Algorithms for the Steiner Problem in Networks

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Tobias Polzin

Saarbrücken
Mai, 2003

Datum des Kolloquiums: 16. Mai 2003

Dekan: Professor Dr. Philipp Slusallek

Gutachter:
Professor Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken
Professor Dr. William J. Cook, Georgia Institute of Technology, Georgia, USA

Abstract

The Steiner problem in networks is the problem of connecting a set of required vertices in a weighted graph at minimum cost. It is a classical $\mathcal{NP}$-hard problem with many important applications. For this problem we develop, implement and test several new techniques. On the side of lower bounds, we present a hierarchy of linear relaxations and class of new relaxations that are the currently strongest polynomially solvable linear relaxations. On the side of preprocessing techniques, we improve some known reduction tests and introduce powerful new ones. For upper bounds we introduce the successful concept of heuristic reductions. Finally, we integrate these blocks into an exact algorithm. For the exact algorithm and for the different components we present very good computational results on the large benchmark library SteinLib.

Kurzzusammenfassung

Das Steiner Problem in Netzwerken ist das Problem, eine Menge von Basisknoten in einem gewichteten Graphen kostenminimal zu verbinden. Es ist ein klassisches $\mathcal{NP}$-schweres Problem mit vielen Anwendungen. F¨ur dieses Problem entwickeln, implementieren und bewerten wir einige neue Techniken. Bez¨uglich unterer Schranken stellen wir eine Hierarchie von Linearen Relaxationen auf und entwickeln eine Klasse von Relaxationen, die die zur Zeit st¨arksten polynomiell l¨osbaren Linearen Relaxationen darstellen. Im Bereich des Preprocessing verbessern wir einige bekannte Reduktionstests und entwickeln einige starke neue Tests. F¨ur obere Schranken f¨uhren wir das erfolgreiche Konzept der “heuristischen Reduktionen” ein. Schließlich werden diese Bausteine zu einem exakten Algorithmus zusammengesetzt. Sowohl der exakte Algorithmus, als auch die einzelnen Komponenten erzielen sehr gute experimentelle Resultate auf der umfassenden Benchmarkbibliothek SteinLib.

Acknowledgments

I want to express my heartfelt thanks to my parents Heidi and Thomas Polzin, Ricarda Ott, and all my friends for their love, understanding, constant support and encouragement.

Also I would like to thank my colleague Siavash Vahdati Daneshmand for the inimitable collaboration and Prof. Kurt Mehlhorn, for his support, encouragement, and the opportunity to work in the stimulating atmosphere at the MPI group of Algorithms and Complexity. And thanks to the people at the MPI who create this atmosphere.

Hannah Bast, Susan Hert, and Bobbye Pernice have proofread parts of my thesis. Thank you.

I gratefully acknowledge the financial support provided by the Max Planck Institute for Computer Science.

Contents

Chapter 1

Introduction

1.1 About This Work

The Steiner problem in networks is the problem of connecting a set of required vertices in a weighted graph at minimum cost. This is a classical $\mathcal{NP}$-hard problem with many important applications in network design in general and VLSI layout in particular. The primary goal of our research has been the development of empirically successful algorithms. This means we designed and implemented algorithms that

1. generate Steiner trees of low cost in reasonable running times (upper bounds),
2. prove the quality of a Steiner tree by providing a lower bound on the optimal value (lower bounds),
3. or find an optimal Steiner tree (exact algorithms).
4. As an important prerequisite for the first three tasks, we used preprocessing techniques to reduce the size of the original problem without changing the optimal solution (reduction tests).

The value of our algorithms is measured by comparing our results to those of other research groups on the huge and well-established benchmark library for Steiner tree problems [SteinLib, KMV01]. In the case of exact algorithms, this measure is the best one can get because of the $\mathcal{NP}$-hardness result. For upper and lower bounds, the extensive experimental evaluation gives much sharper and much more relevant information than a worst-case analysis.

Why is one interested in solving Steiner tree problems? First of all, there are a lot of practical problems that can be modeled as Steiner tree problems. The question remains why an optimal solution is relevant. One could argue that in practical applications the computation of a heuristic Steiner tree and a lower bound that proves a gap of less than one percent is sufficient, because typically there are bigger inaccuracies in the model anyway. Still, the development of exact algorithms has a solid practical justification. The best algorithms for upper and lower bounds rely heavily on the computation of very sharp bounds for subproblems of the original problem, and the best results are achieved when an optimal solution of these subproblems can be found. In addition, the empirical success of our exact algorithms (even large benchmark instances with 30000 vertices can be solved in a matter of minutes) puts into question the justification of heuristic upper bound algorithms.

Our work is also motivated from the theoretical side: Classical $\mathcal{NP}$-hard combinatorial optimization problems like the "Traveling Salesman Problem" (TSP), scheduling problems, and the Steiner tree problem, have attracted researchers as test environments for their methods for decades. As a consequence, all important techniques in the fields of Operations Research and Combinatorial Optimization have been tried on these problems, leading to a respected competition for the best implementation. The relevance of this is twofold: On the one hand, the current state of affairs (which problems can be solved and what are the best known upper and lower bounds) reflects to some extent the state of the art in solving combinatorial optimization problems. On the other hand, these classical problems have always been "creation engines" for new methods and techniques that are also applicable to other problems. The outcome has been such general techniques as the cutting plane approach, which was invented in the TSP context. Likewise in this work, we present appealing approaches (e.g., partitioning as reduction technique, reduction-based heuristics) that may be applicable to other problems.

Similar to many other elaborate optimization packages, our package for the Steiner tree problem consists of a large collection of different components that interact extensively. In fact, our best programs for generating upper bounds, lower bounds, and exact solution all use essentially the same code, and just arrange the use of the components in different ways. Therefore, it is not possible to give

a concise description of "how to produce a good upper bound" in some dozen lines of pseudo-code. Hence, we have to give a bottom-up description: We will first describe the different building blocks separately and give pointers to the necessary connections of the blocks elsewhere. Still, we cannot provide a close grained picture of our program. This becomes obvious given the fact that merely printing the code without any further explanation requires roughly 1000 pages. Therefore, we describe the algorithms on a rather abstract level, and give only pointers to the description of standard techniques used.

This work summarizes the current state of our research, which has already been presented at conferences (ESA, APPROX, Combinatorial Optimization) and has been published in journals (Discrete Applied Mathematics, Operations Research Letters). It has already received considerable recognition in research on the Steiner tree problem, visible in citations in recently published literature [RUW02, PW02, UdAR99, Uch01, KMV01, CT01] and university lectures [JKP$^+$02].

In the following, we will give a list of our main contributions:

Lower Bounds:

- There are many (mixed) integer programming formulations of the Steiner problem. The corresponding linear programming relaxations are of great interest particularly, but not exclusively, for computing lower bounds, but not much was known about the relative quality of these relaxations. We compare the linear relaxations of all classical, frequently cited integer programming formulations of this problem from a theoretical point of view with respect to their optimal values. We present several new results, establishing very clear relations between relaxations, which have often been treated as unrelated or incomparable, forming a **hierarchy of relaxations**.
- We introduce a **collection of new relaxations** that are stronger than any known relaxation that can be solved in polynomial time, and place the new relaxations into our hierarchy. Further, we show how such a relaxation can be used in practical algorithms. Except for the flow-balance constraints introduced by [KM98], this is the first successful attempt to use a relaxation that is stronger than Wong's directed cut relaxation from 1984 [Won84].

Preprocessing/Reduction techniques:

- For some of the classical reduction tests, which would have been too time-consuming for large instances in their original form, we design **efficient realizations**, improving the worst-case running time to $O(m + n \log n)$ in many cases. Furthermore, we design new tests, filling some of the gaps left by the classical tests.
- Previous reduction tests were either alternative based or bound based. That means to simplify the problem they either argued with the existence of alternative solutions, or they used some constrained lower bound and upper bound. We develop a framework for **extended reduction tests**, which extends the scope of inspection of reduction tests to larger patterns and combines for the first time alternative-based and bound-based approaches effectively.
- We introduce the new concept of **partitioning-based reduction techniques**, which has a significant impact on the reduction results in some cases.
- We integrate all tests into a **reduction packet**, which performs stronger reductions than any other package we are aware of. Additionally, the reduction results of other packages can be achieved typically in a fraction of the running time. ("The result reported in Polzin

and Daneshmand is by far the best result in terms of both number of instances solved to optimality and computational times factoring into the difference in cpu speed."[CT01].)

Upper Bounds:

- We present variants of known path heuristics, including an empirically fast variant with a fast worst-case running time of $O(m + n \log n)$. The previous running time for this kind of path heuristic was $O(rm \log n)$.
- We introduce a new meta-heuristic, **reduction-based heuristics**. On the basis of this concept, we develop heuristics that achieve typically sharper upper bounds than the strongest known heuristics for this problem despite running times that are smaller by orders of magnitude.

Exact Algorithm:

We integrate the previously mentioned building blocks into an exact algorithm that achieves very good running times.

- For most benchmark instances the program computes the exact solution in running times that are shorter than the running times of other authors by orders of magnitude. ("Computational times reported in [PV01c] are far better than the rest." [CT01])
- There are 73 instances in SteinLib that have not been solved by any other research group. We have been able to solve 32 of them.
- For geometric Steiner problems, our algorithm for general networks is (together with a preprocessing phase [WWZ01] that exploits some of the geometric properties) the fastest algorithm and beats the specially tailored MSTH approach [WWZ00], which has received much attention.

Additionally, we present a procedure that uses the fixed-parameter tractability of the Steiner problem for subgraphs of small width.

In Chapter 2, we study some relaxations of the problem and methods for computing lower bounds using them; they are also frequently used in the following chapters. In Chapter 3, reduction techniques are discussed, which play a central role in our approach. These techniques are also the basis of the reduction-based heuristics, which we introduce in Chapter 4 on upper bounds. In Chapter 5, the building blocks from the previous chapters are integrated into an exact algorithm, which is shown to be empirically successful.

Some comparison of our empirical results to those of other authors is presented in Section 3.8 for reduction techniques, in Section 4.6 for upper bounds, and in Section 5.4 for the exact algorithm. Detailed results for the exact algorithm are given in the appendix (see page 105).

Most of the background information relevant to this work can be found in Hwang, Richards and Winter [HRW92]; we have tried to keep the notation compatible with that book. The basic definitions are repeated in Section 1.3.

The implementation and most of the results presented were produced jointly with Siavash Vahdati Daneshmand [PV00, PV01a, PV01c, PV01e, PV02a, PV02b, PV03]. I declare that my contribution to these results constituted at least half of the work.

1.2 About Experimental Results in this Work

In each of the following sections, we will report on the experimental behaviour of certain algorithms. We do not claim that algorithms can be evaluated beyond doubt by running them on a set of test instances. But when considering (exact) algorithms for an $\mathcal{NP}$-hard problem, there is no fully satisfactory alternative. Proving guaranteed performance ratios for certain components (like heuristics for computing upper bounds) cannot be a complete substitute, because such results are often too pessimistic due to their worst case character or lack of better proof techniques. From a comparative point of view, a much sharper differentiation is necessary; particularly in the context of exact algorithms, where even marginal differences (small fractions of a percent) in the value of the bounds can have a major impact on the behaviour of the algorithm.

In addition, we consider the comparability of results a critical issue, which strongly suggests using benchmark instances. There are two major benchmarks for the Steiner problem in networks: the collection in the OR-Library [Bea90] and SteinLib [SteinLib]. The instances of the OR-Library are much older, with the advantage that more comparative results exist on them. On the other hand, only one type of instance is represented (sparse and random). The library SteinLib is much more extensive, containing instances of all common types. But giving experimental results for all these instances in each section would make the work unreasonably long, so we have chosen a compromise option: For the intermediary results (for example concerning upper bounds or reductions), we give average results on each group of the problem instances from SteinLib (if the gap to the optimal solution is measured, we restrict ourselves to those groups where all optimal values are known). For the final results of the complete algorithm, however, we additionally give results for all instances in SteinLib. We leave aside some very small and easy instance groups.

In Table 1.1, we give a brief description of the instance classes of SteinLib. For more comprehensive information, see [SteinLib]. The column "Instances" gives the number of instances in the group. The column "Status" shows whether all instances of this column have been solved by other authors and by us ("solved"), or only by us ("*solved*" in italics), or if there are some "unsolved" instances.

Also it must be mentioned that for actual tests, we did not always implement the data structures and algorithms with the best known (worst-case) time bound, especially if the extra work did not seem to pay off. So, statements concerning worst-case time bounds for a component merely mean the possibility of implementation of that component with that bound.

All results in this work (except for Section 5.4.1) were produced single-threaded on a Sunfire 15000 with 900 MHz SPARC III+ CPUs, using the operating system SunOS 5.9. We always used the GNU g++ 2.95.3 compiler with the -O4 flag. As it is a multi-processor machine with shared memory, it is slower than a single processor system with the same processor. A comparison of the running times in Section 5.4.1 and in the appendix shows that the machine is approximately half as fast as a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor, which was used in Section 5.4.1.

We used ExpLab [HPKS02] and CVS to address the issue of reproduceability of experiments.

