5      $L \leftarrow L \cup \{z_k\}$
6      forall $v_j \in V \setminus L$ do
7          $s(v_i, v_j) \leftarrow \min\{s(v_i, v_j), \max\{s(v_i, z_k), d(z_k, v_j)\}\}$
8          if $v_j \notin R$ and $s(v_i, v_j) \leq s(v_i, z_k)$ then
9              $L \leftarrow L \cup \{v_j\}$
10     end
11 end
12 return $s$
```

s is usually tested against and upper bound. We can cut down on computation time, if we use an upper bound for s and the tests stay valid. We can calculate s for pairs of terminals efficiently using an MST. If we additionally know the distance to the r closest terminals for every vertex, we can calculate such an upper bound. Let $z_{i,j}$ be the j-closest terminal to v_i and $v_i, v_j \in V$, the upper bound $\hat{s}$ can be calculated as follows:

$$\hat{s}(v_i, v_j) = \min_{a,b \in \{1,\ldots,r\}} \{\max\{d(v_i, z_{i,a}), s(z_{i,a}, z_{j,b}), d(z_{j,b}, v_j)\}\}$$

We can calculate the r closest terminals by modifying *Dijkstra's algorithm* (see Chapter 3.2.1). The calculation takes time $\mathcal{O}(m + n \log n)$ [Dui93, Pol03].

We can also use *Dijkstra's algorithm* to directly approximate s. Given two vertices v_i and v_j, a run of the algorithm produces the mapping l. The mapping contains the distances from v_i to any other vertex the algorithm encountered before finding v_j. Running the algorithm twice, once from v_i to v_j yields two such mappings l_1 and l_2. Let $R' = R \setminus \{v_i, v_j\}$ and $d_z = \min_{z_k \in R'} \min\{l_1(z_k), l_2(z_k)\}$. We can approximate $s(v_i, v_j)$ by $\min\{l_1(v_j), d_z\}$. In order to speed up competition we add some modifications to the algorithm:

1. The algorithm does not proceed from terminals other than v_i and v_j.
2. If we test against a value l_w we stop if we exceed this value.
3. We define a limit l_d and stop after iterating over a total of l_d edges.
4. During the second run, the algorithm does not proceed from vertices encountered in the first run.

Note that although the modifications speed up the approximation, it is not guaranteed to find the desired path. In this case we can construct a path by using vertices found in both runs. Let V' be the set of all vertices in both l_1 and l_2. The length of the shortest path found by the algorithm is therefore $d_v = \min\{l_1(v_j), \min_{v_k \in V'}(l_1(v_k) + l_2(v_k))\}$. We can now approximate $s(v_i, v_j)$ by $\min\{d_v, d_z\}$ [Reh15]. We refer to this measure as $s_c(v_i, v_j)$.

Although the algorithm does not always return a value, it is very useful to test against a known weight, for example the weight of an edge. Note that if the use of edge e_{ij} is prohibited, this approximates the *restricted bottleneck Steiner distance.*

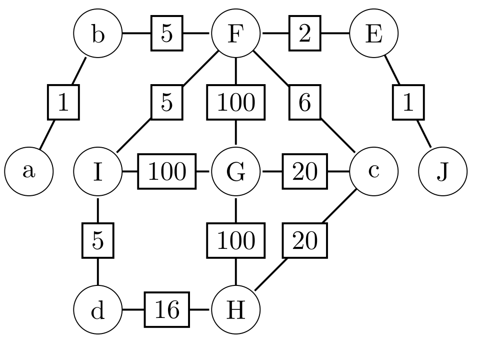


Figure 3.4: The running example graph

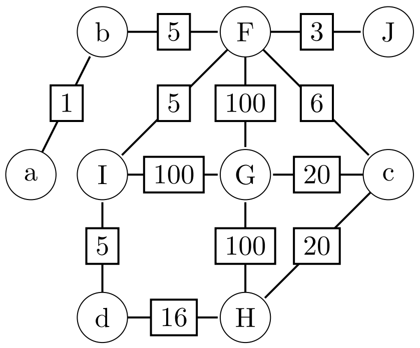


Figure 3.5: Graph after applying the degree 2 reduction on vertex E

3.3.2 Exclusions

Exclusion try to identify areas of the graph that can be removed. For this end, we need to prove that for any Steiner tree T containing a specific graph component, there exists another Steiner tree of equal or lower cost without it. In this case we can remove the tested component.

In this chapter we mainly present the tests used in the reductions. The transformation is usually the removal of the tested vertex or edge. Since we do not need this part in an SMT, no transformation on the solution is required afterwards

Non-Terminals of Degree 0 or 1

This is a simple test that allows to efficiently check for removable vertices. Although it is generalized in subsequent tests, the fast runtime makes it invaluable.

Lemma 8. *[HRW92b, Chapter 2.1.1] No vertex $v_i \in V, v_i \notin R$ of degree 1 or 0 can be in a Steiner minimal tree.*

Assuming that v_i has degree one and e_{ij} is its incident edge: If a Steiner tree T contains v_i, it is a leaf node. $|T|$ could then be reduced by removing e_{ij} and T would remain a Steiner tree. Although the graph is usually connected, some reductions may disconnect the graph. The lemma allows us to remove disconnected non-terminals.

Example 3.3.1. In Figure 3.4 vertex J can be safely removed. ■

Non-Terminals of Degree 2

Instead of testing for components that cannot be part of an SMT, this test identifies graph regions that do not have an impact on the construction of an SMT.

Lemma 9. *[HRW92b, Chapter 2.2.2] Given a vertex $v_i \in V, v_i \notin R$ of degree two. Let $e_{ij}, e_{i\ell}$ be the two incident edges. Let G' be a graph obtained by removing v_i and adding edge $e_{j\ell}$ with cost $c(e_{ij}) + c(e_{i\ell})$, then $|smt_G(R)| = |smt_{G'}(R)|$. If G already contains $e_{j\ell}$ with $c(e_{ij}) + c(e_{i\ell}) \geq c(e_{j\ell})$, v_i can be removed.*

Whenever a Steiner vertex v_i of degree two is in an SMT, both neighbors have to be as well. Otherwise we would have a non-terminal leaf. We can therefore replace v_i by an edge connecting its neighbors. Should this edge be in the resulting SMT, we have to reintroduce v_i and replace the edge by the original two edges it represents [HRW92b].

Example 3.3.2. In Figure 3.4 vertex E has degree 2 and has been removed in Figure 3.5. ■

Steiner Distance

This test makes use of the previously introduced *bottleneck Steiner distance* to identify removable edges.

Lemma 10. *[DV89, Chapter 3] Any edge e_{ij} with $c(e_{ij}) > s(v_i, v_j)$ can be removed from G. The same holds for any edge e_{ij} such that $c(e_{ij}) > \overline{s}(v_i, v_j)$ [Pol03, Dan03].*

This test is usually referred to by different names. The test using the exact values is usually referred to as Paths with many Terminals (PTm). We also use this name if we use $\hat{s}$ instead of s. We use the name Steiner Distance Circuit (SDC), if we use s_c [Reh15].

Example 3.3.3. Figure 3.6 marks the edges identified by PTm. Edges e_{FG}, e_{GH} and e_{GI} can be removed as their end vertices can be reached more cheaply. e_{cH} can be removed because the Steiner distance of path $c\, e_{cF}\, F\, e_{FI}\, I\, e_{Id}\, d\, e_{dH}\, H$ is 16. ■

Long Edges

This test complements the previous test by identifying removable edges missed by the PTm test.

Lemma 11. *[Pol03, Lemma 20] Let $M_{D_G(R)}$ be the MST for the distance network defined for R. Furthermore, let c_{max} be the maximum cost among all edges in $M_{D_G(R)}$. Any edge e_{ij} with $c(e_{ij}) > c_{max}$ can be removed from G.*

Example 3.3.4. In the graph displayed in Figure 3.4, we can construct a spanning tree such that c_{max} is 15. We can connect b to a with cost 1, to c with cost 11 and to d with cost 15. This allows us the eliminate edges e_{cG} and e_{dH} not identified by the PTm test.■

Non-Terminals of Degree k (NTD_k)

This test is often referred to as *Bottleneck Degree m Test (BD_m)* [DV89]. As the name suggests, it is a generalization of the NTD_2 test. The goal here is to identify Steiner vertices that do not have degree higher than two in any SMT. We define a distance network using s the following way: $D'(V) := (V, V \times V, s)$.

Lemma 12. *[DV89, Chapter 4] For a $v_i \in V \setminus R$ with $|\delta(v_i)| \geq 3$ exists an SMT where v_i has maximum degree two if: For every $A \subseteq \delta(v_i)$ with $|A| \geq 3$, it holds that*

$$\sum_{e_{ij} \in A} c(e_{ij}) \geq |D'(\{v_j \mid e_{ij} \in A\})|$$

Given such a vertex v_i and its incident vertices $V_A = \{v_j \mid e_{ij} \in \delta(v_i)\}$. Similar to the NTD_2 test, we can remove v_i and replace it by edges connecting its neighbors. We add edges $\{(v_j, v_\ell) \in V_A \times V_A \mid v_j \neq v_\ell\}$ with costs $c(e_{j\ell}) = c(e_{ij}) + c(e_{i\ell})$ for $v_j, v_\ell \in V_A, v_j \neq v_\ell$.

Note that this does not really reduce the graph, as it introduces numerous new edges. In conjunction with the previously discussed tests, many of the new edges can be eliminated. This usually leads to an overall reduction in the number of vertices and edges.

Whenever an SMT T uses one of the edges, v_i is reintroduced and the edge replaced by the original two edges, similar to the NTD_2 reduction.

Example 3.3.5. Several vertices in Figure 3.4 can be reduced using the NTD_k reduction. For vertex I the sum of incident edge costs is 110. An MST among its neighbors costs 30, because $s(I, G) = 20$. Furthermore, for vertex H the incident edge sum is 136 and the cost of an MST is 56. Although vertex G has degree 4, it can be reduced as well. An MST among its neighbors costs 32 while any subset of three or more incident edges costs at least 220. Figure 3.7 shows the graph after reducing vertex I. ■

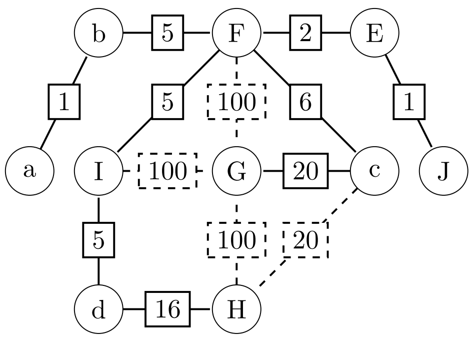


Figure 3.6: Edges removable by the Steiner distance test are dotted

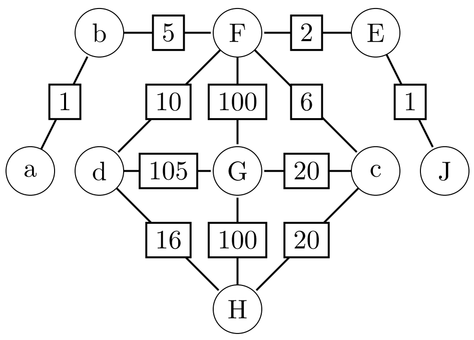


Figure 3.7: Graph after applying the NTD_k reduction on vertex I

3.3.3 Inclusions

The previously discussed reductions allow us to eliminate parts of the graph. Inclusions try the opposite. We try to identify parts of the graph that are part of at least one SMT. This can be achieved by showing that for a Steiner tree containing the component, no other Steiner tree of equal or lower weight exists, that does not contain the component. These tests are also the only way to reduce the number of terminals.

Central to inclusion reductions is the concept of contraction. Any edge e_{ij} that has been identified to be in an SMT can be contracted by performing the following transformation:

- For each edge $e_{j\ell} \in \delta(v_j) \setminus \{e_{ij}\}$ add edge $e_{i\ell}$ with cost $c(e_{j\ell})$.
- If $v_j \in R$ then $R = R \cup \{v_i\}$.
- Remove vertex v_j and edges $\delta(v_j)$.

Given an SMT T for the reduced graph, we transform it into an SMT for the unreduced graph:

- Add vertex v_j and edge e_{ij}.
- If any of the previously added edges $e_{i\ell}$ are in T, replace them by $e_{j\ell}$.

Next, we discuss different tests that allow us to identify such edges.

Terminals of Degree 1

This simple test allows us to quickly identify contractible edges. Although these edges are also identified by subsequent tests, this one runs faster and can therefore be performed more often.

Lemma 13. *[HRW92b, Chapter 2.2.1] For any terminal $z_i \in R$ of degree one, the incident edge $\{z_i, v_j\}$ is in every SMT.*

Note that we assume that the edge costs are positive. The above lemma would also hold for edges of cost 0, as they would not increase the cost of the graph.

Example 3.3.6. In Figure 3.4 terminal a can be contracted into b. Subsequently b can be contracted into F making it a terminal. ■

Minimum Terminal Edge

The following lemma is a generalization of the previous lemma and allows for more edges to be contracted. The test is also called *Short-Terminal-to-Terminal Edges (STTE)* [HRW92b].

Lemma 14. *[HRW92b, Chapter 2.2.2] If the nearest vertex incident to a terminal z_i is also a terminal z_j, the edge $\{z_i, z_j\}$ is in at least one SMT .*

Example 3.3.7. In Figure 3.4 a is the nearest neighbor of b. Therefore {a, b} can be contracted. Even if a were connected to other vertices, since their weight cannot be lower than 1 and therefore the lemma would still be applicable. ■

Nearest Vertex

This test tries to find contractible edges incident to terminals. The idea is: If the shortest edge is short enough compared to the other edges, it is in an SMT.

Lemma 15. *[HRW92b, Chapter 2.2.3] Let z_i be a terminal with degree at least 2. Let $e' = \{z_i, v_{\ell_1}\}$, $e'' = \{z_i, v_{\ell_2}\}$ be a shortest and second shortest edge incident to z_i. There exists at least one SMT containing e', if there exists a terminal z_j such that $z_j \neq z_i$ and*

$$c(e'') \geq c(e') + d(v_{\ell_1}, z_j)$$

In case the test fails, we can extend it. If we can show the test condition holds for any edge that connects v_{ℓ_2}, this is equal to showing it for e'':

Lemma 16. *[Reh15, Lemma 8] Given the definitions of the previous lemma, there exists at least one SMT containing e' if:*

$$c(e) \geq c(e') + d(v_{\ell_1}, z_j) \quad \textit{for all } e \in \delta(z_i) \cup \delta(v_{\ell_2}) \setminus \{e', e''\}$$

Executing the test requires computing d which is expensive. It is possible to redefine the test using Voronoi regions to execute it faster. Instead of calculating $d(v_{\ell_1}, z_j)$ we use $d(v_{\ell_1}, base(v_{\ell_1}))$ if $base(v_{\ell_1}) \neq z_i$, otherwise we use the distance to the second closest terminal [Pol03].

Example 3.3.8. Figure 3.8 shows a graph with Voronoi regions indicated via coloring. Three terminals b, c and d have degree at least 2. The reduction can be applied to all these terminals. For terminal d the shortest edge has cost 5 and the second shortest 16. The shortest path to the next terminal is 15. For c the argument is similar and the shortest incident edge of b connects to a terminal. ■

Short Links

This reduction uses the notion of links to contract edges. Links are edges that connect two Voronoi regions, therefore edges such that $e_{ij} \in E$ and $base(v_i) \neq base(v_j)$ [Pol03].

Lemma 17. *[Pol03, Lemma 23] Given a terminal z_i and the two shortest links of its Voronoi region $e_{j_1\ell_1}, e_{j_2\ell_2}$ such that $base(v_{j_1}) = base(v_{j_2}) = z_i$. $e_{j_1\ell_1}$ belongs to at least one SMT if*

$$c(e_{j_2\ell_2}) \geq d(z_i, v_{j_1}) + c(e_{j_1\ell_1}) + d(v_{\ell_1}, base(v_{\ell_1}))$$

Example 3.3.9. Figure 3.8 shows the Voronoi regions of the running example. Edges in black are links. For c the shortest link is {c, F} and the cost of the second shortest link is 20. The path from c to b is of cost 11, therefore {c, F} can be contracted. A similar argument holds for the region of terminal d. ∎

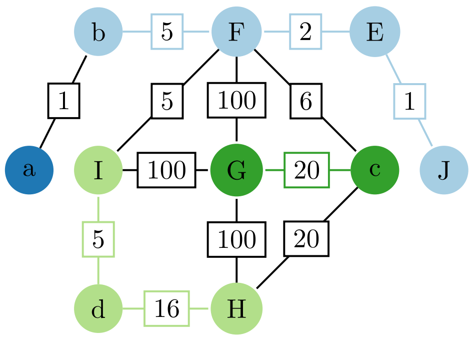


Figure 3.8: Voronoi regions of the graph

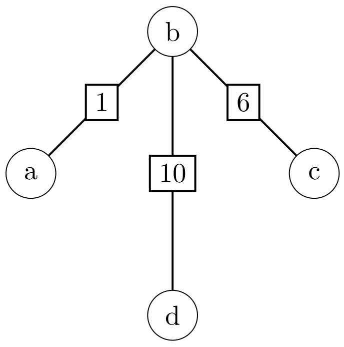


Figure 3.9: The reduced distance network

3.3.4 Bound based

In Chapter 3.1 we discussed approximations and how they can define bounds for our instance. We distinguish between global and local bounds. The approximations discussed so far compute bounds for the whole instance or global bounds. Local bounds give us a limit for the cost of any SMT containing a specific component. Whenever a local lower bound exceeds a global upper bound, we can remove the tested component.

We have already established a method to compute global upper bounds in Chapter 3.1.1. In this chapter, we discuss local lower bounds. The first method we discuss uses Voronoi regions to compute these bounds and the other the already discussed *dual ascent algorithm*.

Voronoi region based bounds

In the following we present several lower bounds for vertices and edges. For this we introduce the function $radius : R \to \mathbb{N}$, returning the length of the shortest path leaving the Voronoi region of a terminal. For convenience it is assumed, that the terminals are ordered in ascending radius value. Furthermore, $z_{v,1}, z_{v,2}, z_{v,3}$ denote the closest, second closest and third closest terminal to the vertex v.

Note that the bounds were originally stated and proved using the normal distance measure d [Pol03] and then restated using restricted distance $\underline{d}$ [Reh15].

The following lemma defines lower bounds that can be established quickly.

Lemma 18. *[Reh15, Lemma 10] Let $v \in V \setminus R$ and T be a Steiner tree containing v:*

$$\underline{d}(v, z_{v,1}) + \underline{d}(v, z_{v,2}) + \sum_{\ell=1}^{k-2} radius(z_\ell)$$

is a lower bound for $|T|$.

The same idea can be applied to obtain a lower bound for edges.

Lemma 19. *[Reh15, Lemma 11] Let $e_{ij} \in E$ and T be a Steiner tree containing e_{ij}:*

$$c(e_{ij}) + \underline{d}(v_i, z_{v_i,1}) + \underline{d}(v_j, z_{v_j,1}) + \sum_{\ell=1}^{k-2} radius(z_\ell)$$

is a lower bound for $|T|$.

Example 3.3.10. Using Figure 3.8 we can see that the two shortest links have cost 1 and 5 yielding a 6 for the radius values. We can now calculate the lower bound for H. The two closest terminals are d and c with a total cost of 36. The resulting upper bound is 42.

In a similar fashion we can calculate a lower bound for edge {d, H}. The distance from d to the nearest terminal is 0 and from H to the nearest terminal 16. Together with the radius value and edge cost, this yields a lower bound of 38. ■

We can also establish bounds based on the degree of the vertex. Using this idea, we can define a bound based test similar to the NTD_k test already discussed.

Lemma 20. *[Reh15, Lemma 13] Let $v \in V \setminus R$ and T be a Steiner tree containing v. If v has degree at least three in T, then*

$$\underline{d}(v, z_{v,1}) + \underline{d}(v, z_{v,2}) + \underline{d}(v, z_{v,3}) + \sum_{\ell=1}^{k-3} radius(z_\ell)$$

is a lower bound for $|T|$.

The transformation for any identified vertex is the same as in the NTD_k test, see 3.3.2 for details.

Example 3.3.11. The shortest link has cost 1. The three closest terminals for vertex F have distances 5, 6 and 10. Therefore, the lower bound for a Steiner tree where F has degree greater than two is 22. ■

In Chapter 3.1.2 we discussed a lower bound based on 1-trees in the distance network. We can use this idea and apply it to Voronoi regions.

Lemma 21. *[Reh15, Lemma 14] Let*

$$\begin{aligned} E' &= \{\{base(v_i), base(v_j)\} \mid e_{ij} \in E, base(v_i) \neq base(v_j)\} \\ c'(z_a, z_b) &= \min\{\min\{\underline{d}(z_a, v_i), \underline{d}(z_b, v_j)\} + c(e_{ij}) \mid v_i \in N(z_a), v_j \in N(z_b)\} \\ G' &= (R, E', c') \end{aligned}$$

The weight of an MST for G' is a lower bound for the weight of any Steiner tree for (G, R).

This lemma can be used to define a lower bound for edges and vertices. Let w' be the weight of the spanning tree for G' minus the length of its longest edge:

- Let $v \in V$, $w' + \underline{d}(v, z_{v,1}) + \underline{d}(v, z_{v,2})$ is a lower bound for a Steiner tree containing v as a Steiner node.
- Let $e_{ij} \in E$, $w' + \underline{d}(v_i, z_{v_i,1}) + c(e_{ij}) + \underline{d}(v_j, z_{v_j,1})$ is a lower bound for a Steiner tree containing e_{ij}.

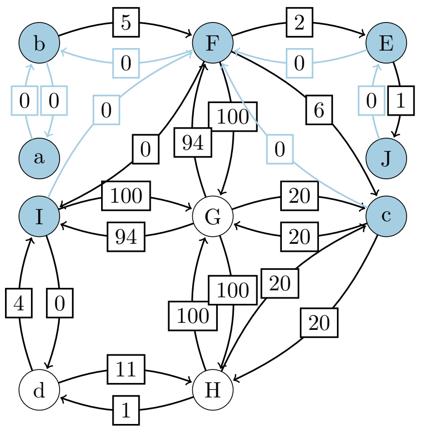


Figure 3.10: Graph after running dual ascent. $\tilde{w} = 22$

Example 3.3.12. Figure 3.9 shows the reduced distance network G' for our running example. The graph is already a minimal spanning tree and has cost 17 and without the longest edge it is 7.

We again calculate a lower bound for H. The sum of costs for the two closest terminals is 36. Therefore the lower bound is 43. For the edge {d, H} the calculation yields a lower bound of 39. ■

Dual ascent

We already discussed the dual ascent algorithm in Chapter 3.1.3. Besides a lower bound $\tilde{w}$, the algorithm also calculates reduced arc costs $\tilde{c}$. We discuss how to use these results to establish local lower bounds. r denotes the terminal designated as root.

Lemma 22. *[Pol03, Chapter 3.4.2] Let $\tilde{w}$ be the lower bound and $\tilde{c}$ be the reduced edge costs after a dual ascent run. Furthermore, let $\tilde{d}$ be the distance function using the reduced edge costs and directed paths. Given a vertex $v_i \in V$,*

$$\tilde{w} + \tilde{d}(r, v_i) + \min_{z_j \in R'} \tilde{d}(v_i, z_j)$$

is a lower bound on any Steiner tree containing v_i.

A similar bound can be derived for any edge $e_{ij} \in E$. Let

$$c_i = \tilde{d}(r, v_i) + \tilde{c}(a_{ij}) + \min_{z_\ell \in R'} \tilde{d}(v_j, z_\ell)$$
$$c_j = \tilde{d}(r, v_j) + \tilde{c}(a_{ji}) + \min_{z_\ell \in R'} \tilde{d}(v_i, z_\ell)$$

$\tilde{w} + \min\{c_i, c_j\}$ is a lower bound for any Steiner tree containing e_{ij}.

Note that the lower bound for an edge e_{ij} is calculated by taking the minimum over the corresponding arcs a_{ij}, a_{ji}. Furthermore, the lower bound on terminals is always 0.

The tests can be sped up, if a Voronoi partitioning based on the reduced costs is created first.

Example 3.3.13. Figure 3.10 shows the running example after a run of dual ascent using c as a root. The resulting lower bound is 22. We can now calculate the lower bound for H. The shortest path from c to H using the 0 edges has cost 11. The shortest path from H to a terminal is 1. Together with the graph's lower bound this yields 34 as the lower bound for H. Note that a Steiner tree using H has minimum weight 43 under the assumption that it must not be a leaf.

Similarly we can calculate the lower bound for the edge {d, H}. We have to calculate both variants: c to d and H to closest terminal as well as c to H and d to closest terminal. The shortest path from c to d hast cost 0 and the closest terminal to H is d with distance 1, totaling at 12. The shortest path from c to H has cost 11 and the distance from d to the closest terminal is 0, also totalling at 12. Together with the graph's lower bound this yields a lower bound of 34 for the edge. ■

3.3.5 Inaccurate Data

Many reductions use the same data: distances, Voronoi regions, Steiner distances, and upper bounds. In order to avoid expensive recalculations, the data is usually stored. As reductions remove part of the graph, they can cause the data to be outdated. Some reductions produce correct results even if stale data is used, although they might not be as effective [HRW92b]. It is therefore useful to know how different reductions affect data and when the data needs to be refreshed.

Inclusion tests contract edges, thereby usually decreasing distances. Therefore distances, Voronoi regions, Steiner distances and upper bounds decrease. As the tests compare against an upper bound, they remain correct if the stored distances are higher [HRW92b].

Exclusion tests remove part of the graph and can cause distances to increase. Voronoi regions and upper bounds stay the same. Steiner distances usually change as well, except for the long edges and SDC test by design.

In this sense the NTD_k test is not a real exclusion test as it restructures the graph more than remove parts. It therefore does not change distances, but the upper bound may change.

Tests that compare against a lower bound stay correct even if the stored data is higher. This affects the SDC, long edge and NTD_k reduction. Conversely, tests that compare against an upper bound stay correct even if the data is lower than in reality. As is the case for the Voronoi reductions.

The dual ascent reduction as well as the simpler non-terminal and terminal tests tests do not rely on stored data.

This concludes our discussion of reductions. In the next and final theoretical part, we look at methods to improve our approximations.

3.4 Improving Steiner Trees

Every graph that is connected, contains all terminals and uses only edges from the input graph is a Steiner tree. Furthermore, every Steiner tree is an upper bound for the weight of an SMT. This implies two things:

1. We can find Steiner trees in a subgraph of G as long as all terminals are in one connected component.
2. Given a Steiner tree T, we can generate new Steiner trees by replacing parts of it.

In this chapter, we use this ideas to improve our upper bounds. Although many approximations for Steiner trees exist [HRW92c], we only use RSPH in our work. The methods we present in this chapter are nonetheless applicable to all approximation methods that generate Steiner trees.

3.4.1 Alternative Graphs

The idea behind this strategy is, to reduce the input graph in the hope that the approximation finds a better bound using the reduced graph. Since we compute an upper bound it suffices to compute Steiner trees. It is therefore not necessary for the reduced graph to contain an SMT for the original graph. We discuss different methods to find subgraphs of G that may improve approximation results.

Dual Ascent

We discussed this algorithm in Chapter 3.1.3. It iteratively creates larger and larger components, until it has established one component that contains all terminals. It is straightforward to use the final component as an input for our approximations. We therefore create a graph that contains all edges e_{ij} such that $\tilde{c}(a_{ij}) = 0$ or $\tilde{c}(a_{ji}) = 0$. While the dual ascent algorithm guarantees that this graph contains a Steiner tree, it may not contain an SMT [Pol03].

Pruning

The concept of comparing lower to upper bounds has been discussed several times. In Chapters 3.3.4 and 3.3.4 we discussed methods that use this concept to remove components of the graph.

So far we always computed a valid upper bound. Using an invalid upper bound, i.e. one that is lower than the optimal value, would result in potentially sub-optimal results. For approximations the validity is not a concern. As long as we arrive at a Steiner tree, we have the desired result. We can therefore assume any upper bound, reduce the graph using the bound and finally run the approximation algorithm. Note that this is similar to guessing the value of $|smt_G(R)|$.

A common way to establish the bound is by defining a reduction goal. We therefore choose the bound, so that a fixed percentage of edges or vertices are eliminated. If we want to eliminate 10 % of the edges and the graph has 100 edges, we use the 90th highest weight to compute our upper bound [Pol03].

We can extend this methods to include other reductions as well. After pruning the graph, we run other reductions, shrinking the graph even more. We can also run the pruning in several passes. In each pass we lower the upper bound to eliminate more edges or vertices, run the reductions and try to find a Steiner tree. We can also prohibit the reduction of edges from a known Steiner tree to guarantee the existence of at least one solution. We keep the smallest Steiner tree over all passes as an upper bound [Pol03].

Recombination

It is common to calculate several Steiner trees. RSPH alone is performed in several passes, each of the improvement strategies presented here may produce distinct Steiner trees. Furthermore, reductions are performed in passes, often recalculating upper bounds. We can collect these trees and *recombine* them, in a hope to find a better one.

The general idea is, to select a certain number of Steiner trees from a pool. A new graph is created by combining the vertices and edges of the chosen trees. Reductions and pruning can then be applied to the resulting graph [Pol03, Reh15]. Finally a Steiner tree is approximated using this graph.

It is also possible to adjust the edge weights in a way that reflects in how many Steiner trees they occur. Different strategies have been suggested, either penalizing high-frequency edges with higher costs to achieve diversity or lowering their costs to intensify their usage [RUW00, Reh15]. The weights are adapted before a Steiner tree is computed and restored before any reductions or local searches are applied [Reh15].

Parameters of this strategy are the size of the pool, the selection strategy defining how many and which Steiner trees to choose, as well as how the edge weights are adapted.

3.4.2 Local search

Approximated solutions can often be improved by applying small changes. Given a Steiner tree we try to replace components in the hope of obtaining a new Steiner tree with lower weight. All methods described here run in $\mathcal{O}(m \log n)$ [UW12].

For the rest of this section we assume we are optimizing a Steiner tree $T = (V', E', c)$. Furthermore, a *key-vertex* is a non-terminal in V' of degree at least three and a *crucial vertex* is either a key-vertex or a terminal. Let C denote the set of crucial vertices.

Vertex Insertion

This method tries to find a better solution by adding a new vertex. Given a vertex $v_i \in V \setminus V'$, and $A = \{e_{ij} \in \delta(v_i) \mid v_j \in V'\}$. Therefore, A contains all edges that directly connect v_i to T. If $|A| > 1$ we can try to improve T using the following strategy. We add the first edge from A to T, thereby adding v_i to T. For every other edge $e_{ij} \in A$ we find the most expensive edge e' on the path from v_i to v_j. If $c(e') > c(e_{ij})$ remove e' and add e_{ij}. This effectively computes a minimal spanning tree for $V' \cup \{v_i\}$ [UW12]. If the resulting tree's weight is lower than T we keep the new tree as T, otherwise we restore T. The search is performed for all $v_i \in V \setminus V'$[2].

Key-Path Exchange

A key-path is a path in T that connects two crucial vertices. This method removes a key-path and tries to reconnect the two resulting components. Whenever we find a cheaper path, we keep the new graph, otherwise we restore T. A straightforward way to find a new path, is to use *Dijkstra's algorithm* [Voß92, VSA96][3].

The algorithm's runtime can be improved using Voronoi partitioning. We partition the graph into regions defined by the vertices in T and store the links, ordered by the length of the path they represent. Whenever we delete a path, we repair the partitioning and keep track of the new links. Finding an alternative path now only requires to check the prioritized links [UW12].

[2] The original source was not available, [UW12] refers to [Min90]

[3] [VSA96] also references [Dow91] which was not available

Key-Vertex Elimination

This method tries to find a smaller Steiner tree after the removal of a key-vertex. For each key-vertex v_i, we try to remove the vertex. This splits T into three or more components. We then create a distance network, where each component is represented by a vertex. The edges have cost equal to the shortest path between these components in G. If the MST of this distance network has smaller cost than T, we can replace v_i in T by adding the paths represented by the MST's edges [DV97].

Calculating the distance network and its MST is expensive. As with the previous method we can use Voronoi partitioning to speed up the calculation. Keeping track of the links allows us to create the distance network without explicitly calculating the paths [UW12].

This concludes the discussion of the theoretical background. We now turn to its practical applications.

CHAPTER 4

Implementation

In the previous chapter, we laid out the theoretical foundation for our solver. In this chapter, we discuss our implementation[1] for PACE. With the theoretical knowledge in mind, we focus on solver specific details and findings that are not derivable from the presented theory.