1.3 Definitions and Notations

For any undirected graph $G = (V, E)$, we define $n := |V|$, $m := |E|$, and assume that (v_i, v_j) and (v_j, v_i) denote the same (undirected) edge $\{v_i, v_j\}$. A network is a weighted graph (V, E, c) with an edge weight function $c : E \to \mathbb{R}$. We sometimes refer to networks simply as graphs. For each edge (v_i, v_j), we use terms like cost, weight, length, etc. of (v_i, v_j) interchangeably to denote $c((v_i, v_j))$ (also denoted by $c(v_i, v_j)$ or c_{ij}). For any directed network $\vec{G} = (V, A, c)$, we use $[v_i, v_j]$ to denote

Class Name	Instances	$\|V\|$	Status	Description
D	20	1000	solved	sparse random with varying graph parameters, OR-Library
E	20	2500	solved	
X	3	52-666	solved	complete with Euclidean weights
ES1000FST	15	2532–2984	solved	rectilinear, derived with `geosteiner` [WWZ01] from 1000 (rsp. 10000) random points in the plane, rsp. instances from TSPLIB [Rei91], only other algorithm for solving these instances uses additional information computed in the geometric preprocessing phase (see Section 3.6.1)
ES10000FST	1	27019	solved	
TSPFST	76	89–17127	*solved*	
I080	100	80	solved	incidence networks, constructed with the aim of being difficult for known techniques, introduced by Duin [Dui93].
I160	100	160	solved	
I320	100	320	solved	
I640	100	640	unsolved	
MC	6	97–400	solved	constructed difficult instances
PUC	50	64-4096	unsolved	constructed difficult instances: hypercubes, from code covering and bipartite graph [RdAR$^+$01].
SP	8	6–3997	unsolved	constructed instances, combination of odd wheels and odd circles, difficult for Linear Programming approaches
VLSI	116	90–36711	solved	grid graph with holes (not metric) from VLSI design, SteinLib instance groups `alue`, `alut`, `diw`, `dmxa`, `gap`, `msm`, and `taq`
LIN	37	53–38418	*solved*	grid graph with holes (not metric) from VLSI design
WRP3	63	84–3168	*solved*	wire routing problems from industry [ZR00]
WRP4	62	110–1898	solved	
1R	27	1250	solved	2D cross grid graph [Fre97]
2R	27	2000	solved	3D cross grid graph [Fre97]

Table 1.1: Classes of Problem Instances in [SteinLib]

the directed edge, or arc, from v_i to v_j; and define $a := |A|$.

The **degree** of a vertex $v_i \in V$ is the number of incident edges of v_i. For any network G, $c(G)$ denotes the sum of the edge weights of G.

The **Steiner problem in networks (NSP)** can be formulated as follows: Given a network $G = (V, E, c)$ and a non-empty set R, $R \subseteq V$, of **required vertices** (or **terminals**), find a subnetwork $T_G(R)$ of G that contains a path between every pair of terminals and minimizes $\sum_{(v_i,v_j)\in T_G(R)} c_{ij}$.

We define $r := |R|$. For ease of notation we assume $R = \{v_1, \ldots, v_r\}$. If we want to stress that v_i is a terminal, we will write z_i instead of v_i. The vertices in $V \setminus R$ are called **non-terminals**. Without loss of generality, we assume that the edge weights are positive and that G (and $T_G(R)$) are connected. Now $T_G(R)$ is a tree, called **Steiner minimal tree** (for historical reasons). A **Steiner tree** is an acyclic, connected subnetwork of G, spanning (a superset of) R. We call non-terminals in a Steiner tree its **Steiner nodes**.

The directed version of this problem (also called the Steiner arborescence problem) is defined similarly (see [HRW92]): In addition to G and R, a root $z_1 \in V$ is given and it is required that the solution contains a path from z_1 to every terminal in R. Every instance of the undirected version can be transformed into an instance of the directed version in the corresponding bidirected network, by fixing a terminal z_1 as the root. We define: $R^{z_1} := R \setminus \{z_1\}$.

With $d(v_i, v_j), d_{ij}$ we denote the length of a shortest path between v_i and v_j.

For a given network $G = (V, E, c)$ and $W \subseteq V$, the corresponding distance network is defined as $D_G(W) = (W, W \times W, d)$.

For each terminal z_i, one can define a neighborhood $N(z_i)$ as the set of vertices that are not closer to any other terminal. More precisely, a partition of V is defined:

$$V = \dot{\bigcup_{z_i \in R}} N(z_i) \text{ with } v_j \in N(z_i) \Rightarrow d(v_j, z_i) \leq d(v_j, z_k) \quad \text{(for all } z_k \in R).$$

If $v_j \in N(z_i)$, we call z_i the **base** of v_j (written $base(v_j)$). In accordance with the parlance of algorithmic geometry, we call $N(z_i)$ the **Voronoi region** of z. We consider two terminals z_i and z_j as neighbors if there is an edge (v_k, v_l) with $v_k \in N(z_i)$ and $v_l \in N(z_j)$. Given G and R, the Voronoi regions can be computed in time $O(m + n \log n)$. Using them, a minimum spanning tree for the corresponding distance network $D_G(R)$ (we denote this tree by $T'_D(R)$) can be computed in the same time [Meh88].

1.4 Applications and Background

1.4.1 History

The Steiner problem in networks is the combinatorial variant of the much older Euclidean Steiner problem, which asks for the minimal tree that connects a given set of points in the plane. A special case of this problem has already been discussed before 1640 by Fermat:

Given three points in the plain, find a point that minimizes the sum of the distances to the given points.

Jacob Steiner (1796-1863) considered a generalization of this problem for r points (the **generalized Fermat problem**), but not the (Euclidean) Steiner problem. These two problems are identical only in the case $r = 3$. The "Steiner" problem is believed to have been presented first by Gauß (see [Aro96]). The first (terminating) algorithm for the Euclidean Steiner problem was given by Melzak

[Mel61]. More information on the Euclidean Steiner problem and its history is contained in Hwang et al. [HRW92]. Its relation to the network variant will be discussed later.

The Steiner problem in networks was explicitly formulated for the first time by Hakimi [Hak71] and Levin [Lev71]. Since then hundreds of articles have been published concerning different variants of this problem. A good (although not fully up-to-date) overview is given in Hwang et al. [HRW92].

1.4.2 Related Problems

Euclidean Steiner Problem (ESP)

Definition 1 Given a finite set R of points in the (Euclidean) plane, find a point set S together with a minimal spanning tree T for $R \cup S$, such that T has minimal length concerning the L_2-norm (Euclidean distance).

In comparison to the ESP, the Steiner problem in networks (NSP) is in some sense more general: The cost function can be general and the network does not need to have geometric properties. On the other hand, in the NSP the set of potential Steiner nodes is finite, whereas in the ESP every point in the plain can be part of a feasible solution. But it was proven that a minimal Steiner tree for an ESP instance has at most $r - 2$ ($r = |R|$) Steiner points (nonterminals of degree at least 3) and that the number of possible topologies is finite [HRW92].

Furthermore it holds for every ESP instance I and for every $\epsilon > 0$ that it can be approximated by an NSP instance I_ϵ with fewer than $const \cdot \frac{r^2}{\epsilon^2}$ vertices (i.e., the quotient of the lengths of the optimal solutions of I_ϵ and I is at most $1 + \epsilon$). The basic idea is to put a grid with appropriate granularity on the convex hull of the given points and to use the grid points as possible Steiner points (for details see [HRW92]). Admittedly, the direct application of this method is not practical, as for a guarantee of 1% one needs a complete network with up to $const \cdot 10^4 \cdot r^2$ vertices.

Rectilinear Steiner Problem (RSP)

Definition 2 Given a finite set R of points in the (Euclidean) plane, find a point set S together with a minimal spanning tree T for $R \cup S$, such that T has minimal length concerning the L_1-norm (Manhattan/rectilinear distance).

The similarity between the RSP and the ESP is obvious, but an observation by Hanan [Han66] showed that the RSP is a special case of the NSP:

Draw horizontal and vertical lines through the given points. Define a network $G = (V, E, c)$, with V as the set of all line crossings and E corresponding to the line segments. Hanan showed that an optimal solution in G is an optimal solution for the original RSP instance.

Although it is possible to translate every RSP instance into an NSP instance, this approach has the obvious disadvantage that one loses the knowledge of the special structure of the RSP instance. A refined scheme that uses some geometric based preprocessing before the translation to an NSP instance still loses the geometric information. Nevertheless, in combination with the NSP algorithms presented in this work, it is in many cases the fastest approach for solving an RSP instance (see Section 5.4.1).

We do not discuss the RSP in more detail in this work. For an overview of the very extensive literature for the RSP, see [HRW92, Zac01]. For recent results, see [WWZ00, PV03].

1.4.3 Applications

The different variants of the NSP are among those problems in Combinatorial Optimization that have most applications. It is easy to imagine possible applications. A voluminous book [CD01] is bdevoted to Steiner tree applications. As examples, we describe some applications of the NSP from different fields:

Routing in Computer Networks

In **multiple destination routing** (**MDR**) we are given a network $G = (V, E, c)$ with a **source** $s \in V$ and a set of **destinations** $D \subset V$ $(s \notin D)$. The cost function c can be a complicated function with arguments like delay of a channel, fees, and so on. Two well-known special cases are $|D| = 1$ (**single destination routing**) and $|D| = n - 1$ (**broadcasting**). A **routing tree** T is a subnetwork of G, containing all vertices of $D \cup \{s\}$ and can be viewed as a directed tree rooted at s with leaves from D. An **NC (network cost) optimal routing** asks for a routing tree with minimal total cost.

We present a possible approach for NC optimal routing (for details see [BKJ83]): First, a (minimal) Steiner tree T for G with $R = D \cup \{s\}$ is calculated. The source sends a copy of the message to each of its neighbors in T, together with information on the respective subtree. Each vertex that receives a message repeats this procedure for its subtree.

For a similar application of the NSP in directed networks, see [BD93]. For a distributed variant that uses local information, see [NK94].

VLSI Layout

Several variants of the NSP have numerous applications in VLSI layout. A simple scenario is to find a connection for a set of points on a chip that should carry the same signal.

For some (classical) problems in VLSI layout it may be more appropriate to formulate them as RSP, in many cases also additional constraints (e.g., delay) have to be satisfied or multiple disjoint trees have to be found. But in many applications a formulation as NSP is indicated, e.g., because there are obstacles on the chip (see for example Koch and Martin [KM96]).

For an overview of applications of the Steiner problem in VLSI layout see Korte, Pr¨omel and Steger [KPS90] or Lengauer [Len90].

Phylogeny

Phylogeny is the study of the evolution of life forms. A central problem in Phylogeny is the task of reconstructing an evolutionary tree for a set of (biological) species. One typical variant is the following:

Each given species is represented by some segment of its DNA code. Each DNA sequence is identified with a sequence of m letters of a finite alphabet A (i.e., a vector from A^m). Then we have a (complete) network $G = (V, E, c)$ with $V = A^m$, where the cost function c represents the "distance" between two sequences; in the simplest case this is the Hamming distance of the vectors.

The task is now to find a (minimal) Steiner tree in G, where the set of given species corresponds to the set of terminals.

The literature in this area is very voluminous and deals with a number of very heterogeneous definitions and aims. An overview is given in [HRW92].

1.4.4 Some Effi ciently Solvable Special Cases

Some important special cases of the NSP can be solved very efficiently. This forms the basis for many algorithms for the NSP.

$r = n$: The solution is a minimal spanning tree (MST) of G. It can be obtained in time $O(m + n \log n)$.

$r = 1$: The solution is the given terminal.

$r = 2$: The solution is a shortest path between the two given terminals. It can be obtained in time $O(m + n \log n)$.

$r = 3$: Let $R = \{z_1, z_2, z_3\}$. It is obvious that every minimal Steiner tree has at most one vertex with degree three. Thus, we need to find a vertex v that minimizes $d(v, z_1) + d(v, z_2) + d(v, z_3)$. Let v^* be such a vertex. For the solution calculate shortest paths trees with roots z_1, z_2 and z_3. This can be done in time $O(m + n \log n)$.

Note that for some of the tasks there are algorithms with even better running time guarantees [PR00, KKT95, Tho97], but they are considered to be impractical because of the constant factors hidden in the O-notation.

1.5 Reformulations

Reformulations of a problem are typically used in two cases: Either they are used for simpler proofs of properties of the original problem, or they are used for calculating lower bounds based on linear programming relaxations.

1.5.1 Complete Networks

If it is desirable (e.g., for some simpler proofs), every network G can be transfered to a (with respect to the Steiner problem) equivalent, complete network G^* by introducing new edges with a weight $> c(G)$ between each pair of non-adjacent vertices.

1.5.2 Distance Networks

Lemma 1 Let $D := D_G(V)$ be the distance network of G. It holds that $c(T_G(R)) = c(T_D(R))$.

Proof:
I) $c(T_G(R)) = c(T_{G^*}(R)) \geq c(T_D(R))$, because no edge in D is longer than the corresponding edge in G^*.
II) $c(T_D(R)) \geq c(T_G(R))$, because every edge in $T_D(R)$ can be replaced by a corresponding shortest path in G. □

Corollary 1.1 For each NSP instance there is an equivalent instance that fulfills the triangle inequality. If the preprocessing time for the calculation of all pairs shortest paths (e.g., $O(n^3)$) is acceptable, one can always assume metric graphs.