The structure of this chapter is similar to that of the previous one. We first present how we calculate the bounds in our solver. Next, we lay out the details of our solving and reduction modules. In the last part we discuss weaknesses of our implementation

4.1 Approximations

In our implementation we use bounds extensively. During the reduction phase we need lower and upper bounds to determine if components in the graph can be part in an SMT. During the solving phase the upper bound is not as important and primarily serves to reduce memory consumption. A lower bound is used as a guiding heuristic (this is discussed in detail in Chapter 4.2).

Although the algorithms for the heuristics are in principle simple, the implementations are comparatively extensive. Their implementation also includes optimizations and local searches, which add considerable complexity. This extra effort is justified as these heuristics together with the solver code are the most called modules. Small improvements in runtime or tightness can make a big difference.

We first discuss RSPH, our main method of obtaining an upper bound. Afterwards, we present our approach to calculate lower bounds using *dual ascent*.

[1]The source code of our solver can be found at `https://github.com/ASchidler/pace17`

4.1.1 RSPH

Whenever an upper bound is required, this heuristic is called. During the reduction phase, all bound based reductions (see Chapter 3.3.4) use it. Furthermore, *dual ascent* uses it on subgraphs on G during pruning and recombination (see Chapter 3.1.3).

The algorithm is usually limited to 20 roots. Since dual ascent calls it several times during one run, it uses a lower limit of 10 roots, to reduce runtime.

Shortest paths

The implementation of this heuristic is rather straightforward. In every iteration we choose a terminal and add the shortest path to our solution. In a first implementation we used precalculated distances to determine the closest terminal. In each iteration we ran *Dijkstra's algorithm* to find the shortest path.

This worked as a proof of concept, but was inefficient. We implemented it again using the method suggested in the literature and discussed in Chapter 3.1.1. We run Dijkstra's algorithm from the root. As soon as we find another terminal, we add the shortest path. Furthermore, we change the distances for all added vertices to 0 and add them to the queue. This "warm starts" the algorithm. We run it again until we find terminal, add the path and repeat the whole process until all terminals have been added.

During the first tests, the results of the two versions differed, sometimes significantly. This was surprising, as they were supposed compute the same Steiner tree. Close examination showed that they chose different shortest paths. This led to different connection points for subsequent terminals and subsequently more differences. The significant difference turned out to be the direction in which the algorithm is run. In the first version it started at the terminal and searched the tree, the second version searched in the opposite direction.

Since the algorithm is greedy, it prefers shorter edges first. Once it reaches a vertex with minimal cost, the path is never overridden in case of tied costs. We added an additional value that acted as a tie breaker, in case two elements have equal cost. This caused variations similar to the ones observed before. Although we tried different tie breaking strategies, we could not find one that achieved overall better results. The tie breaking strategies are:

1. The number of edges in the path. This prefers paths with many edges, providing more possible connection points.
2. An increasing multiplier for the path cost. This prefers paths that have fewer edges and cheaper edges at the end.
3. A decreasing multiplier for the path costs. This prefers paths that have more edges and cheaper edges at the start.

Root selection

In every RSPH run the algorithm chooses several roots. The result from the best root is then used as an upper bound. For larger instances it becomes infeasible to try every possible root in every run. We therefore store the roots used in the previous run sorted by increasing result. For the next run we use half of them and fill the other half with terminals not used before.

4.1.2 Dual Ascent

As described in Chapter 3.1.3, we use this algorithm to calculate both lower and upper bounds. The associated reduction is the only one that does not run in $\mathcal{O}(m + n \log n)$. We use it nonetheless, as its effectiveness usually justifies the extra runtime.

Terminal selection

In every iteration of dual ascent we have to select an active terminal. The choice of this terminal impacts the result. A good selection strategy can therefore tighten the computed bounds.

We implemented dual ascent three times with slight variations. The original goal was to improve the existing implementation. Tests showed that the new implementations were not strictly better. Each implementation performs best for some instances. All implementations use the number of incoming edges as the deciding metric when choosing the next component. The main difference is, when the incoming edges are counted:

1. The first implementation keeps track of the components and uses the incoming edge count established before the edge weights are reduced. This is the slowest implementation, but produces the overall best results.
2. The second implementation counts the incoming edges while it reduces the edge weights. It is the fastest version, but produces the overall worst bounds.
3. The last implementation counts the incoming edges after all updates have been performed. This method is exact, but slow. The runtime and the quality of the approximation is between the other two implementations.

When used for reduction purposes, we alternate between implementation one and two. Therefore, half of the runs are calculated using one implementation and half using the other. If used as a guiding heuristic, we use either the second implementation or the third, as the first one is too slow.

Upper bounds

In order to find better upper bounds, we try to find new subgraphs of G that in conjunction with RSPH produce better bounds. For this end we use three methods:

1. *Dual-Ascent-Merge* is based on combining several dual ascent graphs.
2. *Prune-Ascent* tries to use pruning on a dual ascent graph.

3. *Solution-Merge* combines several Steiner trees generated by the previous methods.

Dual-Ascent-Merge uses the set of all graphs that have been generated by the dual ascent runs. It is run several times with a changing subset of available dual ascent graphs. From each graph the zero cost paths from root to each terminal are calculated and added to an auxiliary graph using the original edge costs. Furthermore, the cost of each edge is increased by $(1 + l - o) * 100$ where l are the number of dual ascent graphs used and o how often the edge occurred in a zero cost path. Therefore edges that occur more often get priority. On this auxiliary graph the shortest path heuristic is run with ten roots and local searches are performed after the original edge weights have been restored.

Prune-Ascent only uses one dual ascent graph at a time. An auxiliary graph is created with all edges that have weight zero in the dual ascent graph. The original edge weights are used for this graph. In three iterations this graph is reduced and for each iteration RSPH with ten roots is run. In each iteration non-bound-based reductions are applied to the graph in three passes. Afterwards bound based reductions are run and an upper bound is chosen in a way that at least 10% of vertices are eliminated. All vertices that are in the current best upper bound are protected against removal. For the implementation several other options were considered and decided against:

- The bound could be chosen in a way that eliminates a certain amount of edges instead of vertices. This did not produce better bounds.
- Besides eliminating vertices using the new bound, edges could be eliminated. This worsened the bound for sparse graphs, but improved it for dense graphs. As it did not improve it enough for the challenge, the idea was discarded.
- The run could be performed on a copy of the main graph instead of an auxiliary graph. As the main graph is usually larger, this took considerably more computation time, while producing inferior results.
- Instead of defining the bound using vertices, it could be reduced by a fixed percentage. This amounts to guessing the real optimum. This again produced inferior results compared to the implemented version.
- More pruning iterations could be performed. This produces better upper bounds, but ultimately the improvement did not justify the added computation time.

Solution-Merge is very similar to *Dual-Ascent-Merge*. The only difference is that *Solution-Merge* adds all edges from a set of previous feasible solutions instead of zero length paths.

Of these three methods, *Prune-Ascent* is the most computationally expensive. Each run means three reduction processes, distance and upper bound calculations. Nonetheless it often improves the upper bound especially for dense graphs. Dual-Ascent-Merge was not described as such in literature, but produced good bounds for some instances and was therefore included.

Parameterization

We introduced several parameters that control the behaviour of the reduction. The number of roots is decided, based on size and density of the graph. Furthermore, three parameters, each controlling how often the previously discussed upper bound methods are run, are also adapted to the properties of the graph.

The algorithm runs slow on dense graphs, due to the number of edges to consider. We therefore limit the number of runs and upper bound generations for theses instances. As the runtime also increases with the graph size, we reduce the number of runs as the graph gets bigger.

The roots for the runs are chosen such that half of them are the best from the previous run (if available) and the rest are terminals not used in the previous run.

This concludes our presentation of the approximation implementations. Next, We discuss implementation details about our solving module.

4.2 Solving Algorithm

In our solver module we use the *Dijkstra-Steiner algorithm* as discussed in Chapter 3.2.3. We added several improvements, either not mentioned or not explicitly specified in the original paper [HSV16]. In this chapter, we discuss our adaptions to the algorithm.

We also show benchmark results for the different features we describe. All benchmarks discussed here have been performed on the 200 PACE instances using the same limitations of 30 minutes runtime and 6 GB memory limit. More information on the benchmark environment can be found in Chapter 6. Table 4.1 and Figure 4.1 show the benchmark results. *tuw* is the baseline of our solver with all features enabled. The other results are discussed subsequently. The column *Common Runtime (s)* shows the aggregated runtime over all instances that were solvable with every configuration.

Name	Solved	Runtime (s)	Common Runtime (s)	Memory (GB)
tuw	188	10985.62	7512.30	26.39
tuw-config-store	187	9499.72	7519.85	25.53
tuw-config-root	186	10109.63	7808.71	25.98
tuw-config-da	177	8576.86	8576.86	27.98

Table 4.1: Different solver configuration settings compared.

4.2.1 Root selection

The original algorithm always uses the last terminal as the root for the calculation [HSV16]. The tightness of the bounds computed by RSPH and dual ascent depend a lot on the roots used. This suggests that a good root selection may also improve the

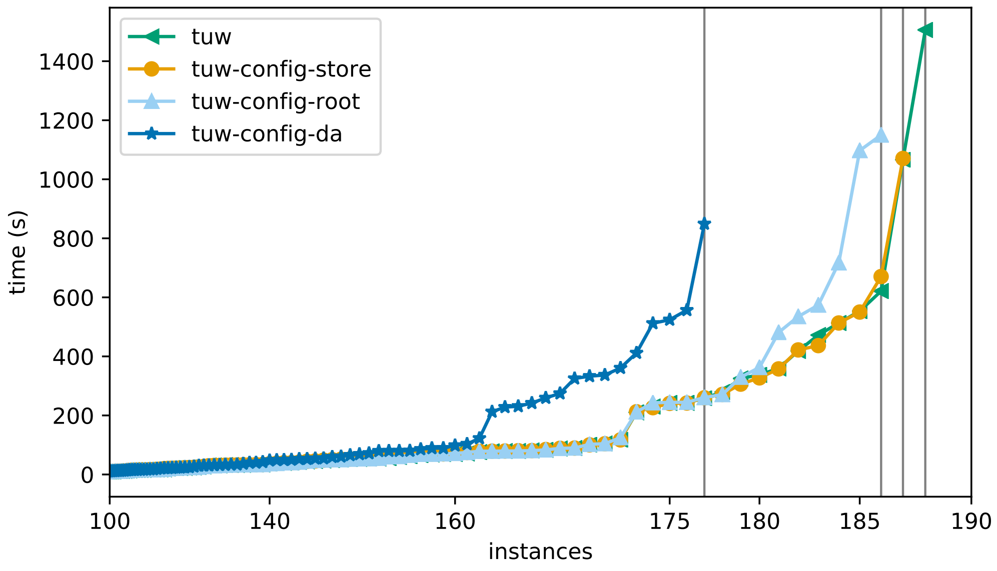


Figure 4.1: Comparison plot of different solver configurations. The markings show the runtime (y-axis) for an instance (x-axis). The instances are sorted by ascending runtime.

performance of the solver. For this end we use the following root selection strategy: Use the best root from the last dual ascent reduction run. If none is available, use the best root from the last RSPH run. If none is available default to the last terminal. Note that, although as long as the reductions have been run, roots are available from both heuristics, due to the nature of the reductions, they may no longer be part of the graph.

In Table 4.1 and Figure 4.1 we compare how the solver performs with (*tuw*) and without (*tuw-config-root*) the root selection strategy. The results show an improvement in both the runtime and the amount of instances solved.

4.2.2 Label Store

In each iteration the current sub-solution is combined with other sub-solutions at the current vertex. This process creates up to 2^k entries per vertex. It is therefore one of the more time consuming operations when dealing with complex instances.

The combination process is performed for all other sub-solutions with a disjoint set of terminals. In an attempt to optimize this step we looked for a data structure that allows us to identify them, without iterating over every entry. The goal was therefore to avoid examining non-disjoint sub-solutions.

We use integers to represent subsets. If the n-th bit is one, then the n-th element of the set is in the subset represented by the integer. For a four element set $\{1, 2, 3, 4\}$, we need four bits to represent its subsets. The number 5 in binary is 0101, therefore

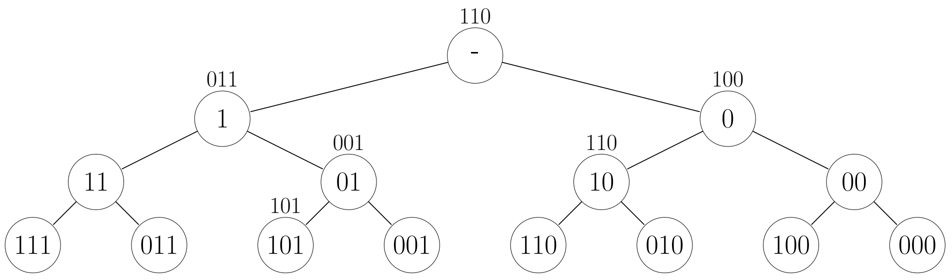


Figure 4.2: The label store with 15 vertices. Numbers in the vertices are the vertex label. Numbers above are the values stored at the vertices.

element 1 and 3 are in the subset represented by 5. It is therefore possible to use binary operators to compare sets. The binary *and* operations performs a set intersection and the *or* operation a set union.

The data structure used is a binary tree with a specific structure. Each vertex in the tree has a label, determined by its position. Starting with the least significant bit, every left branch means the bit at the current level is set, otherwise unset. Figure 4.2 shows the basic structure. The label is drawn inside the vertex.

Whenever a new entry is inserted we search for the first vacant vertex. For this search, the binary value is compared bit by bit. Starting at the root, whenever the bit is 1 we continue our search in the left branch, otherwise in the right. Note that the root accepts any value.

When the store is queried for all stored sets disjoint from I, we examine I bit by bit. We now recursively build a set of sets. We start with the root: $B_0 = \{root\}$, now we construct B_{i+1} considering only children of vertices in B_i. We add all right children, signifying 0 branches. If the bit at position $i+1$ is not set we also add the left children. Let B be the union of all B_i and J the values stored at the vertices in B. We now return all values in J that are indeed disjoint from I. This process cuts down the sets we need to consider by ignoring whole subtrees.

We benchmarked the solver with label store and without. In Table 4.1 and Figure 4.1 the entry *tuw* uses the label store, while the entry *tuw-config-store* uses a simple list. During the development phase, the store made a greater difference. With further improvements in other modules, the effect of the store became smaller. Nonetheless it allows us to solve one instance more.

Example 4.2.1. Figure 4.2 shows the state of the label store after inserting 110, 011, 001, 100, 101 and 100. We now query the store for all disjoint sets of 011. $B_0 = \{-\}$ and $B_1 = \{0\}$ as the first bit of the query set is set. $B_2 = \varnothing$ as the right child of 0 is empty and the second bit is set. We therefore have $B = \{-, 0\}$ and corresponding values 110 and 100. Of these to only 100 is really disjoint. Note that we cut the number of entries to check down from 6 to 2. ∎

4.2.3 Heuristics

While the original paper suggested three different guiding heuristics, we only used one of them [HSV16]. Only the *1-tree* heuristic proved feasible for our purposes, as the others were too slow. The implementation is straight forward, with a cache, so the heuristic value for a subset of vertices can be reused.

While not suggested in the original paper, we also use dual ascent with great success. This heuristic is not consistent and required the changes described in Chapter 3.2.4. In Table 4.1 and Figure 4.1 we compare the solver with and without *dual ascent* as a guiding heuristic. The entry *tuw-config-da* shows how the solver performed without it. The results confirm that this addition has a great impact on our solvers performance.

When used as a guiding heuristic, dual ascent performs only one run using the same root as the solving algorithm. As discussed in Chapter 4.1.2 the algorithm has been implemented multiple times. When the heuristic is called the first time, it calculates the bound, once with each of the fast implementations. For the rest of the solving run it uses the method that produced the better bound.

After calculating the bound, the heuristic value for all vertices is calculated and cached. Subsequent calls for the same terminal subset are answered using the cached values. To limit the memory consumption, only the heuristic values for the last 15 000 queried subsets are kept. Note that the solver chooses the root that produced the best result in the last dual ascent run, improving the bounds produced by the guiding heuristic.

We use *dual ascent* for small sparse graphs and *1-trees* otherwise. While dual ascent provides better bounds, it is too slow to be feasible as a guiding heuristic for larger graphs. Whenever a graph has fewer than 2000 edges and $\frac{m}{n} \leq 3$, we use dual ascent.

4.2.4 Priority Queue

In each iteration the algorithm chooses a vertex-subset tuple that minimizes the known weight of the sub-solution. The tuple is chosen from a collection representing a queue. We use a priority queue to efficiently select the next tuple. Therefore, the tuples are prioritized by weight. Profiling results showed that for complex instances, the priority queue code is one of the most executed parts of the solver. We use two different strategies: *bucket queues* and *d-heaps*.

The entries in *d-heaps* are stored in a tree. Every vertex has d or fewer children. For every vertex the *heap condition* holds: the value of a vertex is lower than that of its children. The minimal element is therefore always the root. We add new elements as a leaf and then swap it with the parent, until the heap condition is restored. When removing the smallest element, we swap the root with a leaf and then remove it. The new root is then swapped with its children until the heap condition holds. Therefore, d-heaps are always balanced and changes can be performed in logarithmic time [EEK17][2].

[2]The original source was not available, [EEK17] refers to [Joh75]

Python provides a *binary heap* or *2-heap* implementation *heapq*, often used for priority queues. We tried the generalized concept *d-heaps*. Here we store *d* different values per vertex instead of two. In theory *d-heaps* should perform better, if the amount of insert statements is significantly higher, than the removal of an element. This behaviour can be expected if the bounds are not very tight, as is the case with complex instances. We compared different values for *d*. Table 4.2 shows the results. The differences are negligible, we use the best performing value of 16.

Name	Solved	Runtime (s)	Memory (GB)
tuw-config-d16	188	10997.23	26.47
tuw-config-d4	188	11004.03	26.54
tuw-config-d2	188	11025.96	26.31
tuw-config-d8	188	11147.63	26.51
tuw-config-d32	188	11252.28	26.49

Table 4.2: Comparison of the performance using different d-heaps as a priority queue

The other approach we use is that of *bucket queues*. The idea is to have one bucket for each value. When we add an element, we put it in the bucket corresponding to its value. This avoids costly management of the queue structure. We have the list of buckets, a pointer to the non-empty bucket of lowest weight and a lookup table. The lookup table allows us to change the weight of existing entries. Whenever we add a new entry, we select the bucket in constant time and add the entry to the lookup table. If the bucket was empty before, we check if we need to update the pointer. To remove an element, we take a random element from the bucket referred to by the pointer. If the bucket is empty, we search for the next non-empty bucket and update the pointer.

This approach avoids managing the elements itself. Adding an element takes two entries and removing and entry is also quick, provided the bucket is not empty. Therefore, the performance depends on the value range covered and how often buckets are empty [MS08a]. The more entries fall into a narrow value range, the better the queue performs. We therefore used a threshold for the upper bound to decide when to use bucket queues and when d-heaps. Table 4.3 shows the comparison using different thresholds. The use of bucket queues has negligible impact on the runtime. We nonetheless use them, as they provide a theoretical benefit and no empirical disadvantage.

This concludes the discussion of our solving module's details. In the next part we show how our implementation manages reductions.

4.3 Reductions

This module runs first and tries to reduce the graph's size. Apart from the bounds already discussed, the implementation of the reductions is straightforward from the

Name	Solved	Runtime (s)	Memory (GB)
tuw-config-bucket-50k	188	11001.43	103.31
tuw-config-bucket-20k	188	11008.14	103.26
tuw-config-bucket-5k	188	11027.34	103.44
tuw-config-bucket-10k	188	11044.97	104.12
tuw-config-bucket-0k	188	11057.56	104.29

Table 4.3: Comparison of the performance using different thresholds for the use of a bucket queue.

theory. Therefore, we do not discuss the reductions in detail. The focus in this chapter is on the configuration of the reduction process.

4.3.1 Organization of Reductions

We perform the reductions in passes. Each pass consists of all reductions, executed in a specific order. Each reduction has a threshold. If the number of edges removed by the reduction is below the threshold, it deactivates itself. This avoids wasting runtime for negligible results. The threshold depends on the runtime of the reduction. Fast reductions continue as long as they remove at least one edge, slow reductions have higher thresholds. *Dual ascent* uses a threshold of 1 %.

The order of the tests follows the following idea: Fast reductions are performed continuously before and after other reductions. These are tests *Non-Terminals of Degree 0, 1 and 2*, *Terminals of Degree 1* and *Minimum Terminal Edge*. This potentially simplifies the graph before more expensive reductions are run. The order of the other reductions is as follows:

1. Long Edges
2. Steiner Distance based reductions
3. NTD_k
4. Dual Ascent
5. Inclusions

The idea is to first remove edges, so more vertices are viable for NTD_k. Afterwards we run dual ascent on a hopefully smaller graph. Inclusion reductions often cause data changes that are incompatible with the other reductions. This requires the data to be recalculated. We therefore run these reductions last.

Some reductions are very slow or ineffective on dense graphs. We therefore change the behaviour of the reduction process for graphs where $\frac{m}{n} > 5$. The dual ascent reduction adapts its parameters accordingly, as discussed. The SDC and NTD_k reductions are skipped.

4.3.2 Cached data

Reductions require information about the graph. This information is often expensive to calculate and used by several reductions. For this end, we cache the following information:

- The distance between terminals and vertices.
- The *bottleneck Steiner distances* between terminals.
- A Voronoi partitioning of the graph.
- For each vertex we also store two lists of terminals. One is ordered by proximity based on d and the other based on $\overline{d}$.

Reductions may change the underlying data rendering the calculated information invalid. The changes cause a shift in one directions, either the real values become lower or higher, as discussed in Chapter 3.3.5. To avoid expensive recalculations, we use a flag for each of the stored types of information. Whenever a reduction changes the graph, it adapts this flags. These flags signify if the data is accurate, potentially lower than the real values or higher. Each reduction checks at the beginning, if it would still be valid given the settings of the flags. Whenever this check fails, the data in question is recalculated. E.g. the NTD_k reduction requires that s is either accurate or higher than in reality. After each pass all data marked as inaccurate is discarded. This avoids that reducible parts of the graph are not identified because of outdated data.

This completes the presentation of our implementation's details. We discuss some weaknesses in the last part.

4.4 Weaknesses

Our solver is able to solve 188 of the 200 PACE instances. We analyzed the remaining 12 instances to find the weaknesses of our solver. In general any instance that has many sub-solutions with similar weight are troublesome for the solver. If the heuristic is not able to clearly distinguish between them, the algorithm examines all of them.

The class of dense graphs has this property. The effect is exacerbated if the edges have the same or very similar weight. This is not only problematic for the solving algorithm. The reductions fail to identify reducible graph components in dense graphs. Furthermore, guiding heuristic and pruning become a ball-in-chain, as their computation time increases significantly, while they produce inferior results. They are nonetheless required to keep memory consumption in check.

The solver also performs poorly for complex instances with many terminals. This is unsurprising, as the goal was to develop a solver for instances with few terminals.

Another weakness of our solver is its memory consumption. Both runtime and memory consumption could probably be greatly improved if we reimplemented the solver in a more efficient programming language. During development, Python allowed us to quickly implement new ideas. We think that this contributed more to our solvers performance, than a more efficient programming language would have.

We explore how our solver compares to the other participants next.

CHAPTER 5

PACE Results

Our solver was written as a submission for PACE. In this chapter, we present the results of the competition. We also take a closer look at other submissions. Our goal is to find distinct features, that sets the different submissions apart. We also discuss if these features could enhance our solver.

PACE consisted of two rounds. After the announcement of the challenge, 100 ST instances were publicized. After the submission deadline, all submissions were tested against 100 private instances. These were not publicized, but the instances had properties similar to the public instances. The submissions where then ranked by how many private instances they were able to solve within the time and memory limit of 30 minutes and 6 GB. The results are shown in Table 5.1. Our solver placed fourth, with little distance to the top submissions and a considerable gap between fourth and fifth place.

The solvers could be tested using the website `http://www.optil.io`. The site provided a live ranking showing how the solvers performed for the public instances. The optil.io platform was also used to establish the final ranking using the private instances.

Due to the big gap between the top four submissions and the fifth place, we limit our discussion in the remainder of this chapter to the top 3 submissions. We first examine benchmarks for the top solvers and then take a closer look at the implementations of the solvers.

5.1 Benchmark

We take a closer look at how the different solvers perform. The official PACE results do not offer any information on runtime or memory consumption. In order to obtain this data, we benchmarked the top four solvers again on the public and private instances. The codebase for the other solvers from the time of submission. Because we added features

Place	Score	Participants
1	95	Team *wata_sigma*: Yoichi Iwata, Takuto Shigemura
2	94	Team *Jagiellonian*: Krzysztof Maziarz, Adam Polak
3	93	Team *reko*: Thorsten Koch, Daniel Rehfeldt
4	*92*	*Team tuw: Andre Schidler, Johannes Fichte, Markus Hecher*
5	67	Team *UWarsaw*: Michał Błaziak, Krzysztof Kiljan, Dominik Klemba, Marcin Mucha, Wojciech Nadara, Jakub Pawlewicz, Marcin Pilipczuk, Mateusz Radecki, Michał Ziobro
6	66	Team *noname*: Suhas Thejaswi
6	66	Team *DXHPeter*: Peter Mitura, Ondřej Suchý
6	66	Team *Johannes*: Johannes Varga
9	48	Team *sspspt*: Saket Saurabh, P. S. Srinivasan, Prafullkumar Tale

Table 5.1: Final PACE 2018 results [BS18]

since the submission, we tested both, the current version (*tuw*) and the submitted version (*tuw-pace*) of our solver.

We benchmarked the solvers using all 200 PACE instances. We also used the same time and memory constraints as PACE. The benchmark environment is described in Chapter 6. The aggregated results can be found in Table 5.2 and Figure 5.1. The detailed results are listed in Appendix A. Note that the solvers performed better in our benchmarking environment than they did on optil. The expected result for our solver would be 186 solved instances, 94 public and 92 private. The column *Common* shows the total runtime for those instances solved by all solvers.

The results show that *wata_sigma* is considerably more efficient than the other submissions. Although *reko* is tied for first place in this listing, it fails on two fairly simple instances. This suggests it could outperform the other submissions in terms of solved instances with limited extra work. Our solver is the least efficient in terms of resources. This was to be expected due to the use of Python. It is nonetheless not much slower than expected. In Figure 5.1 we see that especially for harder instances, it is on par with the other submissions. Furthermore, the changes to our solver paid off, as it solves more instances, unfortunately they also make it a little slower on almost all instances.

We will now examine the different submissions more closely.

5.2 First Place – *wata_sigma*

The solver *wata_sigma* has the same basic architecture as ours and is implemented in *Rust*. It first uses reductions to simplify the instance and then uses a solving algorithm to compute an SMT. Since there is no publication available yet, all conclusions in this chapter have been drawn from the source code [YS18].

Name	Private	Public	Runtime (s)	Common (s)	Memory (GB)
wata_sigma	95	94	156.73	94.16	1.51
reko	95	94	9078.32	5130.77	5.57
tuw	94	94	10985.62	7675.36	26.39
Jagiellonian	94	93	5645.55	3599.72	35.35
tuw-pace	93	94	10004.67	7116.27	25.39

Table 5.2: Comparison of solvers submitted for PACE

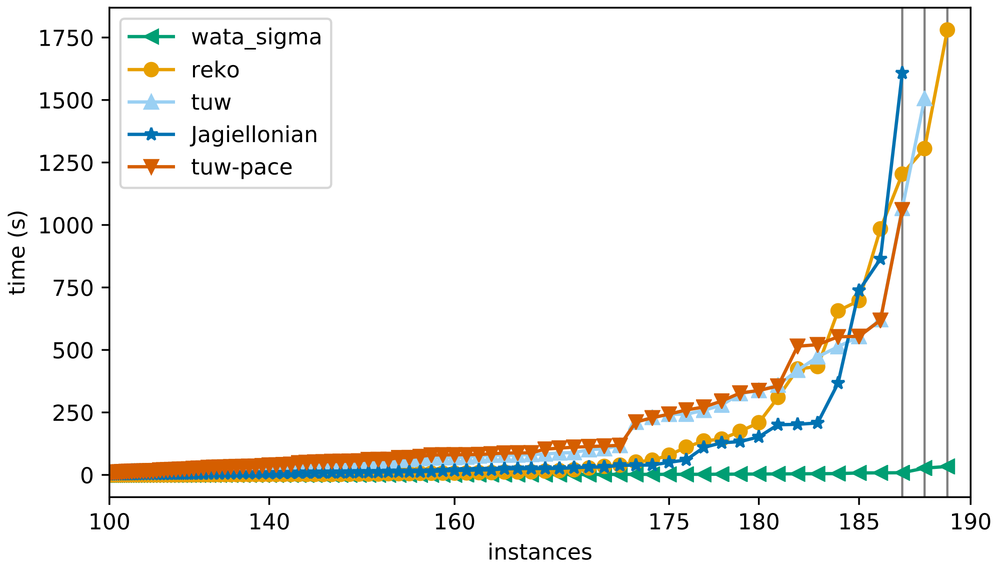


Figure 5.1: Comparison plot of solvers submitted for PACE. The markings show the runtime (y-axis) for an instance (x-axis). The instances are sorted by ascending runtime.

For the solving module a dynamic programming approach is used. Although the *Dreyfus-Wagner algorithm* is the classical $\mathcal{FPT}$ approach for ST, there exists another similar algorithm [EMV87]. It is similar in approach and runtime bounds. Originally it was designed to compute minimum-flows in networks and can be adapted to solve ST instances.

Many familiar reductions are used: Non-Terminals of Degree 0, 1 and 2 reductions and *Steiner Distance* reductions. Additionally it uses two reductions we will discuss subsequently: The *Nearest Special Vertices* test and the *Degree 3 Test*.

5.2.1 Minimum-Cost Network Flows and Steiner Trees

The *Minimum-cost Network Flow Problem (MCNFP)* tries to optimize the flow of a commodity in a network. An instance is defined by a directed graph (V, A), a demand

x_i for each vertex v_i and costs $c_{ij} : \mathbb{R} \mapsto \mathbb{R}$ for each arc a_{ij}. A solution to the problem is a *flow* $f = (f_{ij})$ that defines for each arc a_{ij} how much of the commodity flows from v_i to v_j. For f it holds that

$$\sum_{a_{ji} \in A} f_{ji} - \sum_{a_{i\ell} \in A} f_{i\ell} = x_i.$$

Therefore, the difference between the incoming and the outgoing commodity of a vertex is exactly the demand. The goal is to find the flow that minimizes

$$\sum_{a_{ij} \in A} c_{ij}(f_{ij}).$$

Note that the existence of a minimum flow implies that the sum over r is 0 [EMV87].

Given an ST instance we can transform it into a MCNFP instance. We first transform it into a Steiner arborescence instance with root r_0. We set $x_0 = |R| - 1$ and for all other terminals $z_i \in R'$ we set $x_i = 1$. Furthermore we set

$$c_{ij}(f_{ij}) = \begin{cases} c(a_{ij}) & \text{if } f_{ij} \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Any minimum flow now contains all terminals and minimizes the sum of edge costs.

The algorithm uses a *send-and-split* method to solve this problem. Given an instance (G, r, c) and $\emptyset \subset I \subseteq R$, we introduce some definitions.

$$x_I = \sum_{v_i \in I} x_i$$

$$A_I = \begin{cases} A & \text{if } x_I \geq 0, \\ \{(v_j, v_i) \mid a_{ij} \in A\} & \text{otherwise} \end{cases} \quad \text{and} \quad c^I_{ij}(f_I) = \begin{cases} c_{ij}(x_I) & \text{if } x_I \geq 0 \text{ or } a_{ij} \notin A_I, \\ c_{ji}(-x_I) & \text{otherwise.} \end{cases}$$

Furthermore, Let $l(v_i, I)$ be the solution for the following instance: We set $x_j = 0$ for all $v_j \notin I$ and then $x_i = x_i - x_I$.

Note that the solution to the original instance is $l(v_i, R)$ for any vertex v_i. The definition of A_I is used to either receive commodities from the neighbors in case of a deficit or send them commodities in case of a surplus. We now define

$$l'(v_i, I) = \min_{\emptyset \subset J \subset I} (l(v_i, J) + l(v_i, I \setminus J)).$$

This is called *splitting* and allows us to combine sub-solutions. The solution can now be calculated recursively by using the following formula (*sending*) [EMV87]

$$l(v_i, I) = \begin{cases} 0 & \text{if } |I| = 1, \\ \min\{\min_{a_{ij} \in A_I} (c^I_{ij}(x_I) + l'(v_j, I)), l'(v_i, I)\} & \text{else.} \end{cases}$$

The next step is to apply the algorithm to ST. As long as edges are used the costs are independent of the amount of commodities flowing. Further, for every arc a_{ij} exists a counter-arc a_{ji} with the same costs. Therefore $A = A_I$ and the cost of *sending* a commodity from v_i to v_j is $d(v_i, v_j)$. We can now simplify the algorithm and state $l(v_i, I) = \min_{v_j \in V}(d(v_i, v_j) + l'(v_i, I))$. This is exactly the *Dreyfus-Wagner algorithm* (see Chapter 3.2.2).