An explicit transformation of the instance is not advisable in most cases, as one loses many structural properties that could be exploited by algorithms. Furthermore, the necessary time for the calculation of all pairs shortest paths can be very large.

1.5.3 Directed Networks

Definition 3 Let $\vec{G} = (V, A, c)$ be a directed network, and $R \subseteq V$ a nonempty set of terminals with a special root vertex $z_1 \in R$. The Steiner problem in $\vec{G}$ is to find a subnetwork $T_{\vec{G}}(R)$ of $\vec{G}$, such that

1. in $T_{\vec{G}}(R)$ there is a path from z_1 to every other terminal,
2. $c(T_{\vec{G}}(R))$ is minimal.

Every instance of the Steiner problem in an undirected network G can be translated into an instance of the Steiner problem in a directed Network $\vec{G}$: Replace every undirected edge (v_i, v_j), by two directed arcs $[v_i, v_j]$ and $[v_j, v_i]$, each with weight $c(v_i, v_j)$. The root vertex can be chosen arbitrarily from R. Each solution $T_{\vec{G}}(R)$ yields a solution $T_G(R)$ by replacing each directed arc by the corresponding undirected edge.

1.5.4 Minimal Spanning Trees with Degree Constraints

Let $G = (V, E, c)$ with terminals R be an NSP instance. Introduce a new vertex v_0 and connect it with every vertex from $V \setminus R$ and one arbitrary, but fixed $z_1 \in R$ by zero edges. Let $G_0 = (V_0, E_0, c_0)$ be the network resulting from this transformation. We consider a minimal spanning tree (MST) $T_0 = \bar{T}_{G_0}(V_0)$ with the additional constraint that in T_0 every vertex from $V \setminus R$ that is adjacent to v_0 has degree 1.

Lemma 2 Remove v_0 and all incident edges from T_0. The resulting tree $\hat{T}$ is a minimal Steiner tree for R in G.

Proof:
I) $c(\hat{T}) \geq c(T_G(R))$, because $\hat{T}$ is a tree and connects all terminals.
II) $c(\hat{T}) \leq c(T_G(R))$: Add v_0 to $T_G(R)$ and connect v_0 with zero edges to z_1 and all vertices $v_i \notin T_G(R)$. This is a spanning tree for G_0 that satisfies the degree constraints and has cost $c(T_G(R))$. □

A similar reformulation is also possible for the directed version. Let $\vec{G} = (V, A, c)$ be a directed network with terminals R and a root vertex $z_1 \in R$. We introduce an additional vertex v_0 with zero arcs $[v_0, v_i]$ (for all $v_i \in V \setminus R$) and $[v_0, z_1]$. Let $\vec{G}_0 = (V_0, A_0, c_0)$ denote this extended network. With a similar argument as in the last proof, it follows that finding $T_{\vec{G}}(R)$ is essentially the same as finding a directed spanning tree (arborescence) $\vec{T}_0$ with root v_0 in $\vec{G}_0$, such that in $\vec{T}_0$ every descendant of v_0 has degree 1.

1.6 Complexity Results

1.6.1 General Propositions

The decision variant of the Steiner problem in networks (with $c : E \to \mathbb{N}$) is strongly $\mathcal{NP}$-complete (Karp [Kar72]). The optimization variant is $\mathcal{NP}$-hard.

It is still $\mathcal{NP}$-hard to solve the Steiner problem for many important metrics, for example:

- shortest distance in networks (direct consequence of 1.5.2),
- Euclidean distance (Garey, Graham and Johnson [GGJ77]),
- Manhattan distance (Garey and Johnson [GJ77]),
- Hamming distance (Foulds and Graham [FG82]).

1.6.2 Special Networks

$\mathcal{NP}$-hard cases

The Steiner problem is still $\mathcal{NP}$-hard for most important classes of graphs, for example:

bipartite networks, are between R and $V \setminus R$ and have weight 1. The proof is a simple reduction from EXACT-COVER BY 3-SETS, see for example [HRW92].

planar networks (Garey and Johnson [GJ77]), the special case for edge weights 1 is open.

complete networks with edge weights $\{$**1,2**$\}$ (Bern and Plassman [BP89]).

A detailed overview of this is given in Johnson [Joh85].

Polynomially Solvable Cases

As we deal with the general Steiner problem, considering polynomially solvable cases could seem only weakly motivated. But this is not the case. A simple scheme is provided by the previously mentioned cases $r = 2$, $r = 3$ and $r = n$. The very efficient algorithms for these cases are used in nearly every algorithm for the Steiner problem. Other schemes are more pretentious: It could be possible that the transformations of the original instance performed by reduction techniques yield subinstances that are polynomially solvable special cases. For most known polynomially solvable cases (trees, series parallel networks, Halin networks, ...) we found these approaches to be not helpful, but the special case of constant pathwidth could be used fruitfully, as we show in Section 5.2.

For a more comprehensive list of polynomially solvable cases, see [HRW92, Chapter 5].

1.6.3 Approximability

The Steiner problem is $\mathcal{APX}$-complete (Bern and Plassman [BP89]), even if edge weights are only from $\{1, 2\}$. This means there exists $\epsilon > 0$, such that it is $\mathcal{NP}$-hard to find a $(1 + \epsilon)$-approximation for this problem (Arora [Aro94]).

There are a large number of approximation algorithms with constant approximation ratio. Most of the classical heuristics for generating upper bounds guarantee a ratio of close to two.

Better ratios were obtained by a series of algorithms that improved the ratio step by step. The fastest of these algorithm has a ratio of approximately 1.84 and a running time of $O(r(m + n \log n + rn))$ [Zel93, DV97]. But the empirical results of this algorithm were only mediocre and do not justify the comparably long running times.

The most recent result is a ratio of $1 + \ln(3)/2 \approx 1.55$ [RZ00], but the polynomial describing the running time has a k in the exponent that has to go to ∞ to reach this ratio. Thus, the result is hardly of practical relevance. For a recent survey on approximation results, see [GHNP01].

At this point we mention that Arora [Aro96] (and, independently, Mitchell [Mit96]) developed a polynomial time approximation scheme for the Euclidean Steiner problem in $\mathbb{R}^2$ (as well as for a huge number of other $\mathcal{NP}$-hard geometric problems). The methods used are quite general and the proofs hold for all geometric norms L_p $(p \geq 1)$.

Chapter 2

Theoretical and Practical Aspects of Linear Programming Relaxations for the Steiner Problem

2.1 Introduction

This chapter deals with theoretical and practical aspects of linear programming approaches for the Steiner problem. The basic idea is to reformulate the given problem as an integer linear program. Then, the linear relaxation of these reformulation can be solved (or approximated). This is useful, not only for computing lower bounds, but also as the basis of many empirically successful heuristics for computing upper bounds and sophisticated reduction techniques, culminating in an exact algorithm, which achieves very good empirical results.

There are many (mixed) integer programming formulations of the Steiner problem in networks. Although the strength of the corresponding linear relaxation has great impact on the practical effectiveness of the algorithms that are based on it, not much was known about the relative quality of these relaxations. In Sections 2.3 to 2.8, we compare the linear relaxations of all classical, frequently cited and some modified or new integer programming formulations of this problem from a theoretical point of view with respect to their optimal values. We present several new results, establishing clear relations between relaxations, which have often been treated as unrelated or incomparable.

In Section 2.7, we introduce a collection of new relaxations that are stronger than any relaxation we are aware of, and place the new relaxations into our hierarchy. Further, we show how one relaxation of the collection can be used in practical algorithms. Except for the flow-balance constraints introduced to the Steiner problem by [KM98], this is the first successful attempt to use a relaxation that is stronger than Wong's directed cut relaxation (Section 2.3) from 1984 [Won84].

In Section 2.9, we report on our experimental study of some of these relaxations. Their previously mentioned algorithmic application for upper bound calculation, reduction techniques, and in the exact algorithm will be explained in detail in the corresponding chapters.

2.2 Additional Definitions for Lower Bounds

Here we recall two reformulations of the Steiner problem from Section 1.5, because they are used in some relaxations. One uses the directed version: Given $G = (V, E, c)$ and R, find a minimum weight arborescence in $\vec{G} = (V, A, c)$ ($A := \{[v_i, v_j], [v_j, v_i] \mid (v_i, v_j) \in E\}$, c defined accordingly) with a terminal (say z_1) as the root that spans $R^{z_1} := R \backslash \{z_1\}$.

The problem can also be stated as finding a degree-constrained minimum spanning tree T_0 in a modified network $G_0 = (V_0, E_0, c_0)$, produced by adding a new vertex v_0 and connecting it through zero cost edges to all vertices in $V \backslash R$ and to a fixed terminal (say z_1). The problem is now equivalent to finding a minimum spanning tree T_0 in G_0 with the additional restriction that in T_0 every vertex in $V \backslash R$ adjacent to v_0 must have degree one. For more details on this reformulation, see [BP87, Bea89]. Again, a similar directed version for a network $\vec{G}_0$ can be defined, this time by adding zero cost arcs $[v_0, v_i]$ (for all $v_i \in V \backslash R$) and $[v_0, z_1]$ to $\vec{G}$.

We introduce some additional definitions that make the notation of linear relaxations easier to understand.

A **cut** in $\vec{G} = (V, A, c)$ (or in $G = (V, E, c)$) is defined as a partition $C = \{\overline{W}, W\}$ of V ($\emptyset \subset W \subset V; V = W \dot{\cup} \overline{W}$). We use $\delta^-(W)$ to denote the set of arcs $[v_i, v_j] \in A$ with $v_i \in \overline{W}$ and $v_j \in W$. For simplicity, we write $\delta^-(v_i)$ instead of $\delta^-(\{v_i\})$. The sets $\delta^+(W)$ and, for the undirected version, $\delta(W)$ are defined similarly. A cut $C = \{\overline{W}, W\}$ is called a **Steiner cut** if $z_1 \in \overline{W}$ and $R^{z_1} \cap W \neq \emptyset$ (for the undirected version: $R \cap W \neq \emptyset$ and $R \cap \overline{W} \neq \emptyset$).

In the integer programming formulations we use (binary) variables x_{ij} for each arc $[v_i, v_j] \in A$ (respectively X_{ij} for each edge $(v_i, v_j) \in E$), indicating whether an arc is part of the solution ($x_{ij} =$

1) or not ($x_{ij} = 0$). Thus, the cost of the solution can be calculated by the dot product $c \cdot x$, where c is the cost vector. For any $B \subseteq A$, $x(B)$ is short for $\sum_{a \in B} x_a$, and $A(W)$ denotes $\{[v_i, v_j] \in A | v_i, v_j \in W\}$ for any $W \subseteq V$. For example, $x(\delta^-(W))$ is short for $\sum_{[v_i,v_j] \in A, v_i \notin W, v_j \in W} x_{ij}$.

Let P_1 be an integer linear program. The corresponding linear relaxation is denoted by LP_1. The dual of such a relaxation is denoted by DLP_1 and a Lagrangian relaxation by $LaLP_1$. The value of an optimal solution of the integer programming formulation (for given $\vec{G}$ and R), denoted by $v(P_1)$, is of course the value of an optimal solution of the corresponding Steiner arborescence problem in $\vec{G}$. Thus, in this context we are interested in the optimal value $v(LP_1)$ of the corresponding linear relaxation, which can differ from $v(P_1)$. The notations P_1 (or LP_1) always denote an integer (or linear) program corresponding to an arbitrary, but fixed instance (G, R) of the Steiner problem (with G replaced by $\vec{G}$, G_0 or $\vec{G}_0$ when appropriate).

For a linear program P_x, the identifier x describes the program (see Table 2.1 for further explanation of the abbreviations for linear programs).

C	cut based	Section 2.3.1
F	flow based	Section 2.3.2
F^R	common-flow formulation	Section 2.7.3
F^{j_1,j_2}	restricted version of F^R (multiple roots)	Section 2.7.4
F^2	restricted version of F^R (only one root)	Section 2.7.4
$2T$	two-terminal formulation	Section 2.3.2
T	tree based	Section 2.4
mT	multiple tree based	Section 2.6
T_0	degree constraint spanning tree based (with v_0 vertex)	Section 2.4.1
$\vec{T}$	directed version of T	
U	undirected version	
$X + FB$	with added flow-balance constraints	Section 2.6.2
$X-$	weaker version of X	
$X++$	aggregated version of X	
X'	modified version of X	

Table 2.1: Abbreviations for linear programs and their meaning.

We compare relaxations using the predicates **equivalent** and (**strictly**) **stronger**: We call a relaxation R_1 stronger than a relaxation R_2 if the optimal value of R_1 is no less than that of R_2 for all instances of the problem. If R_2 is also stronger than R_1, we call them equivalent, otherwise we say that R_1 is strictly stronger than R_2. If neither is stronger than the other, they are **incomparable**.

2.3 Cut and Flow Formulations

In this section, we state the basic flow and cut-based formulations of the Steiner problem. There are some well-known observations concerning these formulations, which we cite without proof.

2.3.1 Cut Formulations

The directed cut (or dicut) formulation was stated in [Won84].

$$\begin{array}{rcll} \boxed{P_C} \quad c \cdot x & \rightarrow & \min, & \\ x(\delta^-(W)) & \geq & 1 \quad (z_1 \notin W, R \cap W \neq \emptyset), & (1.1) \\ x & \in & \{0,1\}^{|A|}. & (1.2) \end{array}$$

The constraints (1.1) are called Steiner cut constraints. They guarantee that in any arc set corresponding to a feasible solution, there is a path from z_1 to any other terminal.