The distinguishing feature of this solver is its pruning strategy, which we will discuss next.

5.2.2 Pruning

In Chapter 3.2.2 we discussed that the best-case runtime for the Dreyfus-Wagner algorithm is exponential in the number of terminals. This bound also holds for this application of the MCNFP algorithm, as established above. The *wata_sigma* solver achieves its excellent results by applying very successful pruning strategies. We already discussed pruning in Chapter 3.2.5. We use the notation $T(v_i, I)$ for an SMT in $smt_G(I \cup \{v_i\})$.

Any tuple (v_i, I) represents a graph in $smt_G(I \cup \{v_i\})$. Whenever we can show that there exists no $T' \in smt_G(R \setminus I)$, such that $v_i \in V[T']$, we can discard (v_i, I). This is the concept behind pruning.

The solver uses three different pruning techniques. These are applied during the sending (or propagation) phase. Therefore, the algorithm has calculated the values for a subset $\emptyset \subset I \subseteq R$ and is about to send it to the other vertices. Pruning marks every vertex v_i either as *valid* if $T(v_i, I)$ may be a subgraph of any $T' \in smt_G(R)$, or *invalid* if there exists no SMT that contains $T(v_i, I)$ as a subgraph. When merging sub-solutions $T(v_i, I)$ and $T(v_i, J)$ all vertices that are valid in both sub-solution are merged and retain their flag.

The first strategy is the same as in our solver. When propagating the values, the solver uses Dijkstra's algorithm and propagates the valid flag as well. It stops propagation as soon as it finds the first terminal not in the current subset I. Any vertex that has not been flagged as valid before, remains invalid.

The second strategy applies the following idea: Imagine that we remove all vertices from the graph. We now add vertex by vertex to the graph. We always add the vertex v_i next that maximizes $l(v_i, I)$. After adding each vertex, we check if there exists a connected component containing all vertices in $R \setminus I$, if so we stop and remember the vertex v_j we added last. Since before we added the vertex, there could not have been a Steiner tree for $R \setminus I$, any SMT in $smt_G(R \setminus I)$ must contain a vertex v_ℓ, such that $l(v_\ell, I) \leq l(v_j, I)$. This gives us an upper bound and we can mark all vertices $v_i \in V$, such that $l(v_i, I) > l(v_j, I)$ as invalid. For the computation of this value we assume that $l(v_i, I) = \infty$ per default, as some vertices may not have a value due to the previous strategy.

The last strategy is the most intricate. The solver uses ideas expressed in Chapter 3.4.2. The first idea is the following: Given a tree T. If we add an edge between any two vertices we have a cycle. We now have to remove an edge from the cycle to restore the tree property. If any edge e in the cycle has higher weight than the added edge, we can lower the weight of the tree, by removing e.

This idea can be extended to paths instead of edges. We use the same definition for key-paths as in Chapter 3.4.2: Given a Steiner tree T, every terminal and any non-terminal of degree 3 or higher in T is a crucial vertex. A key-path is a path connecting two crucial vertices without any intermediary crucial vertices. Given a Steiner tree T, if we add a path S between two vertices in T, we create a key-path and a cycle. If any key-path S' in the cycle has higher weight than S, we can obtain a Steiner tree of lower weight by removing S'.

Using this idea, we can establish a bound on the distance between subtrees in an SMT. Given an SMT T for our current instance. Imagine a subtree in T_1, rooted at v_i. Furthermore, let $v_j \in V[T_1]$ and w be the maximum length of a key-path in the path between v_i and v_j. For all $v_\ell \in V[T] \setminus V[T_1]$ it must hold $d(v_j, v_\ell) \geq w$. Otherwise we could add the path from v_j to v_ℓ and remove the longest key-path to obtain a smaller tree, contradicting that T is an SMT.

Let V' be the set of all valid vertices in the current iteration for subset $\emptyset \subset I \subseteq R$. For every $v_i \in V'$ and every vertex $v_j \in V[T(v_i, I)]$ we store the maximum length of a key-path in the path between v_i and v_j in $T(v_i, I)$ as $f_{v_j}(v_i)$. Further, let $B = \bigcap_{v_i \in V'} V[T(v_i, I)]$ be the set of vertices in all trees. We now compute $b_{v_j} = \min_{v_i \in V'} f_{v_j}(v_i)$ for all $v_j \in B$. Since vertices in B are in all trees, these vertices are guaranteed to be in every $T \in smt_G(I)$. Since we do not know, which of the trees in $smt_G(I)$ will connect best with the remaining terminals, we take the minimum over all the key-path values. For all $v_i \in V, v_j \in B$ we can now mark v_i as invalid if $d(v_i, v_j) < b_{v_j}$.

This concludes the discussion of the solving module.

5.2.3 Nearest Special Vertices

The *Nearest Special Vertices (NSV)* test is a generalization of the *Nearest Vertex* and *Short Links* test (see Chapter 3.3.3) [HRW92b] and based on the following lemma:

Lemma 23. *[DV89, Theorem 1] Given an edge $e_{ij} \in E$. If there exist two terminals $z, z' \in R$, such that $d(z, z') = d(z, v_i) + c(e_{ij}) + d(v_j, z')$ and $\overline{s} \geq d(z, z')$ then e_{ij} is included in at least one SMT.*

The test can be performed efficiently using an MST. Let $T = M_G$: For each edge $e_{ij} \in E[T]$, we remove the edge from T and compute the two resulting connected components. If each component contains at least one terminal, the edge is on a shortest path between two terminals. To calculate $\overline{s}$ we find the shortest edge e_{st} that reconnects the two components. If no such edge exists, we can contract e_{ij}, otherwise $\overline{s} = c(e_{st})$. After we perform the test, we re-add e_{ij} to T and proceed with the next edge [DV89].

The test can be performed in $\mathcal{O}(n^2)$ [DV89], which exceeds our desired limit of $\mathcal{O}(n \log n)$. Furthermore, repeated application of *Nearest Vertex* and *Short Links* reduces the graph by a similar amount [Pol03]. The NSV test has therefore not been included in the tuw solver.

5.2.4 Non-Terminals of Degree 3

This is a test that we first encountered in this solver. It tests the incident edges of all non-terminals with degree 3. The idea is, that if we can reach all the neighbors of a vertex without using a specific edge, we do not need the edge.

Let $\underline{d}_X(v, v')$ be the distance of the shortest path between v and v' without using any vertex in I. If v' is not reachable from v without using v_i we assume that $\underline{d}_X(v, v') = \infty$. Given a $v_i \in V \setminus R$, such that $|\delta(v_i)| = 3$, we define $\{e_{ij}, e_{i\ell}, e_{io}\} = \delta(v_i)$ and $B = \{v_j, v_\ell, v_o\}$.

The tests are performed on a subset of permutations of the neighbors $\{(x_1, x_2, x_3) \mid x_1, x_2, x_3 \in B, x_2 \neq x_1, x_3 > x_2\}$. Therefore, three permutations, where each neighbor is exactly once at position one. Furthermore, only permutations such that $\underline{d}_{\{v_i\}}(x_1, x_2) \leq c(v_i, x_1) + c(v_i, x_2)$ and $\underline{d}_{\{v_i\}}(x_1, x_3) \leq c(v_i, x_1) + c(v_i, x_3)$ are considered. Therefore such permutations where x_1 can be reached from the other neighbors, without incurring a higher cost by avoiding v_i.

The edge $\{v_i, x_1\}$ can be eliminated if the weight of an MST in $(B, B \times B, \underline{d}_{\{v_i\}})$ is less than or equal to $\sum_{v' \in B} c(v_i, v')$. Therefore, an SMT containing all three neighbors, does not have to include v_i as well.

If the test above fails, we can extend the test to the neighbors of neighbors. The following test is performed recursively: Let $X = \{v_i\}, p = x_1, w_{v_i,p} = c(v_i, x_1)$:

1. If $\min\{\underline{d}_X(x_2, p), \underline{d}_X(x_3, p)\} \leq c(v_i, p)$, then remove $\{v_i, x_1\}$. We can reach p efficiently without using v_i.
2. If $p \in R$ stop. Since we have a terminal, we have to add p and the search for an alternative path failed.
3. Otherwise find all neighbors q of p, such that

$$\min\{\underline{d}_X(x_2, p), \underline{d}_X(x_3, p)\} > \max\{w_{v_i,p}, c(p, q)\}.$$

   If more than one such neighbor exists, we can stop, as we cannot determine if an alternative path exists. If none such neighbor exists, we found an alternative path and can eliminate $\{v_i, x_1\}$. If one such neighbor exists, we set $p = q, w_{v_i,p} = w_{v_i,p} + c(p, q), I = I \cup \{p\}$ and go to 1.

All vertices where the test succeeds are of degree 2 afterwards and therefore removed in a subsequent *Non-Terminals of Degree 2* test (see Chapter 3.3.2).

This concludes our discussion of this submission. The *Non-Terminals of Degree 3* reduction yielded good results in our solver and has been added permanently. The very efficient use of resources in this solver suggests that the pruning methods used could provide a great enhancement to our solver. Unfortunately due to the different algorithm the pruning strategies are not directly applicable.

5.3 Second Place – Jagiellonian

The *Jagiellionian* solver is implemented in *C++* and uses the *Dijkstra-Steiner algorithm* to solve instances. Besides the implementation language, the main difference is its architecture. The instance is first reduced and then the solver tries to split it into two smaller instances. If this is possible the solver calls itself recursively for the two sub-instances. Every sub-instance goes through the same reduction-split cycle. Once the instances cannot be partitioned further, the SMTs are computed and the solutions are merged together, until an SMT for the original instance is obtained. We will subsequently discuss splitting and the new reductions introduced in this solver.

As for the previous submission, no publication exists yet. The details discussed here have been discerned from the source code [MP18].

5.3.1 Heavy Edges

Heavy Edges is a graph transformation. It does not directly remove components, but allows for other reductions to find more reducible edges and vertices. The idea is that we can reduce the weight a terminal's incident edges, if we can show that it will be a leaf. Let $z_i \in R$ be a terminal, such that all incident edges have the same weight w_{z_i}. Furthermore, let $w_{max} = \max\limits_{e_{ij}, e_{i\ell} \in \delta(z_i)} d(v_j, v_\ell)$. If $w_{max} < w_{z_i}$ we can set $c(e_{ij}) = w_{max} + 1$ for all $e_{ij} \in \delta(z_i)$.

The reduction of edge weights may tighten bounds and allow other reductions to identify more reducible components. Note that the condition of equal weight among incident edges guarantees that an SMT using any of the edges before the reduction still is an SMT after the reduction. Furthermore, the weight condition in the neighborhood of z_i guarantees that the terminal is a leaf in any SMT. Given a Steiner tree containing two distinct edges $e_{ij}, e_{i\ell}$: Removing the first edge and adding the path from v_j to v_ℓ to the tree yields a cheaper Steiner tree.

5.3.2 Minimum Spanning Tree Contraction

Minimum Spanning Tree Contradction is a new inclusion reduction in addition to those discussed in Chapter 3.3.3. It uses the following rule: Given terminals $z_i, z_j \in R$ and a connecting edge $e_{ij} \in E$. If e_{ij} is contained in an MST for G it is in at least one SMT.

An MST is not always unique. Imagine a fully connected graph with three vertices and uniform edge costs. Now any two edges form an MST. We like to test all edges that are part of any MST, it is therefore not enough to simply compute one MST. *Kruskal's algorithm* [Kru56] computes an MST by using the following graph property: Given a cycle, the heaviest edge in the cycle is not in any MST. Algorithm 5.1 uses this property. It iterates over all edges ordered by weight. The algorithm uses the mapping C to keep track which vertices are in a connected component. Whenever it finds an edge connecting two yet unconnected components, it connects the two components in C and marks the edge as belonging to an SMT.

Algorithm 5.1: An algorithm returning all edges that are in any MST [MP18]

```
Data: A graph $G = (V, E, c)$
Result: A set of edges $E'$, containing all edges that are in any MST.
1  $E' \leftarrow \varnothing$
2  $Q \leftarrow E$
3  $C(v_i) = v_i$ for all $v_i \in V$
4  while $Q \neq \varnothing$ do
5      $w = \min_{e \in Q} c(e)$
6      $F = \{e \in Q \mid c(e) = w\}$
7      foreach $e_{ij} \in F$, $C(v_i) \neq C(v_j)$ do
8          $E' \leftarrow E' \cup \{e_{ij}\}$
9      end
10     foreach $e_{ij} \in F$ do
11         $C(v_\ell) \leftarrow C(v_i)$ for all $v_\ell \in V$ such that $C(v_\ell) = C(v_j)$
12     end
13     $Q = Q \setminus F$
14 end
15 return $E'$
```

5.3.3 Splitting

The *Jagiellonian* solver uses the concept of *articulation points* to transform the instance into smaller instances. An articulation point is a vertex $v_i \in V$ such that removing v_i results in a disconnected graph with two or more connected components. Should a component not contain any terminals, all its vertices can be removed from G, as no SMT will contain them. Otherwise, it comprises a sub-instance.

For this section, given two graphs $G_1 = (V_1, E_1, c)$ and $G_2 = (V_2, E_2, c)$, we refer to the operation $G = (V_1 \cup V_2, E_1 \cup E_2, c)$ as *merging* the graphs. Further, let $C_1, ..., C_\ell$ be the connected components after the removal of articulation point v_i, we refer to this as v_i induces the components $C_1, ..., C_\ell$. Given a set of vertices B and a graph G, we add B to G', a subgraph of G, by setting $V[G'] = V[G'] \cup B$ and $E[G'] = E[G'] \cup ((V[G'] \times B) \cap \bigcup_{v_i \in B} \delta_G(v_i))$

In order to obtain smaller instances we use an articulation point v_i that induces ℓ components. We can create ℓ sub-instances by adding $\{v_i\}$ to every component. We merge the resulting SMTs and thereby obtain a solution for the original instance. Due to the exponential runtime of the algorithm, decreasing k can reduce the overall runtime considerably.

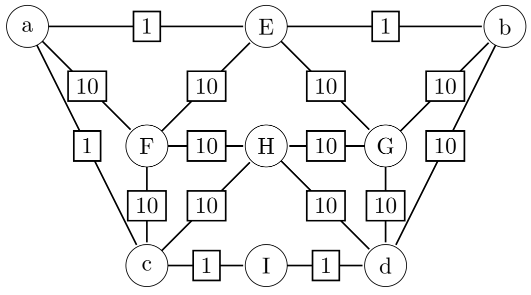


Figure 5.2: A graph with an articulation set $\{E, H, I\}$.

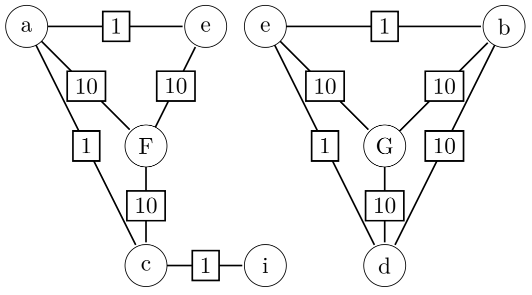


Figure 5.3: A graph split into two instances by the articulation set.

Example 5.3.1. In our running example in Figure 2.1, vertices b, E and F are articulation points. Using F yields the components $C_1 = \{a, b\}, C_2 = \{E, J\}$ and $C_3 = \{c, d, G, I, H\}$. C_2 does not contain any terminals and can be removed. Solving for C_1 yields a Steiner tree with a, b and F. Solving for C_3 computes a Steiner tree using d, I, F and c. Both Steiner trees can be merged and form an SMT for the original instance. ■

We can extend the concept of articulation points to *articulation sets*. Removing all vertices in an articulation set from the graph disconnects the graph into two or more components. Given an algorithm to compute articulation points, we can compute articulation sets of size l the following way: For all $B \subseteq V$ such that $|B| = l - 1$, we temporarily remove B from G and run the algorithm. Any articulation point we find together with B forms an articulation set. We restore G before running the algorithm for the next subset.

We assume that any articulation set we use is minimal. Therefore no subset of it is an articulation set. This is automatically the case, if we search for sets with increasing cardinality. For the remainder of this section we assume that an articulation set splits the graph into two connected components C_1 and C_2. In case more components are obtained, we can merge them until only two remain. Adding the vertices in the articulation set reconnects them.

While the splitting operation is easy for *articulation points* the process is more intricate for sets. Several problems occur. We cannot simply add all vertices in the set as terminals and then merge the solutions. This would create cycles and therefore no SMT. It is therefore necessary to establish which vertices from the set the sub-instances will use. We cannot predict this beforehand, as the only requirement is, that the sub-solutions share at least one vertex.

Another problem is, that the graph induced by one of the sub-instances on the SMT might not be connected. Therefore, the sub-solution required is a disconnected Steiner tree. This is not solvable with an SMT solver.

Example 5.3.2. In Figure 5.2 we have an example graph. The set $B = \{E, H, I\}$ is an articulation set and the edges with weight 1 form an SMT. Let $C_1 = \{a, c, F\}, C_2 = \{b, d, G\}$.

Clearly an SMT for the original instance does not contain H. We can therefore not simply add all vertices in B as terminals. Furthermore, the graph induced by $V[C_2] \cup B$ on the SMT is not connected. ■

This implies that we have to solve two problems:

1. When splitting the instance using an articulation set B, we cannot predict which sub-instance will use which vertices in B.
2. We need a different way to solve the sub-instances. As we showed, for some sub-instances we require a disconnected Steiner tree.

We address the two problems in this order.

We first define our problem in more detail. Given an articulation set B and an SMT T. Let $B' = B \cap V[T]$ and let T_1 be the subgraph induced by $V[C_1] \cup B'$ on T. Furthermore, let T_2 be defined analogously for C_2. Next, we discuss the possible configurations of T_1 and T_2. The case $|B'| = 1$ is similar to using an articulation point. T_1 and T_2 are both SMTs and they can be merged. If $|B'| = 2$ it becomes more complex. Either T_1 or T_2 must be disconnected, otherwise merging them would result in a cycle. Since we have two articulation vertices used, one of the trees must have exactly two connected components. The number of possibilities quickly increases with the size of B'. In general, for any $v_i, v_j \in B', v_i \neq v_j$, the two vertices are in the same connected component in T_1 iff they are in different connected components in T_2.

We established that for any articulation set with more than one element, we cannot find an SMT by solving two sub-instances. For every non-empty subset of B we have to solve for all possible configurations of T_1 and T_2 and take the minimum over all resulting SMTs. This number increases quickly. For $|B| = 1$ there is just one subset with one configuration. In case $|B| = 2$ we have three subsets, with in total $2 + 2 \cdot 1 = 4$ configurations. For $|B| = 3$ this increases to $8 + 3 \cdot 2 + 3 \cdot 1 = 17$ and for $|B| = 4$ we obtain infeasible $1 \cdot 48 + 4 \cdot 8 + 6 \cdot 2 + 4 \cdot 1 = 96$ different configurations. One configuration implies the solving of two sub-instances.

Example 5.3.3. Table 5.3 shows all possible configurations for articulation set $B = \{E, H, I\}$. The sets symbolize components. Therefore $\{E\}$ symbolizes one component containing E and $\{E, I\}, \{H\}$ stands for two connected components, one containing E and I, the other one H. For every B_1, B_2 tuple we solve the two sub-instances and merge the resulting graphs to obtain an SMT. The cheapest solution over all possible configurations is the solution to the original instance. ■

We established how we deal with problem number one, it remains to discuss how to solve problem 2.

At the current state the solver won't produce the required sub-solutions, as it will always produce a connected Steiner tree. We can use a transformation on the sub-instance, that produces the required result. Assume that we want to calculate the configuration

$\lvert B'\rvert = 1$		$\lvert B'\rvert = 2$		$\lvert B'\rvert = 3$	
B_1	B_2	B_1	B_2	B_1	B_2
				$\{E,H,I\}$	$\{E\},\{H\},\{I\}$
$\{E\}$	$\{E\}$	$\{E,H\}$	$\{E\},\{H\}$	$\{E,H\},\{I\}$	$\{E,I\},\{H\}$
		$\{E\},\{H\}$	$\{E,H\}$	$\{E,H\},\{I\}$	$\{H,I\},\{E\}$
$\{H\}$	$\{H\}$	$\{H,I\}$	$\{H\},\{I\}$	$\{H,I\},\{E\}$	$\{E,H\},\{I\}$
		$\{H\},\{I\}$	$\{H,I\}$	$\{H,I\},\{I\}$	$\{E,I\},\{H\}$
$\{I\}$	$\{I\}$	$\{E,I\}$	$\{E\},\{I\}$	$\{E,I\},\{H\}$	$\{E,H\},\{I\}$
		$\{E\},\{I\}$	$\{E,I\}$	$\{E,I\},\{H\}$	$\{H,I\},\{E\}$
				$\{E\},\{H\},\{I\}$	$\{E,H,I\}$

Table 5.3: Illustration of all possible subset combinations for an articulation set $B = \{E,H,I\}$.

$\{v_1, v_2\}, \{v_3\}$ for our current sub-instance. We now merge v_1 and v_3 by adding the edges $\{e_{1j} \mid e_{3j} \in \delta(v_3)\}$ and removing v_3. The vertices can now reach v_1 the same way and with the same cost as they could v_3. The solver can now compute a connected SMT. We transform this SMT into the desired sub-solution by unmerging v_1: We reintroduce v_3 and map all edges back.

Example 5.3.4. In Figure 5.3 we see the two sub-instances for $B' = \{E,I\}$, with $B_1 = \{E,I\}, B_2 = \{\{E\},\{I\}\}$. This is the combination that will result in the SMT for the instance. H has been removed, as it is not in B'. E and I have been marked as terminals. On the right side, the edges to I have been mapped to E. The resulting SMT will contain the edge $\{d,e\}$ which will be transformed to $\{d,I\}$. This will together with the SMT for the left side yield an SMT for the original instance. ∎

As we saw, splitting quickly becomes infeasible with growing size of the articulation set. The solver therefore ensures that the set has a certain quality. Articulation points are always used. In case the articulation set has size 2, each of the two components has to contain at least five terminals and the instance must not have more than 3000 edges. For articulation sets of size 3 the limits are eight and 1000. No articulation sets of higher cardinality are used.

5.3.4 Implementing the features

We tried to enhance our solver with the insights gained by the antecedent analysis. The *MST contraction* has been added. For *heavy edges* and *splitting* we benchmarked our implementation. The results are in Table 5.4. *tuw* is our solver at the current version, with current features and no splitting or heavy edges enabled. *tuw-pace* is added for comparison and is our solver as submitted for PACE. The other entries show our solver with enabled splitting and/or heavy edges. For the PACE instances splitting has no impact and heavy edges worsens the performance. For this reason we disabled splitting per default.

Name	Solved	Runtime (s)	Common Runtime (s)	Memory (GB)
tuw	188	10985.62	10513.38	26.39
tuw-split	188	11499.82	11020.31	26.43
tuw-heavy	187	9253.89	9253.89	25.79
tuw-split-heavy	187	9993.20	9993.20	25.50
tuw-pace	187	10004.67	10004.67	25.39

Table 5.4: Comparison between different tuw configurations using features from the Jagiellonian solver.

5.4 Third Place – reko

reko is the only submission we examined, that was not created specifically for this challenge. The solver has already participated in the DIMACS challenge where it placed third in the single threaded exact ST category [Gam14]. The solver is named *SCIP-Jack* and is an extension for SCIP, a collection of software tools for mathematical problems [GEG+17, Ach09]. The application is written in C [KR18].

For this solver there are several papers describing the different aspects of the application. These heavily influenced the implementation of our solver. The reductions are the same as in our solver, except for those found in the other submissions. We will therefore focus our discussion on the solving algorithm. In contrast to the other two submissions, this one uses a different approach to calculate SMTs.

5.4.1 ILP Reformulation

SCIP-Jack extends the linear programming capabilities of SCIP. This allows SCIP-Jack to use the existing linear programming implementation and only extend it to ST. For this end it uses a similar model to that discussed in Chapter 2.4:

$$\begin{aligned}
\text{minimize: } & \sum_{a \in A} (c(a) \cdot x_a) \\
\text{subject to: } & \sum_{\substack{a_{ij} \in A \\ v_i \notin W \; v_j \in W}} x_{a_{ij}} \geq 1, && \text{for all } W \subset V, r \notin W, W \cap R \neq \emptyset, \\
& x_a \in \{0,1\} && \text{for all } a \in A, \\
& \sum_{a_{ij} \in A} x_{a_{ij}} \begin{cases} = 0 & \text{if } v_j = r \\ = 1 & \text{if } v_j \in R \setminus \{r\} \\ \leq 1 & \text{else} \end{cases} && \text{for all } v_j \in V,
\end{aligned}$$

$$\begin{aligned}
\sum_{a_{ij} \in A} x_{a_{ij}} &\leq \sum_{a_{j\ell} \in A} x_{a_{j\ell}} && \text{for all } v_j \in V \setminus R, \\
\sum_{a_{ij} \in A} x_{a_{ij}} &\geq x_{a_{j\ell}} && \text{for all } v_j \in V \setminus R, a_{j\ell} \in A.
\end{aligned}$$

The additional constraints have been added to tighten the bounds computed for the LP relaxation [Reh15, KM96]. These constraints are based on the idea of commodity flow, already discussed in Chapter 5.2.1. The basic idea is that the commodities flow from the root to the terminals. Every terminal needs to receive the commodity. Every non-terminal can only pass it on, if it receives the commodity.

5.4.2 Branch & Bound

The number of possible solutions to the ILP is limited. As we have one variable per edge, we have 2^m possible variable assignments. We can also represent these possible assignments in a binary tree. Starting from the root, the left edge represents assigning 0 to the first variable, the other edge assigning 1. We apply the same idea to the root's children, this time for the second variable. If we continue this idea for all children, we obtain a tree, with 2^m leafs. Each leaf represents one possible assignment for the variables and each non-leaf a tentative assignment. We call this the *enumeration tree* of our ILP.

For this section we use the following definitions: $x = \{x_1, ..., x_m\}$ is the set of variables in our program. $\hat{x} = \{x_1 = 0, x_2 = 1\}$ is a tentative assignment. We denote by ILP our linear program and by LP its LP relaxation. Given a tentative assignment $\hat{x}$, we obtain the linear program $LP^{\hat{x}}$ by adding equality constraints for all fixed variable values. Furthermore we denote by x^{101} the assignment $\{x_1 = 1, x_2 = 0, x_3 = 1\}$ and given a label $\ell = 101$ we denote by $x^{\ell 1}$ the assignment $x^{\ell} \cup \{x_{|\ell|+1} = 1\}$. We will also use LP^{ℓ} as a shorthand for $LP^{x^{\ell}}$

Branch and Bound (B&B) is a method to systematically explore all possible assignments [LD60, MS08b]. It does so by recursively searching through the enumeration tree. We maintain a stack $L = \{\}$ of linear programs and a best solution *ub*. We start by adding ILP^0 and ILP^1 to L. B&B now takes an element LP^{ℓ} from L. If x^{ℓ} is a tentative assignmentment we add $ILP^{\ell 0}$ and $ILP^{\ell 1}$ to L. Otherwise, let T be the graph represented by x^{ℓ}. If T is a Steiner tree and $|T| < ub$ we set $ub = T$. We continue this until L is empty. Then, *ub* is the SMT for this instance. The idea of adding all possible assignment for a variable to L is called *branching.*

It is easy to see that this will systematically search the enumeration tree. Unfortunately, it is infeasible even for small ST instances due to its exponential runtime. We can drastically reduce the runtime if we check for every tentative assignment x^{ℓ} if it complies with all constraints. If any constraint is violated, we do not add $ILP^{\ell 0}$ and $ILP^{\ell 1}$ to L. We can thereby skip whole subtrees of the enumeration tree.

Additionally B&B uses a technique already encountered multiple times in this thesis: *Bounds*. Whenever the algorithm is at a non-leaf, it has a tentative variable assignment $\hat{x}$. It now checks if it can extend this assignment to an optimal solution. For this it uses a lower bound function $lb(\hat{x})$, returning a lower bound on all assignments $\hat{x}$ can be extended to. If $lb(\hat{x}) > |ub|$ we stop and do not add any sub-programs to L, as the subtree cannot contain an optimal solution.

5.4.3 Branch & Cut

The ILP above has another problem. The number of constraints is exponential, as there exist 2^n subsets. It is therefore not possible to efficiently use the ILP definition above. We require a technique to find out which constraints we need, in order to obtain an SMT.

Imagine an LP with only two variables. The possible variable assignments can be represented by points on a 2-D plane. Without any constraints, every point is a valid solution. Every constraint adds a line to the plane and all valid assignments are now on one side of the line. It therefore *cuts* off possible assignments. After all constraints have been added we end up with a (convex) polygon containing all solutions. In the general case we have more than two variables. Generally constraints introduce hyperplanes and we obtain a (convex) polytope [Dan98].

Branch and Cut (B&C) extends the branching concept with a *cutting phase*. We start with a minimal set of constraints as ILP. In our case we omit the constraints for the vertex subsets. We then try to add cuts as necessary. Therefore, the linear program will be adapted continuously during the solving process. The cutting phase is run on any ILP^ℓ including the initial program, before the algorithm branches.

The cutting phase consists of the following steps:

1. The algorithm solves the LP relaxation of our current linear program ILP^ℓ. Let $\overline{w}$ be the total cost of the assignment.
2. As $\overline{w}$ is a lower bound, we check if $\overline{w} > |ub|$, if so, the current branch cannot contain an optimal solution and we discard ILP^ℓ.
3. If the assignment for LP^ℓ has only integer values, it is a feasible solution for ILP^ℓ. Let T be the graph represented by this solution. If T is a Steiner tree, we discard ILP^ℓ as we have found an optimal solution for this branch. If $|T| < ub$ we update $ub = T$.
4. Otherwise, we search for violated cuts and add them to ILP^ℓ. If we find any, we start again with the first step.
5. If no cuts have been found we branch.

It remains to explain, how the violated cuts are found. In terms of network flows, we are searching for cuts that have no in-flow of commodities. We therefore try to find a cut separating a terminal $z_i \in R \setminus \{r\}$ and the root r. A cut separates the vertices into two sets C_1 and C_2, where $r \in C_1$. We define $A' = A \cap (C_1 \times C_2)$, the arcs connecting the two sets. Whenever we find a cut such that $\sum_{a_{ij} \in A'} x_{a_{ij}} < 1$ we have a violated constraint.

SCIP-Jack uses different methods to find such cuts. One strategy is to search for cuts minimizing this sum [HO92]. Another strategy is to search for cuts that minimize $|A'|$ as this will result in constraints with fewer variables [KM96].

We have skipped some practical details here for brevity. One important factor to implement this efficiently is managing the cuts. If all found cuts are added and never removed, the resulting linear program may become too complex to solve [KM96]. Another consideration is, if cuts are valid for the whole program, or just for the current branch. This depends if the cut is dependent on any variable we branched on.

As the solving algorithm is very different to our approach, we were not able to gain any transferable insights from analyzing this solver.

CHAPTER 6

Benchmarks

In this last chapter we want to discuss how our solver performs outside the PACE context. As the primary goal was to perform well at the competition, non-PACE instances were always a secondary consideration. Although we used other instances for tests, we optimized the solver towards the PACE instances. It was therefore interesting for us to see, how well the solver performs on instances that were unknown to us.

In order to test our solver thoroughly, we used several instance collections publicized online. SteinLib is a large collection of ST instances used in many papers [KMV00]. The results from these instances allow us to compare our solver to others outside the PACE context. We also used the instance collections provided for the 11th DIMACS challenge [Wer16, LLL+14].

For all our benchmarks we used servers with two *Intel Xeon E5-2650 v4* CPUs. Each process ran on exactly one CPU, which boosts the clock speed to 2.9 GHz. The servers werer running *Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-124-generic x86_64)*, *Python 2.7.12* and *GCC 5.4.0 20160609*. Every instance was tested in three separate runs. The result is the average of these runs. For these benchmarks we extended our timelimit to two hours and allowed each instance to use up to 24 GB of memory. We also tested the PACE instances again with these extended limits.

Table 6.1 shows the results aggregated per instance set. The entry *PACE* shows the results for the PACE instances[1].

Name	S	F	Runtime	MO	TO	Original
1R	27	0	2804.63	0	0	24
2R	10	17	2960.90	11	6	11

[1]The original implementation was limited to instances with $|R| \leq 64$. This accounts for some but not all the differences in the results.