A formulation for the undirected version was stated in [Ane80]:

$$\begin{array}{rcll} \boxed{P_{UC}} \quad c \cdot X & \rightarrow & \min, & \\ X(\delta(W)) & \geq & 1 \quad (W \cap R \neq R,\ W \cap R \neq \emptyset), & (2.1) \\ X & \in & \{0,1\}^{|E|}. & (2.2) \end{array}$$

Lemma 3 LP_C is strictly stronger than LP_{UC}; and $\sup\left\{\frac{v(LP_C)}{v(LP_{UC})}\right\} = 2$ [CR94a, Dui93].

We just mention here that $\frac{v(P_{UC})}{v(LP_{UC})} \leq 2$ [GB93]; and that when applied to undirected instances, the value $v(LP_C)$ is independent of the choice of the root [GM93]. For much more information on LP_C, LP_{UC} and their relationship, see [CR94a]. Also, many related results are discussed in [MW95].

2.3.2 Flow Formulations

Viewing the Steiner problem as a multicommodity flow problem leads to the following formulation (see [Won84]).

$$\begin{array}{rcll} \boxed{P_F} \quad c \cdot x & \rightarrow & \min, & \\ y^t(\delta^-(v_i)) & = & y^t(\delta^+(v_i)) - \left\{ \begin{array}{ll} 1 & (z_t \in R^{z_1}\ ;\ v_i = z_t), \\ 0 & (z_t \in R^{z_1}\ ;\ v_i \in V \setminus \{z_1, z_t\}), \end{array} \right. & (3.1) \\ y^t & \leq & x \quad (z_t \in R^{z_1}), & (3.2) \\ y^t & \geq & 0 \quad (z_t \in R^{z_1}), & (3.3) \\ x & \in & \{0,1\}^{|A|}. & (3.4) \end{array}$$

Each variable y^t_{ij} denotes the quantity of the commodity t flowing through $[v_i, v_j]$. Constraints (3.1) guarantee that for each terminal $z_t \in R^{z_1}$, there is a flow of one unit of commodity t from z_1 to z_t. Together with (3.2), they guarantee that in any arc set corresponding to a feasible solution, there is a path from z_1 to any other terminal.

Lemma 4 LP_C is equivalent to LP_F [Won84].

The correspondence is even stronger: Every feasible solution x for LP_C corresponds to a feasible solution (x, y) for LP_F.

The straightforward translation of P_F for the undirected version leads to LP_{UF} with $v(LP_{UF}) = v(LP_{UC})$ (see [GM93]). There are other undirected formulations (see [GM93]), leading to relaxations that are all equivalent to LP_F; so we use the notation LP_{FU} for all of them.

Of course, there is no need for different commodities in P_F. In an aggregated version, which we call P_{F++}, one unit of a single commodity flows from z_1 to each terminal $z_t \in R^{z_1}$ (see [Mac87]). This program has only $\Theta(a)$ variables and constraints, which is asymptotically minimal. But the corresponding linear relaxation LP_{F++} is not a strong one:

$$\boxed{P_{F++}} \qquad c \cdot x \;\rightarrow\; \min,$$

$$Y(\delta^-(v_i)) \;=\; Y(\delta^+(v_i)) - \begin{cases} 1 & (v_i \in R^{z_1}), \\ 0 & (v_i \in V \setminus R), \end{cases} \tag{4.1}$$

$$(r-1)x \;\geq\; Y, \tag{4.2}$$

$$Y \;\geq\; 0, \tag{4.3}$$

$$x \;\in\; \{0,1\}^{|A|}. \tag{4.4}$$

The variables Y describe a flow of one unit from z_1 to each terminal in R^{z_1}.

Lemma 5 LP_F is strictly stronger than LP_{F++}. The worst-case ratio $\frac{v(LP_F)}{v(LP_{F++})}$ is $r-1$ [Mac87, Dui93].

In [Liu90], the two-terminal formulation was stated:

$$\boxed{P_{2T}} \qquad c \cdot x \;\rightarrow\; \min,$$

$$\check{y}^{kl}(\delta^-(v_i)) - \check{y}^{kl}(\delta^+(v_i)) \;\geq\; \begin{cases} -1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_1), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1\}), \end{cases} \tag{5.1}$$

$$(\check{y}^{kl} + \grave{y}^{kl})(\delta^-(v_i)) - (\check{y}^{kl} + \grave{y}^{kl})(\delta^+(v_i)) \;=\; \begin{cases} 1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_k), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1, z_k\}), \end{cases} \tag{5.2}$$

$$(\check{y}^{kl} + \acute{y}^{kl})(\delta^-(v_i)) - (\check{y}^{kl} + \acute{y}^{kl})(\delta^+(v_i)) \;=\; \begin{cases} 1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_l), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1, z_l\}), \end{cases} \tag{5.3}$$

$$\check{y}^{kl} + \acute{y}^{kl} + \grave{y}^{kl} \;\leq\; x \quad (\{z_k, z_l\} \subseteq R^{z_1}), \tag{5.4}$$

$$\check{y}^{kl},\ \acute{y}^{kl},\ \grave{y}^{kl} \;\geq\; 0 \quad (\{z_k, z_l\} \subseteq R^{z_1}), \tag{5.5}$$

$$x \;\in\; \{0,1\}^{|A|}. \tag{5.6}$$

The formulation P_F is based on the flow formulation of the shortest path problem (the special case of the Steiner problem with $|R^{z_1}| = 1$). The formulation P_{2T} is based on the special case with $|R^{z_1}| = 2$, namely the two-terminal (2T) Steiner arborescence problem. In a Steiner tree, for any two terminals $z_k, z_l \in R^{z_1}$, there is a two-terminal tree consisting of a path from z_1 to a splitter node v_s and two paths from v_s to z_k and z_l (v_s can belong to $\{z_1, z_k, z_l\}$). In P_{2T}, $\check{y}$, $\grave{y}$ and $\acute{y}$ describe flows from z_1 to v_s, from v_s to z_k and from v_s to z_l. Note that the flow described by $\check{y}$ can have an excess at some vertices (because of the inequality in (5.1)), this excess is carried by the flows described by $\grave{y}$ and $\acute{y}$ to z_k and z_l (because of (5.2) and (5.3)).

Lemma 6 LP_{2T} is strictly stronger than LP_F [Liu90].

2.4 Tree Formulations

In this section, we state the basic tree-based formulations and prove that the corresponding linear relaxations are all equivalent. We also discuss some variants from the literature, which we prove to be weaker.

2.4.1 Degree-Constrained Tree Formulations

In [Bea89], the following program was suggested, which is a translation of the degree-constrained minimum spanning tree problem in G_0.

$$\begin{aligned}
\boxed{P_{T_0}} \qquad c \cdot X &\rightarrow \min, \\
\{(v_i, v_j) \mid X_{ij} = 1\} &: \text{ builds a spanning tree for } G_0, && (6.1)\\
X_{0k} + X_{ki} &\leq 1 \quad (v_k \in V \backslash R\,;\ (v_k, v_i) \in \delta(v_k)), && (6.2)\\
X &\in \{0,1\}^{|E_0|}. && (6.3)
\end{aligned}$$

The requirement (6.1) can be stated by linear constraints. In the following, we assume that (6.1) is replaced by the following constraints.

$$\begin{aligned}
X(E_0) &= n, && (6.4)\\
X(E_0(W)) &\leq |W| - 1 \quad (\emptyset \neq W \subset V_0). && (6.5)
\end{aligned}$$

The constraints (6.4) and (6.5), together with the nonnegativity of X, define a polyhedron whose extreme points are the incidence vectors of spanning trees in G_0 (see [Edm71, MW95]). Thus, no other set of linear constraints replacing (6.1) can lead to a stronger linear relaxation.

A directed version can be stated as follows.

$$\begin{aligned}
\boxed{P_{\vec{T}_0}} \qquad c \cdot x &\rightarrow \min, \\
x(\delta^-(v_i)) &= 1 \quad (v_i \in V), && (7.1)\\
x(A_0(W)) &\leq |W| - 1 \quad (\emptyset \neq W \subseteq V_0), && (7.2)\\
x_{0i} + x_{ij} + x_{ji} &\leq 1 \quad (v_i \in V \setminus R\,;\ [v_i, v_j] \in \delta^+(v_i)\ ), && (7.3)\\
x &\in \{0,1\}^{|A_0|}. && (7.4)
\end{aligned}$$

Again, the constraints (7.1) and (7.2), together with the nonnegativity of x, define a polyhedron whose extreme points are the incidence vectors of spanning arborescences with root v_0 (see [MW95]). Note that $\delta^-(v_0) = \emptyset$ by the construction of $\vec{G}_0$.

In the literature on the Steiner problem, one usually finds a directed variant $P_{\vec{T}_0-}$ that uses

$$x_{0i} + x_{ij} \leq 1 \quad (v_i \in V \setminus R\,;\ [v_i, v_j] \in \delta^+(v_i)\ )$$

instead of the constraints (7.3) (see for example [HRW92]). Obviously $v(P_{\vec{T}_0-}) = v(P_{\vec{T}_0})$, and $v(LP_{\vec{T}_0-}) \leq v(LP_{\vec{T}_0})$. The following example shows that $LP_{\vec{T}_0}$ is strictly stronger than the version in the literature.

Example 1 Figure 2.1 shows the network $\vec{G}$ with $R = \{z_1, z_2\}$, $\gamma \geq 100$ and the network $\vec{G}_0$. The minimum Steiner arborescence has the value $\gamma + 10$.
The following $\dot{x}$ is feasible (and optimal) for $LP_{\vec{T}_0-}$ and gives the value 11: $\dot{x}_{01} = 1$, $\dot{x}_{03} = \dot{x}_{04} = \dot{x}_{34} = \dot{x}_{43} = \dot{x}_{32} = \dot{x}_{42} = \frac{1}{2}$ and $\dot{x}_{ij} = 0$ (for all other arcs). But for $LP_{\vec{T}_0}$, $\dot{x}$ is infeasible. The optimal value here is: $v(LP_{\vec{T}_0}) = \frac{\gamma}{3} + 14$ (this value is reached for example by $\hat{x}$ with $\hat{x}_{01} = 1$, $\hat{x}_{03} = \hat{x}_{04} = \hat{x}_{13} = \hat{x}_{23} = \hat{x}_{32} = \frac{1}{3}$, $\hat{x}_{42} = \hat{x}_{34} = \frac{2}{3}$ and $\hat{x}_{ij} = 0$ (for all other arcs)). So the ratio $v(LP_{\vec{T}_0-})/v(LP_{\vec{T}_0})$ can be arbitrarily close to 0.

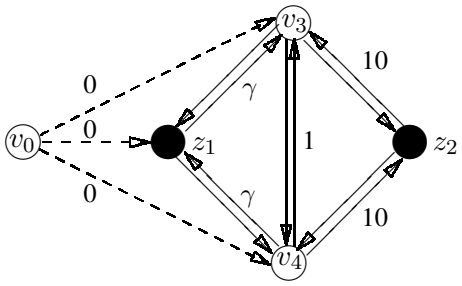


Figure 2.1: Example with $v(LP_{\vec{T}_0-}) \ll v(LP_{\vec{T}_0}) = v(LP_{T_0}) \ll v(P_{T_0})$.

2.4.2 Rooted Tree Formulation

The rooted tree formulation is stated, for example, in [KPH93]:

$$
\begin{aligned}
\boxed{P_{\vec{T}}} \qquad c \cdot x &\rightarrow \min, \\
x(\delta^-(v_i)) &= 1 \quad (v_i \in R^{z_1}), && (8.1)\\
x(\delta^-(v_i) \setminus \{[v_j, v_i]\}) &\geq x_{ij} \quad (v_i \in V \setminus R\,;\, [v_i, v_j] \in \delta^+(v_i)), && (8.2)\\
x(A(W)) &\leq |W| - 1 \quad (\emptyset \neq W \subseteq V), && (8.3)\\
x &\in \{0,1\}^{|A|}. && (8.4)
\end{aligned}
$$

To get rid of the exponential number of constraints for avoiding cycles, many authors have considered replacing (8.3) by the subtour elimination constraints introduced in the TSP-context (known as the Miller-Tucker-Zemlin constraints [MTZ60]), allowing additional variables t_i for all $v_i \in V$:

$$t_i - t_j + n x_{ij} \leq n - 1 \qquad ([v_i, v_j] \in A). \tag{8.5}$$

This leads to the program $P_{\vec{T}-}$ with $\Theta(a)$ variables and constraints, which is asymptotically minimal. The linear relaxation $LP_{\vec{T}-}$ was used by [KP95]. We will now prove the intuitive guess that $LP_{\vec{T}}$ is stronger than $LP_{\vec{T}-}$. Indeed, the ratio $\frac{v(LP_{\vec{T}-})}{v(LP_{\vec{T}})}$ can be arbitrarily close to 0 (see Figure 2.2 on page 30).

Lemma 7 $v(LP_{\vec{T}-}) \leq v(LP_{\vec{T}})$.

Proof: Let $\hat{x}$ denote an (optimal) solution for $LP_{\vec{T}}$. Obviously $\hat{x}$ satisfies the constraints (8.1) and (8.2). We now show that it is possible to construct $\hat{t}$ such that $(\hat{x}, \hat{t})$ satisfies (8.5), too.
We start with an arbitrary $\hat{t}$ (e.g., $\hat{t}_i = 0$ (for all $v_i \in V$)). We define for every arc $[v_i, v_j] \in A$: $s_{ij} := (n-1) - (\hat{t}_i - \hat{t}_j + n\hat{x}_{ij})$; and call an arc $[v_i, v_j]$ *good*, if $s_{ij} \geq 0$; *used*, if $s_{ij} \leq 0$; and *bad*, if $s_{ij} < 0$. Suppose $[v_i, v_j]$ is a bad arc (if no bad arcs exist, $(\hat{x}, \hat{t})$ satisfies (8.5)).
We now show how $\hat{t}_j$ (and perhaps some other $\hat{t}_p$) can be increased in a way that $[v_i, v_j]$ becomes good, but no good arc becomes bad. By repeating this procedure we can make all arcs good and prove the lemma.
In each step we denote by W_j the set of vertices $v_k \in V$ that can be reached from v_j through paths with only used arcs. We define Δ as $\min\{s_{kl} \mid [v_k, v_l] \in \delta^+(W_j)\}$, if this set is nonempty, and ∞ otherwise. Now we increase for all vertices $v_p \in W_j$ the variables $\hat{t}_p$ by $\min\{-s_{ij}, \Delta\}$ (these values can change in every step). By doing this, no arc of $\delta^+(W_j)$ becomes bad. For arcs $[v_p, v_q]$ with $v_p, v_q \in W_j$ or $v_p, v_q \notin W_j$ the value of s_{pq} does not change; and for arcs $[v_q, v_p] \in \delta^-(W_j)$ s_{qp} does

not decrease.
Because $\hat{t}_j$ is increased in every step, there is only one situation that could prevent that $[v_i, v_j]$ becomes good: In one step v_i is absorbed by W_j. But then, according to the definition of W_j, there exists a path $v_j \rightsquigarrow v_i$ with only used arcs. Thus, there exists a cycle $C := (v_i, v_j = v_{k_1}, \ldots, v_{k_l} = v_i)$, with $s_{k_l k_1} < 0$ and $s_{k_{t-1} k_t} \leq 0$ (for all $t \in \{2, \ldots, l\}$). Summation of the inequalities for arcs on the cycle C leads to: $n\hat{x}(C) > l(n-1)$. On the other hand, since $\hat{x}$ satisfies the constraints (8.3), $\hat{x}(C) \leq l-1$. The consequence, $\frac{l-1}{l} > \frac{n-1}{n}$, is a contradiction. □

2.4.3 Equivalence of Tree-Class Relaxations

We now show the equivalence of the tree-based relaxations $LP_{T_0}, LP_{\vec{T}_0}$, and $LP_{\vec{T}}$.

Lemma 8 $v(LP_{\vec{T}_0}) = v(LP_{T_0})$.

Proof:
I) $v(LP_{\vec{T}_0}) \geq v(LP_{T_0})$: Let x denote an (optimal) solution for $LP_{\vec{T}_0}$. Define X with $X_{ij} := x_{ij} + x_{ji}$ (for all $(v_i, v_j) \in E$), $X_{0i} := x_{0i}$ (for all $v_i \in V \setminus R$) and $X_{01} := x_{01}$. It is easy to check that X satisfies all constraints of LP_{T_0} and yields the same value as $v(LP_{\vec{T}_0})$.
II) $v(LP_{T_0}) \geq v(LP_{\vec{T}_0})$: Now let X denote an (optimal) solution for LP_{T_0}. Define Δ with $\Delta_{ij} \in [0,1]$ arbitrarily (for all $(v_i, v_j) \in E$) and set x to $x_{ij} := \Delta_{ij} X_{ij}$, $x_{ji} := (1-\Delta_{ij}) X_{ij}$ (for all $(v_i, v_j) \in E$), $x_{0i} := X_{0i}$ (for all $v_i \in V \setminus R$) and $x_{01} := X_{01}$. Again, it is easy to validate that x satisfies the constraints (7.2) and (7.3) and yields the same value as $v(LP_{T_0})$.
The only question is, whether there is a Δ such that x satisfies the constraints (7.1), too. This question can be stated in the following way:
Is it possible to distribute the "supply" X_{ij} of each edge (v_i, v_j) in such a way to its end-vertices that every vertex $v_i \in V$ gets one unit at the end?
It is known that this problem can be viewed as a flow problem: Construct a flow network with source s, sink t, and vertices u_{ij} (for all $(v_i, v_j) \in E_0$) and u_i (for all $v_i \in V_0$). Every u_{ij} is connected with u_i and u_j through arcs $[u_{ij}, u_i]$ and $[u_{ij}, u_j]$ with capacity ∞. Furthermore, there are arcs $[s, u_{ij}]$ with capacity X_{ij} and arcs $[u_i, t]$ with capacity 1 (or 0, if $i = 0$). The question above is equivalent to the question, whether a flow from s to t with value n can be constructed. The max-flow min-cut theorem says that this is possible if and only if there is no cut $C = \{U, \overline{U}\}$ (with $s \in U$ and $t \notin U$) with capacity less than n (Obviously $U = \{s\}$ and $U = V \setminus \{t\}$ correspond to cuts with capacity n).
Suppose that U corresponds to a cut C with minimum capacity. Define $W := \{v_i \in V_0 \mid u_i \in U\}$, $E_W := \{(v_i, v_j) \in E_0 \mid v_i, v_j \in W\}$, and $E_U := \{(v_i, v_j) \in E_0 \mid u_{ij} \in U\}$. For every $[v_i, v_j] \in E_U$ ($u_{ij} \in U$), u_i and u_j must belong to U ($[v_i, v_j] \in E_W$), because otherwise the capacity of C would be ∞, which is not minimal. It follows that: $E_U \subseteq E_W$.
The capacity of C is:

$$\begin{aligned} |W \setminus \{v_0\}| + X(E_0 \setminus E_U) &\geq |W \setminus \{v_0\}| + X(E_0 \setminus E_W) && \text{(since } E_U \subseteq E_W\text{)} \\ &\geq |W| - 1 + X(E_0) - X(E_W) \\ &= |W| - 1 + n - X(E_W) && \text{(because of 6.4)} \\ &\geq n. && \text{(because of 6.5)} \end{aligned}$$

It follows that the minimal cut has capacity of n. □

Lemma 9 $v(LP_{\vec{T}}) = v(LP_{\vec{T}_0})$.

Proof:
I) $v(LP_{\vec{T}_0}) \geq v(LP_{\vec{T}})$: Let $\hat{x}$ denote an (optimal) solution for $LP_{\vec{T}_0}$. Define $\tilde{x}$ with $\tilde{x}_{ij} := \hat{x}_{ij}$ (for all $[v_i, v_j] \in A$). Because $\hat{x}$ satisfies the constraints (7.1) and in $\vec{G}_0$ only arcs in A are incident with terminals in R^{z_1}, $\tilde{x}$ satisfies the constraints (8.1).
Furthermore, $\tilde{x}$ satisfies the constraints (8.2), because for every arc $[v_i, v_j] \in A$ with $v_i \in V \setminus R$ it holds that:

$$\begin{aligned}
\tilde{x}(\delta^-(v_i) \setminus \{[v_j, v_i]\}) &= \tilde{x}(\delta^-(v_i)) - \tilde{x}_{ji} && (\delta \text{ in } \vec{G}) \\
&= \hat{x}(\delta^-(v_i)) - \hat{x}_{0i} - \hat{x}_{ji} && (\delta \text{ in } \vec{G}_0) \\
&= 1 - \hat{x}_{0i} - \hat{x}_{ji} && (\text{because of } (7.1)\ ) \\
&\geq \hat{x}_{ij} && (\text{because of } (7.3)\ ) \\
&= \tilde{x}_{ij}.
\end{aligned}$$

Finally $\tilde{x}$ satisfies (8.3), because $\hat{x}$ satisfies (7.2).
II) $v(LP_{\vec{T}}) \geq v(LP_{\vec{T}_0})$: Let $\tilde{x}$ denote an (optimal) solution for $LP_{\vec{T}}$. Define $\hat{x}$ with $\hat{x}_{ij} := \tilde{x}_{ij}$ (for all $[v_i, v_j] \in A$) and $\hat{x}_{0i} := 1 - \tilde{x}(\delta^-(v_i))$ (for all $v_i \in V \setminus R^{z_1}$). Notice that for an optimal $\tilde{x}$, $\tilde{x}(\delta^-(v_i)) > 1$ could only be forced by (8.2) for some arc $[v_i, v_l]$ with $\tilde{x}(\delta^-(v_i) \setminus \{[v_j, v_i]\}) = \tilde{x}_{il}$, and it would follow that $1 < \tilde{x}(\delta^-(v_i)) = \tilde{x}_{li} + \tilde{x}_{il}$, but this is excluded by (8.3) (for $W = \{v_i, v_l\}$). So $\hat{x}$ satisfies (7.1) in a trivial way.
The constraints (7.2) are satisfied by $\hat{x}$ for every $W \subseteq V$, because $\tilde{x}$ satisfies (8.3). For $W \subseteq V_0$ with $v_0 \in W$ it holds that:

$$\begin{aligned}
\hat{x}(A_0(W)) &\leq \sum_{v_i \in W \setminus \{v_0\}} \hat{x}(\delta^-(v_i)) && (\text{in } \vec{G}_0) \\
&= \sum_{v_i \in W \setminus \{v_0\}} 1 && (\text{because of } (7.1)\ ) \\
&= |W| - 1.
\end{aligned}$$

Finally for every $[v_i, v_j] \in A$ with $v_i \in V \setminus R$:

$$\begin{aligned}
\hat{x}_{0i} + \hat{x}_{ij} + \hat{x}_{ji} &= 1 - \tilde{x}(\delta^-(v_i)) + \tilde{x}_{ij} + \tilde{x}_{ji} && (\text{in } \vec{G}) \\
&= 1 - \tilde{x}(\delta^-(v_i) \setminus \{[v_j, v_i]\}) + \tilde{x}_{ij} \\
&\leq 1. && (\text{because of } (8.2)\ )
\end{aligned}$$

Thus, $\hat{x}$ also satisfies the constraints (7.3). □

2.5 Relationship between the Two Classes

In this section, we settle the question of the relationship between flow and tree-based relaxations by proving that LP_C is strictly stronger than $LP_{\vec{T}}$. Our proofs also show that LP_C cannot be strengthened by adding constraints that are present in $LP_{\vec{T}_0}$ or $LP_{\vec{T}}$.
First, we show that every (optimal) solution $\hat{x}$ of LP_C has certain properties:

Lemma 10 For every (optimal) solution $\hat{x}$ of LP_C, $W \subseteq V \setminus \{z_1\}$ and $v_k \in W$ the following holds:

$$\hat{x}(\delta^-(W)) \geq \hat{x}(\delta^-(v_k)).$$

Proof: Suppose that $\hat{x}$ violates the inequality for some W and v_k. Among all such inequalities, choose one for which $|W|$ is minimal. For this inequality to be violated, there must be an arc $[v_l, v_k] \in \delta^-(v_k) \backslash \delta^-(W)$ with $\hat{x}_{lk} > 0$. Because of the optimality of $\hat{x}$, $\hat{x}_{lk}$ cannot be decreased without violating a Steiner cut constraint, so there is a $U \subset V$ with $z_1 \notin U$, $U \cap R \neq \emptyset$, $[v_l, v_k] \in \delta^-(U)$, and $\hat{x}(\delta^-(U)) = 1$. Now one has the inequality †:

$$\begin{aligned} \hat{x}(\delta^-(U)) + \hat{x}(\delta^-(W)) &= \hat{x}(\delta^-(U \cup W)) + \hat{x}(\delta^-(U \cap W)) + \\ &\quad \hat{x}(\{[v_i, v_j] \in A \mid v_i \in W \backslash U, v_j \in U \backslash W) + \\ &\quad \hat{x}(\{[v_j, v_i] \in A \mid v_i \in W \backslash U, v_j \in U \backslash W) \\ &\geq \hat{x}(\delta^-(U \cup W)) + \hat{x}(\delta^-(U \cap W)) \end{aligned}$$

Since $z_1 \notin U \cup W$ and $(U \cup W) \cap R \neq \emptyset$, $U \cup W$ corresponds to a Steiner cut, and $\hat{x}(\delta^-(U \cup W)) \geq 1 = \hat{x}(\delta^-(U))$. Using †, one obtains: $\hat{x}(\delta^-(W)) \geq \hat{x}(\delta^-(U \cap W))$. This implies that $\hat{x}$ also violates the lemma for $U \cap W$ and v_k. Since $v_l \in W \backslash U$, we have $|U \cap W| < |W|$, and this contradicts the minimality of W.[1] □

Lemma 11 For every (optimal) solution $\hat{x}$ of LP_C and $v_k \in V \backslash \{z_1\}$ the following holds:

$$\hat{x}(\delta^-(v_k)) \leq 1.$$

Proof: Suppose $\hat{x}$ violates the inequality for v_k. There is an arc $[v_l, v_k] \in \delta^-(v_k)$ with $\hat{x}_{lk} > 0$. Because of the optimality of $\hat{x}$, $\hat{x}_{lk}$ cannot be decreased without violating a Steiner cut constraint, so there is a $W \subset V$ with $z_1 \notin W$, $W \cap R \neq \emptyset$, $[v_l, v_k] \in \delta^-(W)$, and $\hat{x}(\delta^-(W)) = 1$. Together with Lemma 10 (for v_k and W), one gets a contradiction. □

Lemma 12 For every (optimal) solution $\hat{x}$ of LP_C, $v_l \in V \backslash \{z_1\}$, and $[v_l, v_k] \in A$ the following holds:

$$\hat{x}(\delta^-(v_l) \setminus \{[v_k, v_l]\}) \geq \hat{x}_{lk}.$$

Proof: This follows directly from Lemma 10 (for v_k and $W = \{v_l, v_k\}$) by subtracting $\hat{x}(\delta^-(v_l) \setminus \{[v_k, v_l]\})$ from both sides. Note that the special case $v_k = z_1$ is trivial, because $\hat{x}_{l1} = 0$ in every optimal solution. □

Theorem 13 $v(LP_{\vec{T}}) \leq v(LP_C)$.