Name	S	F	Runtime	MO	TO	Original
ALUE	13	2	2638.91	0	2	7
ALUT	7	2	2163.41	0	2	3
B	18	0	11.77	0	0	18
C	18	2	89.90	0	2	8
copenhagen14	14	7	3174.08	0	7	10
csd	5	9	23.16	0	9	6
DIW	21	0	2034.40	0	0	21
DMXA	14	0	206.15	0	0	14
D	18	2	245.86	0	2	8
E	17	3	1936.70	0	3	8
ES10FST	15	0	7.17	0	0	15
ES20FST	15	0	13.03	0	0	15
ES30FST	15	0	29.20	0	0	15
ES40FST	15	0	46.66	0	0	15
ES50FST	15	0	74.53	0	0	15
ES60FST	15	0	105.71	0	0	15
ES70FST	15	0	123.32	0	0	-
ES80FST	15	0	169.70	0	0	-
ES90FST	15	0	246.79	0	0	-
ES100FST	15	0	256.90	0	0	-
ES250FST	14	1	7724.59	0	1	-
ES500FST	0	15	0.00	0	15	-
ES1000FST	0	15	0.00	0	15	-
ES10000FST	0	1	0.00	1	0	-
GAP	13	0	291.53	0	0	12
goemans	240	0	19750.98	0	0	162
I080	92	8	6999.28	0	8	85
I160	78	22	1264.91	20	2	51
I320	59	41	4237.36	13	28	50
I640	47	53	7065.82	15	38	25
LIN	32	5	13126.44	0	5	34
MC	3	3	27.77	1	2	2
MSM	30	0	1180.10	0	0	29
P4E	11	0	20.35	0	0	10
P4Z	10	0	9.05	0	0	10
P6E	15	0	19.12	0	0	14
P6Z	15	0	13.92	0	0	14
PACE	188	12	11002.39	8	4	-
PUC	6	44	128.36	5	39	6
PUCN	3	10	23.09	1	9	3
skutella	1	4	1.07	1	3	1
smc	15	0	6.61	0	0	15

Name	S	F	Runtime	MO	TO	Original
TAQ	13	1	1498.27	0	1	11
SP	5	3	2.72	0	2	5
TSPFST	54	22	37238.36	4	18	3
vienna-geo-original	0	23	0.00	4	19	-
vienna-i-simple[2]	5	80	3662.70	9	68	0
X	3	0	65.49	0	0	2

Table 6.1: The benchmark results for Steinlib instances. Column *S* shows how many instances were solved successfully and *F* how many failed. *MO* lists how many instances failed due to the memory limit and *TO* how many due to the time limit. The column *Original* refers to the number of instances solved by the original *Dijkstra-Steiner* implementation [HSV16].

The results show that the tighter limits imposed by PACE are not the reason that our solver fails to solve more instances. They also confirm the weaknesses discussed in Chapter 4.4. For the PACE instances, the memory limit is more of a problem than the time limit.

In general our solver was able to solve more instances than the original implementation. On the one hand this is expected, as the addition of reductions alone should improve the solver. On the other hand the original implementation was in C++, which gives it an edge in terms of performance.

Although our solver performed comparatively well, it performed worse on some sets. In case of the set *2R* our solver ran out of memory on an instance solved by the original implementation. This may well be due to the original implementation being benchmarked with a 100 GB memory limit.

The set *LIN* revealed another weakness of our solver. The two instances our solver failed to solve were sparse graphs with over 70 000 edges. The reductions take a lot of time on graphs of this size. For these instances the reductions alone took about one hour. We never dealt with graphs of this size during development. Adding some adaptions for such graphs would probably enhance the runtime.

It would also be interesting to compare our solver against one of the other submissions based on these instances. Fortunately *SCIP-Jack* has been benchmarked using SteinLib instances [Reh15]. As the results are not available for full instance sets, the number of solved instances is not a good metric. We can nonetheless compare the solvers based on the publicized results. In general SCIP-Jack performed better than our solver. On the instance sets *D, E* and *PUC* it solved a few instances more and for the sets *I320, I640* and *vienna-i-simple* the differences are considerable. Our solver performed considerably better for the instances in the *LIN* set.

[2]Three instances caused an error regarding the maximum recursion depth in Python.

SteinLib results are also available for the *mozartballs* solver [FLL+17]. Results are only available for some hard instances from the *skutella*, *SP* and *PUC* sets. This solver was able to solve many instances our solver was unable to solve.

The results show that our solver performs well outside the PACE context. They also show that there is room for improvement and suggest starting points.

CHAPTER 7

Conclusion

In this thesis we thoroughly discussed how to solve instances of the Steiner tree problem. We focused on finding SMTs and used approximations to help us in this process. We approached the discussion from two sides. The theoretical side provided all the state-of-the-art knowledge that we used. We thereby built a solid base of proven theorems and algorithms on which we constructed our solver. We then approached the discussion from the practical side. Here we laid out the implementation details of our solver. We focused on the aspects not covered by theory alone.

We also discussed the performance of our solver in the competition, where it performed well. The publication of the source codes allowed us to analyze the other submissions. We presented the results of this analysis. We focused on the details that distinguish the solvers from each other. We also tried to apply the knowledge gained from this analysis to our solver. This allowed further improvements to our implementation. We also presented attempts that weren't fruitful.

In the last part we discussed the results we obtained by benchmarking our solver. The instances we used include many that were used in other papers. This allowed us to compare our results to those of other solvers outside PACE. The comparison showed us that our solver performs well and also suggests starting points of where to start searching in the considerably large room for improvement.

Conclusively we want to consider, that with the exception of *mozartballs* [FLL+17], which was not submitted, the submissions comprise the state-of-the-art. Especially *SCIP-Jack* is an established solver. Nonetheless, our solver was able to play in the same ballpark.

7.1 Future Work

There is a long list of ideas for minor improvements of our current implementation. While none will cause leaps in capability, each may cause small improvements. These are specific to our implementation. It would also be interesting, how our solver would perform if it was implemented in other programming languages.

Two of the three submissions we discussed used yet unknown reductions. This shows that there may still be reductions that we do not know about and that could make a difference. Finding new ones might therefore be a worthwhile endeavour. Especially since new reductions could be used in all solvers.

Tight bounds can be used for both solving and reducing ST instances. For upper bounds RSPH is the most common approximation. We have discussed how its results can be improved by combining different ideas. New local search methods for Steiner trees may improve these upper bounds, as well as new ideas on which graphs to run the approximation. We discussed the way little details in the calculation can effect the results of the approximations. As our dual ascent implementation is quite complex, there is probably room for improvement. Both guiding heuristics used in our solver are expensive to compute. Finding a heuristic that provides a similar quality of bounds but is easier to calculate could also greatly improve our solver.

As the top submission showed. A good pruning strategy can greatly improve runtime and memory consumption. Since the basic solving algorithm is similar to the one we use, it should be possible to transfer the ideas.

Dense graphs were a problem for all the solvers: All the instances where every solver failed were dense graphs. Focusing on finding a method to solve them would therefore improve the capabilities of our solver immensely. There are no reductions that perform well on dense graphs. This could be a starting point as well. These instances are not impossible to solve, as tests with *SCIP-Jack* showed, that given a longer time limit, it was able solve them.

APPENDIX A

PACE Benchmarks

Table A.1 shows the results for each PACE instance per solver. For each solved instance, the time to solve and the peak memory usage are stated. Instances that have not been solved are either marked by *MO* (out of memory), *TO* (time out) or *ERR* (runtime error).

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
001	0.01	0.00	0.02	0.00	0.41	4.00	0.01	0.00
002	1.54	21.00	0.13	4.00	1.54	93.00	0.08	1.00
003	2.97	37.00	0.26	4.00	7.15	123.00	0.15	4.00
004	9.00	48.00	0.17	4.00	5.50	125.00	0.48	9.00
005	11.19	66.00	0.28	4.00	10.65	181.00	3.93	26.00
006	0.01	0.00	0.02	1.00	0.42	4.00	0.01	0.00
007	0.02	0.00	0.03	0.00	0.81	4.00	0.01	0.00
008	0.09	4.00	0.04	2.00	1.94	87.00	0.01	1.00
009	0.01	0.00	0.01	0.00	0.44	4.00	0.01	0.00
010	0.12	4.00	310.11	96.00	2.09	89.00	0.02	0.00
011	0.12	4.00	135.89	62.00	2.31	91.00	0.02	0.00
012	0.04	2.00	0.03	0.00	1.33	87.00	0.01	0.00
013	6.00	19.00	0.06	2.00	2.11	94.00	0.04	1.00
014	5.49	19.00	0.07	2.00	1.88	92.00	0.04	1.00
015	5.46	19.00	0.07	2.00	1.53	92.00	0.04	1.00
016	5.81	19.00	0.07	4.00	2.66	95.00	0.05	1.00
017	6.35	19.00	0.06	2.00	2.38	95.00	0.06	1.00
018	0.26	4.00	0.07	2.00	2.57	97.00	0.18	4.00
019	0.21	4.00	0.08	1.00	5.48	99.00	0.17	4.00
020	0.18	4.00	0.07	4.00	5.99	99.00	0.18	4.00

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
021	0.18	4.00	0.08	2.00	2.25	96.00	0.14	4.00
022	0.22	10.00	0.07	1.00	3.99	99.00	0.16	4.00
023	3.06	107.00	2.06	81.00	78.60	561.00	1.29	62.00
024	3.08	107.00	2.06	81.00	79.35	562.00	1.08	62.00
025	4.12	107.00	2.06	81.00	79.11	561.00	1.31	62.00
026	3.11	107.00	2.06	81.00	81.44	561.00	1.30	62.00
027	0.01	0.00	0.02	1.00	0.52	4.00	0.01	0.00
028	0.06	1.00	0.03	1.00	0.89	4.00	0.00	0.00
029	0.02	0.00	0.03	0.00	0.97	31.00	0.01	0.00
030	0.17	4.00	0.03	1.00	1.22	86.00	0.01	0.00
031	0.06	2.00	0.02	0.00	1.61	88.00	0.02	0.00
032	0.10	4.00	0.03	0.00	1.33	88.00	0.01	1.00
033	0.04	2.00	0.04	1.00	2.36	88.00	0.01	0.00
034	0.13	4.00	0.05	1.00	2.16	92.00	0.01	0.00
035	0.15	4.00	0.12	4.00	3.35	93.00	0.02	0.00
036	2.07	19.00	0.24	4.00	11.18	101.00	0.03	1.00
037	0.18	4.00	0.05	2.00	2.94	95.00	0.03	0.00
038	5.65	19.00	0.07	2.00	8.12	105.00	0.09	4.00
039	0.52	4.00	0.09	2.00	5.41	98.00	0.03	4.00
040	0.26	4.00	0.06	2.00	3.53	99.00	0.04	0.00
041	0.51	4.00	0.07	2.00	8.94	100.00	0.03	0.00
042	0.24	4.00	0.07	1.00	4.27	99.00	0.04	1.00
043	0.30	4.00	0.15	4.00	7.49	100.00	0.03	0.00
044	0.31	4.00	0.15	4.00	10.34	101.00	0.03	1.00
045	1.60	19.00	0.13	4.00	15.24	107.00	0.03	0.00
046	1.75	21.00	0.14	4.00	2.15	106.00	0.09	2.00
047	3.29	37.00	0.27	4.00	8.29	155.00	0.45	4.00
048	1.87	44.00	1.26	41.00	33.51	124.00	1.34	18.00
049	2.23	38.00	0.51	4.00	39.73	121.00	0.91	4.00
050	5.82	122.00	27.57	43.00	100.74	163.00	0.68	4.00
051	10.79	49.00	0.19	4.00	9.61	131.00	0.62	4.00
052	8.88	65.00	0.27	4.00	15.19	163.00	3.90	26.00
053	0.24	4.00	0.05	0.00	6.93	92.00	0.02	0.00
054	0.18	4.00	0.05	0.00	3.97	90.00	0.02	2.00
055	0.06	1.00	0.02	0.00	1.02	57.00	0.01	0.00
056	0.17	4.00	0.03	1.00	1.46	88.00	0.02	0.00
057	0.04	2.00	0.03	1.00	1.49	88.00	0.01	0.00
058	0.03	2.00	0.03	0.00	1.32	87.00	0.01	0.00
059	0.03	2.00	0.03	1.00	2.00	88.00	0.01	0.00
060	0.28	4.00	0.04	1.00	2.71	92.00	0.02	0.00
061	0.04	2.00	0.04	0.00	1.44	88.00	0.01	0.00

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
062	0.58	4.00	0.08	2.00	4.93	95.00	0.02	0.00
063	3.33	19.00	0.09	2.00	5.03	96.00	0.02	0.00
064	0.18	4.00	0.06	1.00	4.94	98.00	0.03	1.00
065	0.20	4.00	0.08	4.00	5.18	99.00	0.03	1.00
066	6.62	20.00	0.13	4.00	7.81	101.00	0.05	1.00
067	4.60	72.00	0.49	4.00	51.50	122.00	0.28	4.00
068	0.10	4.00	0.03	1.00	2.92	89.00	0.01	0.00
069	0.13	4.00	17.05	65.00	5.93	119.00	0.24	4.00
070	0.18	4.00	7.13	52.00	10.06	133.00	0.21	4.00
071	0.03	1.00	0.03	0.00	1.13	87.00	0.01	0.00
072	0.26	4.00	0.04	2.00	3.01	93.00	0.02	0.00
073	0.10	2.00	0.04	1.00	3.71	91.00	0.01	0.00
074	0.08	4.00	0.05	1.00	3.54	92.00	0.02	1.00
075	9.43	19.00	0.10	4.00	8.23	106.00	0.10	2.00
076	0.80	4.00	0.24	4.00	16.63	112.00	0.05	2.00
077	1.37	22.00	0.23	4.00	27.00	119.00	0.19	4.00
078	2.49	23.00	1.02	40.00	21.48	123.00	0.18	4.00
079	2.84	23.00	4.91	42.00	53.64	123.00	0.23	4.00
080	4.22	87.00	12.45	42.00	48.53	129.00	0.42	9.00
081	0.07	4.00	0.03	0.00	2.20	88.00	0.02	0.00
082	0.07	4.00	0.04	0.00	1.23	89.00	0.02	1.00
083	0.40	4.00	0.07	1.00	5.12	96.00	0.01	0.00
084	4.53	19.00	2.71	38.00	36.24	104.00	0.13	4.00
085	0.59	4.00	2.03	58.00	17.20	191.00	0.15	4.00
086	0.92	4.00	TO	TO	47.23	265.00	1.22	56.00
087	0.90	4.00	TO	TO	59.07	265.00	1.12	49.00
088	1.52	19.00	0.10	4.00	5.89	102.00	0.03	0.00
089	0.21	4.00	0.05	1.00	5.78	100.00	0.04	2.00
090	12.48	20.00	0.09	4.00	7.76	105.00	0.07	4.00
091	0.71	4.00	0.16	4.00	9.16	109.00	0.05	1.00
092	0.28	4.00	0.05	0.00	8.86	95.00	0.01	0.00
093	0.03	0.00	0.03	1.00	1.33	91.00	0.01	0.00
094	0.31	4.00	0.03	1.00	4.70	95.00	0.03	1.00
095	0.11	4.00	0.04	1.00	2.09	94.00	0.02	0.00
096	0.12	4.00	0.04	1.00	1.75	94.00	0.02	1.00
097	1.68	19.00	0.14	4.00	9.32	107.00	0.04	2.00
098	0.46	4.00	0.04	0.00	3.44	91.00	0.02	0.00
099	0.49	4.00	0.12	4.00	16.00	100.00	0.05	0.00
100	1.51	33.00	0.07	15.00	11.21	103.00	0.05	0.00
101	2.68	19.00	0.17	4.00	21.49	110.00	0.11	4.00
102	0.06	2.00	0.04	1.00	1.83	92.00	0.02	0.00

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
103	0.06	1.00	0.04	2.00	1.49	93.00	0.02	1.00
104	1.32	21.00	0.06	2.00	4.17	98.00	0.04	1.00
105	8.88	31.00	0.13	4.00	15.66	110.00	0.06	0.00
106	0.03	0.00	0.02	1.00	0.82	4.00	0.01	0.00
107	12.75	21.00	0.23	4.00	12.50	112.00	0.04	0.00
108	4.55	19.00	0.16	4.00	6.61	107.00	0.07	2.00
109	25.97	54.00	0.19	4.00	10.66	114.00	0.08	4.00
110	2.34	19.00	0.09	4.00	5.75	111.00	0.07	2.00
111	17.90	20.00	0.21	4.00	14.07	113.00	0.10	4.00
112	2.91	24.00	4.80	41.00	32.27	126.00	0.23	4.00