Proof: Let $\hat{x}$ be an (optimal) solution for LP_C. We will show that $\hat{x}$ is feasible for $LP_{\vec{T}}$:
Because $\{v_i\}$ corresponds to a Steiner cut for $v_i \in R^{z_1}$, through the use of Lemma 11, $\hat{x}$ satisfies (8.1).
Because of Lemma 12, $\hat{x}$ satisfies (8.2).
Let $W \subseteq V$ be a nonempty set. If $z_1 \in W$:

$$\begin{aligned} \hat{x}(A(W)) &\leq \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) \\ &= \sum_{v_i \in W \backslash \{z_1\}} \hat{x}(\delta^-(v_i)) \qquad \text{(optimality of } \hat{x}) \\ &\leq \sum_{v_i \in W \backslash \{z_1\}} 1 \qquad \text{(Lemma 11)} \\ &= |W| - 1. \end{aligned}$$

[1] In a different context this argumentation was used in [GM93].

Now we assume $z_1 \notin W$ and define $\Delta := \hat{x}(\delta^-(W))$. There are two cases:
I) $\Delta \geq 1$:

$$\begin{aligned}
\hat{x}(A(W)) &= \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - \hat{x}(\delta^-(W)) \\
&\leq \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - 1 \qquad (\Delta \geq 1) \\
&\leq \sum_{v_i \in W} 1 \quad - 1 \qquad \text{(Lemma 11)} \\
&= |W| - 1.
\end{aligned}$$

II) $\Delta < 1$:

$$\begin{aligned}
\hat{x}(A(W)) &= \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - \hat{x}(\delta^-(W)) \\
&\leq \sum_{v_i \in W} \hat{x}(\delta^-(W)) - \hat{x}(\delta^-(W)) \qquad \text{(Lemma 10)} \\
&= (|W| - 1)\hat{x}(\delta^-(W)) \\
&< |W| - 1. \qquad (\Delta < 1)
\end{aligned}$$

It follows that $\hat{x}$ satisfies (8.3) too. □

Corollary 13.1 The proof shows that adding constraints of $LP_{\vec{T}}$ to LP_C cannot improve $v(LP_C)$.

Corollary 13.2 Because the proofs of the equivalence of the tree relaxations require the optimality only in one step of Lemma 9 to show that $\tilde{x}(\delta^-(v_i)) \leq 1$, which is forced by Lemma 11 for each (optimal) solution of LP_C, adding constraints of $LP_{\vec{T}_0}$ to LP_C cannot improve $v(LP_C)$ either.

To show that LP_F and LP_C are strictly stronger than the tree-based relaxations $LP_{T_0}, LP_{\vec{T}_0}$, and $LP_{\vec{T}}$, it is sufficient to give the following example.

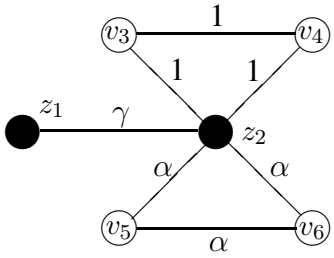


Figure 2.2: Example for $v(LP_{\vec{T}\_}) \ll v(LP_{\vec{T}}) \ll v(LP_F) = v(P_F)$.

Example 2 For the network G (or in the directed view $\vec{G}$) in Figure 2.2 set $\alpha \gg 1$ and $\gamma \gg \alpha$. Obviously, $v(P_F) = v(LP_F) = \gamma$. For $LP_{\vec{T}}$ is $\hat{x}$ with $\hat{x}_{23} = \hat{x}_{34} = \hat{x}_{42} = \frac{2}{3}$, $\hat{x}_{25} = \hat{x}_{56} = \hat{x}_{62} = \frac{1}{3}$, and $\hat{x}_{ij} = 0$ (otherwise) feasible, even optimal, and gives the value $v(LP_{\vec{T}}) = \alpha + 2$. Thus, there is

no positive lower bound for the ratio $\frac{v(LP_{\vec{T}})}{v(LP_F)}$.
With respect to $LP_{\vec{T}\_}$ and $LP_{\vec{T}}$, one observes that $(\dot{x}, \dot{t})$ with $\dot{t}_i = 0$ (for all $v_i \in V$), $\dot{x}_{23} = \dot{x}_{32} = \dot{x}_{34} = \dot{x}_{43} = \dot{x}_{24} = \dot{x}_{42} = \frac{1}{2}$, and $\dot{x}_{ij} = 0$ (otherwise) is an (optimal) solution for $LP_{\vec{T}\_}$ with the value 3. So, there is no positive lower bound for the ratio $\frac{v(LP_{\vec{T}\_})}{v(LP_{\vec{T}})}$.

2.6 Multiple Trees and the Relation to the Flow Model

In this section, we consider a relaxation based on multiple trees and prove its equivalence to an augmented flow relaxation. We also discuss some variants of the former relaxation.

2.6.1 Multiple Trees Formulation

In [KPH93], a variant of $P_{\vec{T}}$ was stated, using the idea that an undirected Steiner tree can be viewed as $|R|$ different Steiner arborescences with different roots.

$$
\begin{array}{|c|rcll}
\hline
P_{m\vec{T}} & c \cdot X & \rightarrow & \min, & & \\
 & X(\delta(v_i)) & \geq & 1 & (v_i \in R), & (9.1)\\
 & X(\delta(v_i)) & \geq & 2s_i & (v_i \in V \setminus R), & (9.2)\\
 & s_i & \geq & X_{ij} & (v_i \in V \setminus R\,;\ (v_i, v_j) \in \delta(v_i)\ ), & (9.3)\\
 & x^k_{ij} + x^k_{ji} & = & X_{ij} & (v_k \in R\,;\ (v_i, v_j) \in E), & (9.4)\\
 & x^k(\delta^-(v_i)) & = & \begin{cases} 1 & (v_k \in R\,;\ v_i \in R \setminus \{v_k\}), \\ 0 & (v_k \in R\,;\ v_i = v_k), \end{cases} & & (9.5)\\
 & x^k(\delta^-(v_i)) & = & s_i & (v_k \in R\,;\ v_i \in V \setminus R), & (9.6)\\
 & \{[v_i, v_j] \mid x^k_{ij} = 1\} & : & \text{contains no cycles} & (v_k \in R), & (9.7)\\
 & X & \in & \{0,1\}^{|E|}, & & (9.8)\\
 & x^k & \in & \{0,1\}^{|A|} & (v_k \in R), & (9.9)\\
 & s_i & \in & \{0,1\} & (v_i \in V \setminus R). & (9.10)\\
\hline
\end{array}
$$

In any feasible solution for $P_{m\vec{T}}$, each group of variables x^k describes an arborescence (with root z_k) spanning all terminals. The variables s describe the set of the other vertices used by these arborescences.

We will relate this formulation to the flow formulations. First, we have to present an improvement of LP_F.

2.6.2 Flow-Balance Constraints and an Augmented Flow Formulation

There is a group of constraints (see for example [KM98]) that can be used to make LP_F stronger. We call them flow-balance constraints:

$$x(\delta^-(v_i)) \leq x(\delta^+(v_i)) \qquad (v_i \in V \setminus R). \tag{10.1}$$

We denote the linear program that consists of LP_F and (10.1) by LP_{F+FB}. It is obvious that LP_{F+FB} is stronger than LP_F. The following example shows that it is even strictly stronger.

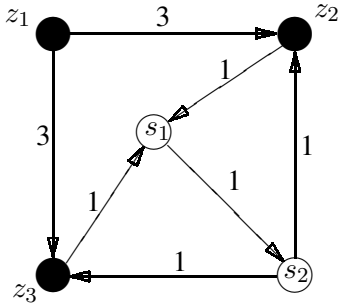


Figure 2.3: Example with $v(LP_F) < v(LP_{F+FB}) = v(P_{F+FB})$.

Example 3 The network $\vec{G}$ in Figure 2.3 with z_1 as the root and $R^{z_1} = \{z_2, z_3\}$ gives an example for $v(LP_F) < v(LP_{F+FB})$: $v(P_{F+FB}) = v(LP_{F+FB}) = 6,\ v(LP_F) = 5\frac{1}{2}$.

Now consider the following formulation:

$\boxed{P_{F'+FB}}$

$$\begin{aligned} c \cdot X &\rightarrow \min, \\ x_{ij} + x_{ji} &= X_{ij} \qquad ((v_i, v_j) \in E), \qquad (11.1) \\ (x, y) &: \text{ is feasible for } P_{F+FB}. \qquad (11.2) \end{aligned}$$

Lemma 14 If (X, x, y) is an (optimal) solution for $LP_{F'+FB}$ with root terminal z_a, then there exists an (optimal) solution $(X, \check{x}, \check{y})$ for $LP_{F'+FB}$ for any other root terminal $z_b \in R\backslash\{z_a\}$.

Proof: One can verify that $(X, \check{x}, \check{y})$ with $\check{x}_{ij} := x_{ij} + y^b_{ji} - y^b_{ij},\ \check{y}^t_{ij} := \max\{0, y^t_{ij} - y^b_{ij}\} + \max\{0, y^b_{ji} - y^t_{ji}\},\ \check{y}^a_{ij} := y^b_{ji}$ (for all $[v_i, v_j] \in A,\ z_t \in R\backslash\{z_a, z_b\}$) satisfies (11.1) and (3.2). Because of $\sum_{[v_j,v_i]\in\delta^-(v_i)}(\check{y}^t_{ji} - \check{y}^t_{ij}) = \sum_{[v_j,v_i]\in\delta^-(v_i)}(\max\{0, y^t_{ji} - y^b_{ji}\} + \max\{0, y^b_{ij} - y^t_{ij}\} + \min\{0, -y^t_{ij} + y^b_{ij}\} + \min\{0, -y^b_{ji} + y^t_{ji}\}) = \sum_{[v_j,v_i]\in\delta^-(v_i)} y^t_{ji} - y^b_{ji} + y^b_{ij} - y^t_{ij}$ (for all $v_i \in V,\ z_t \in R\backslash\{z_a, z_b\}$) the constraints (3.1) are satisfied, too. From (3.1) for y^b, it follows that $x(\delta^-(v_i)) = \check{x}(\delta^-(v_i))$ and $x(\delta^+(v_i)) = \check{x}(\delta^+(v_i))$ for all $v_i \in V\backslash R$; therefore $\check{x}$ satisfies the flow-balance constraints (10.1). Because this translation could also be performed from any (optimal) solution with root terminal z_b to a feasible solution with root terminal z_a, the value $v(LP_{F'+FB})$ is independent of the choice of the root terminal and $(X, \check{x}, \check{y})$ is an (optimal) solution. □

It follows immediately that $LP_{F'+FB}$ is equivalent to LP_{F+FB}.

2.6.3 Relationship between the two Models

We will now show that the linear relaxation $LP_{m\vec{T}}$ (where (9.7) is replaced by linear constraints of the form (8.3)) is equivalent to LP_{F+FB}.

Lemma 15 $v(LP_{m\vec{T}}) = v(LP_{F'+FB})$.

Proof:
I) $v(LP_{m\vec{T}}) \geq v(LP_{F'+FB})$: Let $(\hat{X}, \hat{x}, \hat{s})$ denote an (optimal) solution for $LP_{m\vec{T}}$. Define x with $x := \hat{x}^1$, and y with $y^t := \max\{\hat{x}^1 - \hat{x}^t, 0\}$ (for all $z_t \in R^{z_1}$). Because of (9.4) and the definition of y, $y^t_{ij} = 0$ if $y^t_{ji} > 0$ (for all $[v_i, v_j] \in A$ and $z_t \in R^{z_1}$).

For all $z_t \in R^{z_1}, v_i \in V\backslash\{z_1, z_t\}$ it holds that:

$$\begin{aligned} y^t(\delta^-(v_i)) - y^t(\delta^+(v_i)) &= (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} > \hat{x}^t_{ji}\}) - \\ &\quad (\hat{x}^1 - \hat{x}^t)(\{[v_i, v_j] \in \delta^+(v_i) | \hat{x}^1_{ij} > \hat{x}^t_{ij}\}) \\ &= (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} > \hat{x}^t_{ji}\}) + \\ &\quad (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} < \hat{x}^t_{ji}\}) \quad \text{(because of (9.4))} \\ &= (\hat{x}^1 - \hat{x}^t)(\delta^-(v_i)) \qquad = \qquad 0 \qquad \text{(because of (9.5) or (9.6)).} \end{aligned}$$

With the same argumentation adapted to $v_i = z_t$, it follows that y satisfies (3.1).
The other constraints (3.2) are satisfied in a trivial way. A substitution of (9.4) and (9.6) into (9.2) gives the flow-balance constraints (10.1). Thus, (X, x, y) is feasible for $LP_{F'+FB}$.
II) $v(LP_{m\vec{T}}) \leq v(LP_{F'+FB})$: Let (X, x, y) denote an (optimal) solution for $LP_{F'+FB}$. From lemma 14 we know that there is an (optimal) solution $(X, \hat{x}^r, \hat{y}^r)$ for each choice of the root vertex $z_r \in R$, with the property that $\hat{s}_i := \hat{x}^t(\delta^-(v_i))$ (for any $v_i \in V\backslash R$) has the same value for any choice of $z_t \in R$. With the argumentation of Theorem 13 it follows that $(X, \hat{x}, \hat{s})$ is feasible for $LP_{m\vec{T}}$. □

Corollary 15.1 The constraints (9.1), (9.3), and (9.7) are useless with respect to the value of the linear relaxation $LP_{m\vec{T}}$.

Corollary 15.2 The linear program $LP_{m\vec{T}-}$(with the same objective function as $LP_{m\vec{T}}$) that contains only the equations (9.4), (9.5), and (9.6) is equivalent to LP_F.

2.7 A Collection Of New Formulations

In [PV01a] we introduced the common flow formulation LP_{F^2}, the strongest linear relaxation of polynomial size known so far. Concerning this relaxation we have frequently been asked three questions: How we developed the new relaxation, whether the result can be strengthened (the notation LP_{F^2} looks as if there could be an LP_{F^3}) and how a relaxation of this kind could be used in a practical algorithm. We will address all three issues in the following.

2.7.1 Properties of the Flow/Cut Relaxations

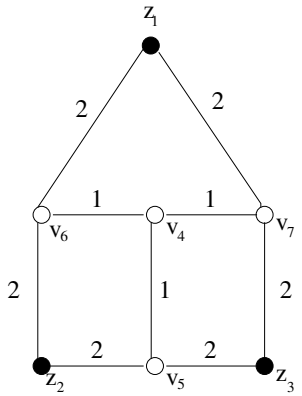


Figure 2.4: Example with $v(LP_C) < v(P_C)$.

A network with a deviation between the integral and the linear solution is shown in Figure 2.4. Choosing z_1 as root yields the value 7.5 as $v(LP_F)$ (setting all x-variables in the direction away from z_1 to 0.5 leads to an optimal solution), while an optimal Steiner tree (and $v(P_F)$) has value 8. Thus the integrality gap is at least $\frac{8}{7.5}$.

Goemans [Goe98] extends the example in Figure 2.4 and constructs networks G_k whose integrality gap comes arbitraily close to $\frac{8}{7}$. The network G_k consists essentially of $\binom{k}{2}$ copies of the network in Figure 2.4. It has $k+1$ terminals $a_0, a_1, ..., a_k$, and k^2 non-terminals $b_1, ..., b_k, c_{12}, ..., c_{ij}, ..., c_{k-1,k}, d_{12}, ..., d_{ij}, ..., d_{k-1,k}$. For any i and j, $1 \leq i < j \leq k$ one includes the network of Figure 2.4 with the following labeling: z_1 is a_0, z_2 is a_i, z_3 is a_j, v_4 and v_5 are c_{ij} and d_{ij}, and v_6 and v_7 are b_i and b_j. An optimal Steiner tree has value $4k$ and the optimal LP_C solution is obtained by setting all x-variables in the direction away from a_0 to $\frac{1}{k}$, yielding the value $\frac{1}{k}(4k + 7\binom{k}{2}) = 3.5k + 0.5$. Thus, the integrality gap approaches $\frac{8}{7}$ with increasing k. This is the largest integrality gap known[2].

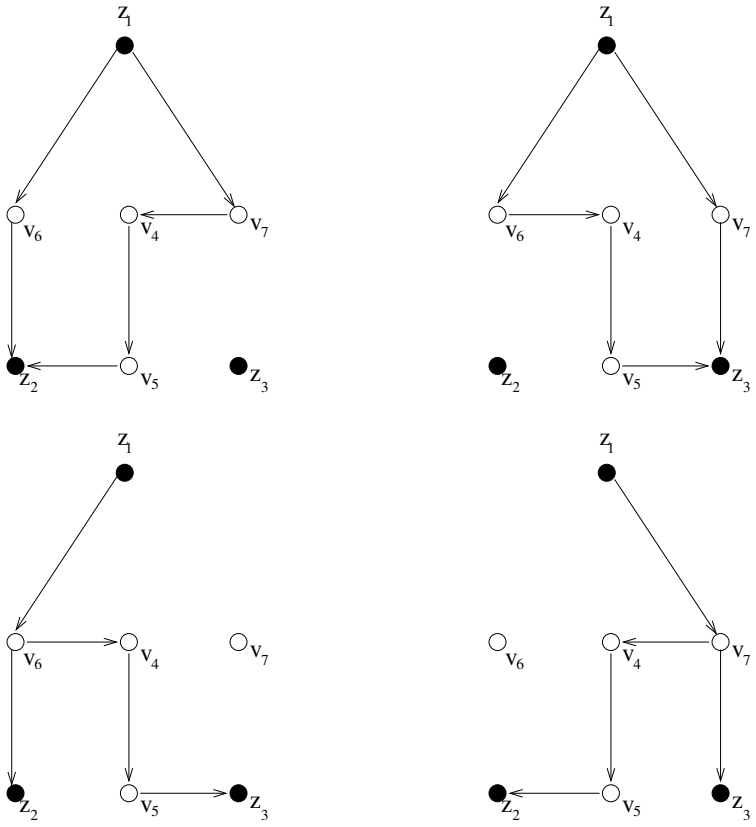


Figure 2.5: y-variables and decomposition into trees.

Figure 2.5 shows how the y-variables of the LP_F solution for Figure 2.4 can be viewed as a linear combination of incidence vectors of paths in some trees. The upper-left network shows the y^2 variables, each arc $[v_i, v_j]$ corresponds to $y^2_{ij} = 0.5$. The upper-right figure shows the same for y^3. The y values can be decomposed into two rooted trees T_1 and T_2 that are depicted in the two figures at the bottom. For any $z_t \in R^{z_1}$, y^t is a linear combination of the incidence vectors of the paths from

[2]Independently, we constructed other examples with the same asymptotic integrality gap. For example, the edges between a_i and b_i can be deleted for every i without deteriorating the asymptotic result.

z_1 to z_t in T_1 and T_2 (in this example both incidence vectors have weight 0.5).

If the x-variables were a linear combination of the incidence vectors of these trees, the optimal value of the linear relaxation would be the same as of the integral formulation. In this example, we see that the arc $[v_4, v_5]$ causes the problem: The linear combination of the incidence vectors of the trees yields 1 for $[v_4, v_5]$, while x_{45} has the value 0.5. Looking at the flow-variables y^2 and y^3 one can see that at vertex v_4 flows of the different commodities enter from different arcs ($y^2_{74} = y^3_{64} = 0.5$), but depart together on arc $[v_4, v_5]$. We denote this situation as a rejoining. Formally, a rejoining of flows of the commodities $\{z_{i_1}, z_{i_2}, \ldots, z_{i_k}\} =: B$ at vertex $v_i \in V$ is defined by the condition $\sum_{e \in \delta^-(v_i)} \max\{y^t_e | z_t \in B\} > \sum_{e \in \delta^+(v_i)} \max\{y^t_e | z_t \in B\}$. In Section 2.7.3 we will show how to attack rejoining of flow in an extended linear program.

This situation is typical. In fact, we do not know of any example for a network with an integrality gap for LP_C that does not include a variant of Figure 2.4. Of course, there may be additional edges, edges replaced by paths, other edge weights and non-terminals replaced by terminals, but still Figure 2.4 is a minor of the network (a minor is a subgraph obtained after a sequence of contractions and deletions, for the relation of minors to linear relaxations for the Steiner problem see [CR94a]). Furthermore, the rejoining of flows of different commodities is a common property of all examples known to us, although more than two flows can rejoin (as we will show in Figure 2.6) and it may need a change of the root to see a rejoining. Figure 2.4 can be used to illustrate the latter: Turn v_4 into a terminal z_4 and choose z_4 as root, the optimal linear solution value stays the same (setting all x-variables in the direction away from z_4 to 0.5 leads to an optimal solution). Again, the y-variables can be decomposed into the same trees T_1 and T_2 (just with a reorientation of the arcs), but now there is no rejoining of flows.

2.7.2 Common Flow

As we have seen in the last section, the typical problematic situation for LP_F can be described as the rejoining of flows of different commodities. We capture this condition in linear constraints with the help of additional variables y^B (for all subsets $B \subseteq R^{z_1}$) that describe the amount of flow going from the root to any terminal $z_t \in B$: $y^B \geq y^t$, for all $z_t \in B$. From Figure 2.5 it is easy to see the flow to z_2 and z_3 ($y^{\{z_2,z_3\}}$) incoming at v_4 is greater than the flow outgoing of v_4. This cannot be the case if the x-variables correspond to a feasible Steiner tree. Thus, we can introduce additional constraints that assure that for each common flow the outgoing flow at any vertex has at least the value of the incoming flow.[3]

As already noted in Section 2.7.1, the same network can or cannot have rejoining of flow, depending on the choice of the root. To detect all rejoining we have to consider all possible roots. Thus, the variables that will be used to describe the common flow formulation will have the shape $y^{r,B}$ and they will describe a flow from z_r to all terminals $z_t \in B \subseteq R^{\{z_t\}}$ (we will skip the chosen root in the notation if the root is clear from the context).

2.7.3 A Collection of Common Flow Formulations

We will first describe the full common flow formulation that has an exponential number of constraints and variables, and describe afterwards how to reduce it to different formulations (e.g., of only polynomial size)

[3] In [PV01a] we used a semantically inverse, but mathematically equivalent definition of the common flow. Meanwhile, we find the new definition more intuitive and compact.

$$
\begin{aligned}
\boxed{P_{F^R}} \qquad c \cdot X &\to \min, \\
y^{r,\{z_t\}}(\delta^-(z_r)) &= y^{r,\{z_t\}}(\delta^+(z_r)) - 1 \quad (z_r \in R; z_t \in R^{z_r}), && (12.1)\\
y^{r,\{z_t\}} - y^{r,\{z_s\}} &\leq y^{s,\{z_t\}} \quad (\{z_r, z_s, z_t\} \in R), && (12.2)\\
y^{r,\{z_s\}}_{ji} - y^{r,\{z_t\}}_{ji} &\leq y^{s,\{z_t\}}_{ij} \quad (\{z_r, z_s, z_t\} \in R; [i,j] \in A), && (12.3)\\
y^{r,B}(\delta^-(v_i)) &\leq y^{r,B}(\delta^+(v_i)) \quad (z_r \in R; B \subseteq R^{z_r}; v_i \in V \setminus (B \cup \{z_r\}), && (12.4)\\
y^{r,B} &\leq y^{r,C} \quad (z_r \in R; B \subset C \subseteq R^{z_r}), && (12.5)\\
y^{r,R^{z_r}}_{ij} + y^{r,R^{z_r}}_{ji} &\leq X_{ij} \quad (z_r \in R; (i,j) \in E), && (12.6)\\
y^{r,\{z_r\}} &= 0 \quad (z_r \in R), && (12.7)\\
y &\geq 0, && (12.8)\\
X &\in \{0,1\}^{|E|}. && (12.9)
\end{aligned}
$$

It follows from the proof of Lemma 14 that the constraints (12.2) and (12.3) redirect the flows from root z_r to flows from any other root $z_s \in R$ to each terminal $z_t \in R$. In particular, from these constraints in combination with (12.7) it follows that $y^{t,\{z_r\}}_{ji} = y^{r,\{z_t\}}_{ij}$ for all $z_r, z_t \in R$. For each flow from a root z_r to a terminal z_t, described by the variables $y^{r,\{z_t\}}$, the constraints (12.1) guarantee an outflow of 1 out of z_r, and (together with the previous observation) an inflow of 1 in z_t. Together with the constraints (12.4) (with $B = \{z_t\}$) and (12.8) this guarantees a flow of one unit from z_r to each terminal $z_t \in R^{z_r}$.
The constraints (12.4) also guarantee for any root z_r and set of terminals B that there is no rejoining at any vertex v_i (to be more precise, there is a penalty for rejoined flows in form of increased $y^{r,B}$ values). The constraints (12.5) set each common flow from a root z_r to a set of terminal C to at least the maximum of all flows from z_r to a subset $B \subset C$ of the terminals.
Finally the constraints (12.6) build the edgewise maximum over all flows from all terminals in the edge variables X.

We do not know of any network where this relaxation has an integrality gap. Unfortunately, we are neither able to prove that there is no integrality gap, nor to do a really broad experimental study to support this (see Section 2.9.3). Note that due to the exponential number of constraints *and* variables the possibility that there is no integrality gap is not ruled out by complexity arguments.