113	7.59	85.00	143.44	45.00	242.37	187.00	0.61	4.00
114	20.52	484.00	984.46	154.00	242.82	300.00	1.12	21.00
115	0.01	0.00	0.02	0.00	0.75	4.00	0.01	0.00
116	0.72	4.00	0.06	2.00	16.47	98.00	0.02	0.00
117	0.02	1.00	0.02	0.00	1.79	88.00	0.01	0.00
118	0.99	4.00	1.54	47.00	17.37	102.00	0.03	1.00
119	0.12	4.00	0.02	1.00	1.30	92.00	0.01	1.00
120	0.47	4.00	0.04	1.00	1.92	93.00	0.02	0.00
121	0.11	4.00	0.04	0.00	2.68	95.00	0.02	0.00
122	0.10	2.00	0.04	1.00	2.14	97.00	0.01	0.00
123	8.77	34.00	0.24	4.00	23.69	120.00	0.11	4.00
124	1.93	199.00	3.26	40.00	50.17	127.00	0.42	4.00
125	0.75	4.00	0.24	4.00	17.56	107.00	0.04	2.00
126	0.35	4.00	0.09	4.00	7.31	104.00	0.05	1.00
127	1.46	19.00	0.81	4.00	11.97	123.00	0.09	4.00
128	3.36	20.00	5.06	40.00	30.13	116.00	0.13	4.00
129	4.28	710.00	11.69	63.00	87.78	131.00	0.71	4.00
130	0.09	4.00	0.04	2.00	1.94	90.00	0.02	0.00
131	0.60	4.00	0.16	4.00	32.14	107.00	0.04	0.00
132	50.42	875.00	0.13	4.00	11.53	110.00	0.15	4.00
133	0.59	4.00	0.12	4.00	4.55	99.00	0.03	0.00
134	2.31	335.00	0.21	4.00	31.80	127.00	0.28	9.00
135	3.38	237.00	3.50	41.00	65.32	138.00	0.31	4.00
136	0.70	4.00	0.09	2.00	19.81	113.00	0.04	2.00
137	19.86	1942.00	0.79	4.00	68.82	126.00	0.46	4.00
138	3.40	64.00	0.07	2.00	4.29	109.00	0.07	0.00
139	1.28	19.00	0.37	4.00	10.26	109.00	0.02	1.00
140	5.92	102.00	1.35	42.00	69.39	146.00	0.28	4.00
141	2.37	440.00	2.16	46.00	22.65	111.00	0.17	4.00
142	4.16	1919.00	1.76	46.00	45.69	119.00	0.24	4.00
143	10.78	100.00	0.08	2.00	7.37	111.00	0.08	4.00

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
144	4.55	331.00	2.18	41.00	39.50	133.00	0.35	4.00
145	1.53	18.00	0.04	2.00	4.43	99.00	0.01	0.00
146	2.77	146.00	0.57	4.00	42.08	113.00	0.32	4.00
147	5.27	392.00	9.27	41.00	61.74	128.00	0.19	4.00
148	12.76	19.00	0.12	4.00	33.55	119.00	0.04	0.00
149	38.54	117.00	1.54	38.00	38.31	123.00	0.23	4.00
150	14.02	221.00	34.16	46.00	258.24	223.00	2.02	21.00
151	26.16	367.00	1203.68	198.00	230.03	324.00	2.32	25.00
152	133.16	1423.00	TO	TO	553.40	443.00	5.26	34.00
153	2.97	500.00	0.39	4.00	25.06	121.00	0.06	2.00
154	10.06	37.00	1.55	38.00	30.81	118.00	0.11	2.00
155	0.02	1.00	0.02	0.00	0.96	4.00	0.01	0.00
156	2.01	80.00	111.31	85.00	89.35	151.00	0.34	9.00
157	4.15	1781.00	1.67	39.00	33.31	124.00	0.11	4.00
158	2.38	679.00	0.46	4.00	30.95	119.00	0.23	4.00
159	6.06	53.00	6.79	41.00	52.76	126.00	0.21	4.00
160	7.52	302.00	7.10	50.00	104.79	159.00	4.33	20.00
161	MO	MO	433.58	380.00	MO	MO	MO	MO
162	MO	MO	208.92	266.00	MO	MO	MO	MO
163	MO	MO	51.42	154.00	MO	MO	MO	MO
164	MO	MO	174.33	220.00	MO	MO	MO	MO
165	MO	MO	697.43	573.00	MO	MO	MO	MO
166	9.41	68.00	0.89	4.00	32.69	119.00	0.13	4.00
167	200.72	466.00	20.60	83.00	418.53	261.00	2.02	47.00
168	1.23	37.00	0.09	2.00	7.45	108.00	0.08	4.00
169	207.89	398.00	1.34	47.00	44.59	118.00	1.15	36.00
170	128.40	1295.00	1.76	38.00	75.58	127.00	0.24	4.00
171	MO	MO	423.92	78.00	MO	MO	MO	MO
172	MO	MO	TO	TO	MO	MO	MO	MO
173	ERR	ERR	TO	TO	MO	MO	MO	MO
174	31.64	142.00	7.98	114.00	117.64	150.00	0.89	4.00
175	2.06	244.00	0.31	4.00	15.71	110.00	0.12	4.00
176	5.66	94.00	9.35	72.00	83.62	167.00	0.48	9.00
177	202.27	1120.00	0.17	4.00	15.80	118.00	0.07	2.00
178	367.39	556.00	3.15	69.00	71.34	129.00	1.55	39.00
179	29.13	36.00	0.17	4.00	13.01	133.00	0.16	4.00
180	18.51	78.00	1.27	38.00	55.14	129.00	0.17	4.00
181	25.08	538.00	TO	TO	337.61	369.00	4.39	27.00
182	38.50	44.00	0.48	4.00	53.93	138.00	0.10	4.00
183	12.86	243.00	0.15	4.00	8.86	116.00	0.08	0.00
184	24.64	606.00	6.27	49.00	279.20	265.00	7.83	23.00

	Jagiellonian		reko		tuw		wata_sigma	
I	t	m	t	m	t	m	t	m
185	TO	TO	0.26	4.00	21.87	126.00	0.21	4.00
186	16.23	42.00	0.14	4.00	9.15	117.00	0.14	4.00
187	1608.02	2973.00	TO	TO	621.25	329.00	2.43	43.00
188	31.41	240.00	11.78	87.00	67.68	171.00	0.26	4.00
189	25.98	695.00	TO	TO	359.33	494.00	3.17	28.00
190	737.87	1384.00	656.46	131.00	1506.00	765.00	2.97	43.00
191	2.46	118.00	0.90	4.00	23.98	133.00	0.24	4.00
192	60.11	1683.00	TO	TO	513.12	361.00	33.72	30.00
193	39.43	272.00	79.06	90.00	472.24	766.00	0.51	4.00
194	TO	TO	1780.28	120.00	TO	TO	2.16	49.00
195	MO	MO	59.18	211.00	TO	TO	MO	MO
196	MO	MO	39.18	93.00	TO	TO	MO	MO
197	863.96	3572.00	15.06	47.00	210.60	451.00	27.18	132.00
198	109.96	1604.00	1305.58	215.00	1066.09	969.00	8.72	59.00
199	152.24	1502.00	TO	TO	325.14	456.00	8.39	55.00
200	ERR	ERR	TO	TO	TO	TO	MO	MO

Table A.1: Solver comparison for each PACE instance. Columns marked t show the run time in seconds, columns marked m show the peak memory usage in megabytes

List of Figures

List of Tables

List of Algorithms

Acronyms

NTD_k Non-Terminals of Degree k. 37, 38, 41, 44, 58, 59
$\mathcal{FPT}$ Fixed Parameter Tractable. 6, 18, 22, 63

B&B Branch and Bound. 4, 74
B&C Branch and Cut. 4, 75

ILP Integer Linear Program. 9, 15, 73–75

LP Linear Program. 9, 74, 75

MCNFP Minimum-cost Network Flow Problem. 63–65
MST Minimum Spanning Tree. 7, 12, 14, 15, 34, 37, 42, 47, 66–69, 72

PACE The Parameterized Algorithms and Computational Experiments Challenge. 1–3, 49, 53, 59, 61, 62, 72, 77, 79–81
PTm Paths with many Terminals. 36, 37

RSPH Repeated Shortest Path Heuristic. 12, 13, 16, 44, 45, 49–54, 82

SDC Steiner Distance Circuit. 36, 44, 58
SMT Steiner minimal tree. 8, 9, 11, 15, 18–23, 25, 27–33, 35–40, 44, 45, 49, 62, 65–75, 81
SPH Shortest Path Heuristic. 12, 14, 16
ST Steiner Tree Problem on Graphs. 1, 5, 7, 9, 12, 13, 15, 18, 22, 23, 25, 26, 28, 33, 61, 63, 64, 73, 74, 77, 82

TSP Traveling Salesman Problem. 12, 14, 15

Bibliography

[Ach09] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[Ane80] Y. P. Aneja. An Integer Linear Programming Approach to the Steiner Problem in Graphs. *Networks*, 10(2):167–178, 1980.

[Bac94] P. Bachmann. *Analytische Zahlentheorie.* Teubner, 1894. https://archive.org/stream/dieanalytischeza00bachuoft#page/402/mode/2up.

[BHKK07] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier Meets Möbius: Fast Subset Convolution. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 67–74. ACM, 2007.

[BS18] E. Bonnet and F. Sikora. 3rd Parameterized Algorithms & Computational Experiments Challenge. https://www.win.tue.nl/~bjansen/talks/PACE2018.pdf, August 2018.

[CFK+15] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, Marcin Pilipczuk, Michal Pilipczuk, and S. Saurabh. *Parameterized Algorithms.* Springer International Publishing, 2015. OCLC: 920804734.

[Dan98] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1998.

[Dan03] S. V. Daneshmand. *Algorithmic Approaches to the Steiner Problem in Networks.* PhD thesis, Universität Mannheim, Mannheim, 2003.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dow91] K. Dowsland. Hill-climbing, simulated annealing and the Steiner problem in graphs. *Engineering Optimization*, 17:91–107, 1991.

[Dui93] C. W. Duin. *Steiner's Problem in Graphs.* PhD thesis, University of Amsterdam, 1993.

[DV89] C. W. Duin and A. Volgenant. Reduction Tests for the Steiner Problem in Graphs. *Networks*, 19(5):549–567, 1989.

[DV97] C. W. Duin and S. Voß. Efficient path and vertex exchange in steiner tree algorithms. *Networks*, 29:89–105, 1997.

[DW72] S. E. Dreyfus and R. A. Wagner. The Steiner Problem in Graphs. *Networks*, 1:195–207, 1972.

[dW02] M. P. de Aragão and R. F. Werneck. On the Implementation of MST-Based Heuristics for the Steiner Problem in Graphs. In D. M. Mount and C. Stein, editors, *Algorithm Engineering and Experiments*, pages 1–15. Springer-Verlag Berlin Heidelberg, 2002.

[EEK17] S. Edelkamp, A. Elmasry, and J. Katajainen. Optimizing Binary Heaps. *Theory of Computing Systems*, 61(2):606–636, 2017.

[EMV87] R. E. Erickson, C. L. Monma, and A. F. Veinott. Send-and-Split Method for Minimum-Concave-Cost Network Flows. *Mathematics of Operations Research*, 12(4):634–664, 1987.

[FGK08] F. V. Fomin, F. Grandoni, and D. Kratsch. Faster Steiner Tree Computation in Polynomial-Space. In D. Halperin and K. Mehlhorn, editors, *Algorithms - ESA 2008*, volume 5193, pages 430–441. Springer, Berlin, Heidelberg, 2008.

[FLL$^+$17] M. Fischetti, M. Leitner, I. Ljubić, M. Luipersbeck, M. Monaci, M. Resch, D. Salvagnin, and M. Sinnl. Thinning out Steiner trees: A node-based model for uniform edge costs. *Mathematical Programming Computation*, 9(2):203–229, 2017.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*, 34(3):596–615, 1987.

[Gam14] G. Gamrath. DIMACS 11: Competition Results. http://dimacs11.zib.de/contest/challenge-results.pdf, 2014.

[GEG$^+$17] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, and B. Mu. The SCIP Optimization Suite 5.0. *ZIB Report*, page 43, 2017.

[HK70] M. Held and R. M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18(6):1138–1162, 1970.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100 – 107, 1968.

[HO92] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut in a Graph. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 165–174. Society for Industrial and Applied Mathematic, 1992.

[HRW92a] F. K. Hwang, D. S. Richards, and P. Winter. Chapter 1: Introduction. In *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*, pages 93–102. Elsevier, 1992.

[HRW92b] F. K. Hwang, D. S. Richards, and P. Winter. Chapter 2: Reductions. In *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*, pages 103 – 124. Elsevier, 1992.

[HRW92c] F. K. Hwang, D. S. Richards, and P. Winter. Chapter 4: Heuristics. In *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*, pages 151–176. Elsevier, 1992.

[HSV16] S. Hougardy, J. Silvanus, and J. Vygen. Dijkstra meets Steiner: A fast exact goal-oriented Steiner tree algorithm. *Mathematical Programming Computation*, 9(2):135–202, 2016.

[Joh75] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1975.

[Kar72] R. M. Karp. Reducibility among Combinatorial Problems. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972.

[KM96] T. Koch and A. Martin. Solving Steiner Tree Problems in Graphs to Optimality, 1996.

[KMV00] T. Koch, A. Martin, and S. Voß. SteinLib: An Updated Library on Steiner Tree Problems in Graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2000.

[KR18] T. Koch and D. Rehfeldt. SCIP-Jack. https://github.com/dRehfeldt/scipjack/, May 2018.

[Kru56] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[Lan09] E. Landau. *Handbuch Der Lehre von Der Verteilung Der Primzahlen*. Teubner, 1909.

[LD60] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497, 1960.

[LLL+14] M. Leitner, I. Ljubić, M. Luipersbeck, M. Prossegger, and M. Resch. New Real-World Instances for the Steiner Tree Problem in Graphs, 2014.

[Mar77] A. Martelli. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence*, 8:1–13, 1977.

[Meh88] K. Mehlhorn. A Faster Approximation Algorithm for the Steiner Tree Problem in Graphs. *Information Processing Letters*, 27(3):125–128, 1988.

[Min90] M. Minoux. Efficient Greedy Heuristics For Steiner Tree Problems Using Reolptimization And Super Modularity. *INFOR: Information Systems and Operational Research*, 28(3):221–233, 1990.

[MP18] K. Maziarz and A. Polak. Exact Steiner Tree with few terminals. https://bitbucket.org/krismaz/pace2018, May 2018.

[MS08a] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures.* Springer Science & Business Media, 2008.

[MS08b] K. Mehlhorn and P. Sanders. Chapter 12: Generic Approaches to Optimization. In *Algorithms and Data Structures*, pages 233–262. Springer-Verlag Berlin Heidelberg, 2008.

[Pap95] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.

[Pol03] T. Polzin. *Algorithms for the Steiner Problem in Networks.* PhD thesis, Universität des Saarlandes, Saarbrücken, 2003.

[Pri57] R. C. Prim. Shortest Connection Networks And Some Generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[PUW18] T. Pajor, E. Uchoa, and R. F. Werneck. A robust and scalable algorithm for the Steiner problem in graphs. *Mathematical Programming Computation*, 10(1):69–118, 2018.

[RC86] V. J. Rayward-Smith and A. Clare. On finding Steiner Vertices. *Networks*, 16:283–294, 1986.

[Reh15] D. Rehfeldt. *A Generic Approach to Solving the Steiner Tree Problem and Variants.* Master thesis, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 2015.

[RUW00] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A Hybrid Grasp with Pertubations and Adaptive Path-Relinking for the Steiner Problem in Graphs, 2000.

[TM80] H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. *Mathematica Japonica*, 24(6):573–577, 1980.

[UW12] E. Uchoa and R. F. Werneck. Fast Local Search for the Steiner Problem in Graphs. *ACM Journal of Experimental Algorithmics*, 17(2):2.2:1–2.2:222, 2012.

[Voß92] S. Voß. Steiner's Problem in Graphs: Heuristic Methods. *Discrete Appl. Math.*, 40(1):45–72, 1992.

[VSA96] M. G .A. Verhoeven, M. E. M. Severens, and E. H. L. Aarts. Local search for Steiner tree problems in graphs. In V. J. Rayward-Smith, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, pages 117–129. Wiley, United States, 1996.

[Wer16] R. F. Werneck. 11th DIMACS Implementation Challenge: Downloads. http://dimacs11.zib.de/downloads.html, 2016.

[WMS92] P. Winter and J. MacGregor Smith. Path-distance heuristics for the Steiner problem in undirected networks. *Algorithmica*, 7(1):309–327, 1992.

[Won84] R. T. Wong. A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph. *Mathematical Programming*, 28(3):271–287, 1984.

[YS18] I. Yoichi and T. Shigemura. Steiner tree solver. https://github.com/wata-orz/steiner_tree, May 2018.

[ZSH+09] Z. Zhang, N. R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner. A* Search with Inconsistent Heuristics. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 634–639, 2009.