2.7.4 Polynomial Variants

We will now show how to derive a polynomial-size relaxation from LP_{F^R}: We simply limit the terminal sets B to a polynomial size, e.g., by choosing two constants $k_1 \geq 1$ and $k_2 \geq 1$ ($k_1 + k_2 < |R|$), and using only those variables $y^{r,B}$ with $|B|$ either at most k_1 or at least $|R| - k_2$. We denote the remaining relaxation with $LP_{F^{k_1,k_2}}$. It has $O(r^{1+\max\{k_1,k_2-1\}}a)$ variables and $O(r^{2\max\{k_1,k_2-1\}}a)$ constraints. For example, if we choose $k_1 = 2$ and $k_2 = 1$ and also use only one fixed root z_r we get a relaxation that is equivalent to LP_{F^2} presented in [PV01a]. Here, we state the relaxation in the form as it is derives from P_{F^R}, using the variables $y^{r,B}$ for a fixed z_r and for $B \in \mathcal{B} = \{C \subseteq R^{z_r} \mid |C| \in \{1, 2, |R| - 1\}\}$

$$
\begin{aligned}
\boxed{P_{F^2}} \qquad c \cdot x &\to \min, \\
y^{r,\{z_t\}}(\delta^-(z_r)) &= y^{r,\{z_t\}}(\delta^+(z_r)) - 1 \quad (z_t \in R^{z_r}), && (13.1)\\
y^{r,B}(\delta^-(v_i)) &\leq y^{r,B}(\delta^+(v_i)) \quad (B \in \mathcal{B}; v_i \in V \setminus (B \cup \{z_r\}), && (13.2)\\
y^{r,B} &\leq y^{r,C} \quad (B \subset C \in \mathcal{B}), && (13.3)
\end{aligned}
$$

$$y^{r,R^{zr}} = x, \tag{13.4}$$
$$y \geq 0, \tag{13.5}$$
$$x \in \{0,1\}^{|A|}. \tag{13.6}$$

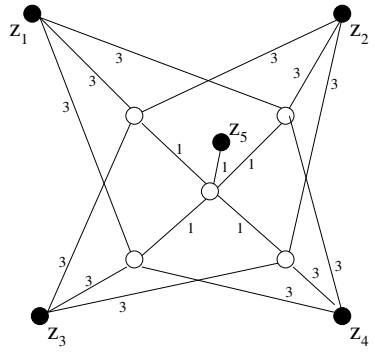


Figure 2.6: $v(LP_F) = 14\frac{1}{3} < v(LP_{F^{2,1}}) = 14\frac{2}{3} < v(LP_{F^R}) = 15 = v(P_{F^R})$.

The drawback of this limitation is that not all rejoinings will be detected by the limited relaxation, as the following example shows: In Figure 2.6, using LP_F with z_1 as root, there is a rejoining of 3 different flows (y^2, y^3 and y^4) at the central vertex. As LP_{F^2} captures only rejoining of 2 flows, there will still be some integrality gap. The figure also shows how k_2 weakens the relaxation: Only if the common flow $y^{r,B}$ to a set of terminals B is considered where z_2, z_3 and z_4 are in B, but z_5 is not in B, the relaxation gives the optimal integer value. Thus, for $k_1 \geq 3$ or $k_2 \geq 2$ there is no integrality gap.

Note that the example in Figure 2.6 can be generalized to a network $G[k_1, k_2]$, which produces an integrality gap for any common flow relaxation that is limited by some constants k_1 and k_2: Around a center vertex v_0 put $k_1 + 2$ edges $[v_0, v_1], [v_0, v_2], \ldots, [v_0, v_{k_1+2}]$ of cost 1. Additionally use $k_1 + 2$ terminals $(a_1, a_2, \ldots, a_{k_1+2})$. Each of the terminals $z_i \in R$ is connected to all vertices $v_j, j \neq 0, j \neq i$ with edges of cost k_1+1. Then connect the center vertex to k_2 different terminals $(b_1, b_2, \ldots, b_{k_2})$ with edges of cost 1. An integer optimal solution has the cost $k_2 + (k_1 + 1)(k_1 + 2) + 2$, but the following X-variables are feasible for the k_1, k_2 limited common flow relaxation: Set X-variables of edges connected to $b_i, 1 \leq i \leq k_2$ to 1, the X-variable for (v_0, v_1) to $1-(k_1+1)^{-1}$ and all other X variables to $(k_1+1)^{-1}$. Thus, the optimal value of the relaxation is at most $k_2+(k_1+1)(k_1+2)+2-(k_1+1)^{-1}$. Figure 2.6 shows the network $G[2, 1]$.

As a consequence, we have derived a collection of polynomial relaxations $LP_{F^{k_1,k_2}}$, where $LP_{F^{k_1,k_2}}$ can be related to $LP_{F^{j_1,j_2}}$. We assume that $k_1 + k_2 \leq j_1 + j_2$.

$k_1 \leq j_1, k_2 \leq j_2$**:** The relaxation $LP_{F^{j_1,j_2}}$ is stronger than $LP_{F^{k_1,k_2}}$ (since it contains a superset of the constraints and variables). If additionally $k_1 < j_1$ or $k_2 < j_2$, the relaxation $LP_{F^{j_1,j_2}}$ is even strictly stronger. Consider the network $G[\min\{k_1, j_1\}, \min\{k_2, j_2\}]$, $LP_{F^{j_1,j_2}}$ gives the optimal solution, while $LP_{F^{k_1,k_2}}$ has an integrality gap.

otherwise: The relaxations are incomparable. To see this, consider the networks $G[\min\{k_1, j_1\}, \max\{k_2, j_2\}]$ and $G[\max\{k_1, j_1\}, \min\{k_2, j_2\}]$.

Now, we show why considering all roots is important. The only difference between the relaxations $LP_{F^{2,1}}$ and LP_{F^2} is that $LP_{F^{2,1}}$ considers all terminals as root.

Lemma 16 The relaxation $LP_{F^{2,1}}$ is strictly stronger than LP_{F^2}.

Proof: The relaxation $LP_{F^{2,1}}$ is stronger than LP_{F^2}, as it contains a superset of the constraints and variables. The network $G[1,1]$ can be used to show that $LP_{F^{2,1}}$ is even strictly stronger than LP_{F^2}. Here, $LP_{F^{2,1}}$ gives the optimal value, but for LP_{F^2} it depends on the choice of the root. Using any of the vertices a_1, a_2, or a_3 as root gives the integer optimal value, while with b_1 as root there is an integrality gap. □

2.7.5 Restricted Version

We tested different restricted versions of the common flow relaxation, but did not come to a definitive conclusion which approach is the best. Nevertheless, we can present a very restricted, but still very useful version.

We observed that in many cases the tracing of the common flow to all $|R|-1$ terminals ($k_2 = 1$) catches most of the problematic situations except for insightfully constructed pathological instances. As depicted in Figure 2.6, to fool the relaxation with $k_2 = 1$, there typically has to be a terminal directly connected to the vertex at which the rejoining happens. In most of the cases where this happens, the terminal is the vertex itself. To attack those situations without the need to introduce a quadratic number of constraints or variables, we restrict the common flow relaxation by $k_1 = k_2 = 1$ on non-terminals, and to $k_2 = 2$ on terminals, but only considering the set $R \setminus \{z_r, z_i\}$ at a terminal z_i (with respect to root z_r). Additionally, we restrict the relaxation to one fixed root z_r, and because we do not need the X variables for different roots anymore, we can use the $y^{R^{z_r}}$ as x variables.

In a branch-cut-price framework it is highly desirable to introduce as few variables as possible. Working with flow-based relaxations makes this difficult, because if a flow variable on one edge is necessary, all variables and all flow conservation constraints have to be inserted to get something useful. Therefore the cut-based relaxations are more suitable in this context. In analogy to P_C, we can use constraints of the form $y_i^{\{z\}}(\delta^-(W)) \geq 1$ for all $W \cap \{z_r, z_i\} = z_i$.

Finally, we eliminate all variables $y^{\{z_i\}}$ and $y_e^{R\setminus\{z_r,z_i\}}$ if e is not in $\delta^-(z_i)$ in the following way: Unwanted $y^{\{z_i\}}$ variables are replaced by $y^{R\setminus\{z_r,z_i\}}$ if the replacement is possible using the constraints (12.5). Similarly, unwanted $y^{R\setminus\{z_r,z_i\}}$ variables are replaced by x if possible. All constraints that still use unwanted variables are deleted.

Now, we have the following formulation:

$$
\begin{aligned}
\boxed{P_{C'}} \quad c \cdot x &\rightarrow \min, \\
x(\delta^-(W)) &\geq 1 \quad (z_1 \notin W, R \cap W \neq \emptyset), && (14.1)\\
y^{R\setminus\{z_r,z_i\}}(\delta^-(W) \cap \delta^-(z_i)) + x(\delta^-(W) \setminus \delta^-(z_i)) &\geq 1 \quad (z_r \notin W, R^{z_i} \cap W \neq \emptyset), && (14.2)\\
x(\delta^-(v_i)) &\leq x(\delta^+(v_i)) \quad (v_i \in V \setminus R), && (14.3)\\
y^{R\setminus\{z_r,z_i\}}(\delta^-(z_i)) &\leq x(\delta^+(z_i)) \quad (z_i \in R^{z_r}), && (14.4)\\
y &\geq 0, && (14.5)\\
x &\in \{0,1\}^{|A|}. && (14.6)
\end{aligned}
$$

This new formulation has fewer than $2|A|$ variables. Although it has an exponential number of constraints, solving the linear relaxation is relatively easy. This will be described in Section 2.9.3.

2.7.6 Relation to Other Relaxations

Lemma 17 $v(LP_{F^2}) \geq v(LP_{2T})$.

Proof: Let (x, y) be an (optimal) solution of LP_{F^2}. For all $\{z_k, z_l\} \subseteq R^{z_r}$ and $k < l$ define $\grave{y}^{kl} := y^{r,\{z_k\}} - y^{r,\{z_k,z_l\}}$, $\acute{y}^{kl} := y^{r,\{z_l\}} - y^{r,\{z_k,z_l\}}$, and $\check{y}^{kl} := y^{r,\{z_k,z_l\}}$. The variables $(x, \check{y}, \grave{y}, \acute{y})$ satisfy the constraints of LP_{2T}. □

Because LP_{F^2} contains the flow-balance constraints and is stronger than LP_{2T}, it is stronger than LP_{2T+FB} (constructed by adding (10.1) to LP_{2T}). It follows directly that LP_{F^2} is also stronger than LP_{F+FB}. The following example shows that it is even strictly stronger than the other stated relaxations.

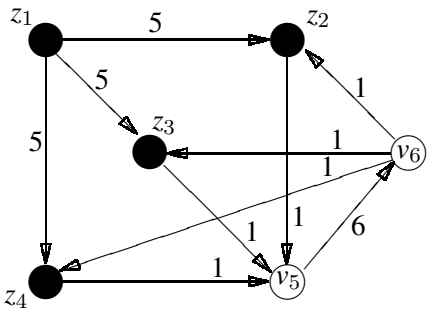


Figure 2.7: Example with $v(LP_{2T}) < v(LP_{F+FB}) = v(LP_{F^2}) = v(P_{F^2})$.

Example 4 Here is an example for $v(LP_{2T}) < v(LP_{F^2}) = v(P_{F^2})$: Setting all x-variables to 0.5 leads to a feasible (and optimal) solution for LP_{2T} with the value 13.5. An optimal solution for LP_{F^2} is $x_{13} = x_{35} = x_{56} = x_{62} = x_{64} = 1$, which forms a Steiner tree with value 14. Notice that this is also an example with $v(LP_{2T}) < v(LP_{F+FB})$. On the other hand, if v_5 is moved to R, $v(LP_{F+FB}) = v(LP_F) = 12 < v(LP_{2T}) = v(LP_{2T+FB}) = 13.5 < v(LP_{F^2}) = v(P_{F^2})$ (The optimal value for LP_{F+FB} is reached by $\hat{x}$ with $\hat{x}_{12} = \hat{x}_{13} = \hat{x}_{14} = \hat{x}_{25} = \hat{x}_{35} = \hat{x}_{45} = 1/3, \hat{x}_{56} = \hat{x}_{62} = \hat{x}_{63} = \hat{x}_{64} = 2/3$). Thus, LP_{F+FB} and LP_{2T} are incomparable.
This example has been chosen because it is especially instructive. For $v(LP_{2T+FB}) < v(LP_{F^2})$, as for all other statements in this work that one relaxation is *strictly* stronger than another, we also know (originally) undirected instances as examples.

The relaxation LP_{F^2}, as well as LP_{2T}, makes it difficult for flows to two different terminals to split up and rejoin by increasing the x-variables on arcs with rejoined flow. One could say that rejoining has to be "payed" for. To get an intuitive impression why LP_{F^2} is strictly stronger than LP_{2T} (or even LP_{2T+FB}), notice that in LP_{F^2}, there is one flow to each terminal and rejoining of each pair of these flows has to be payed for; while in LP_{2T}, it is just required that for each pair of terminals there are two flows and rejoining them has to be payed for. The latter task is easier; for example it is possible (for given x-values) that for each pair of terminals there are two flows that do not rejoin, but there are not $|R^{z_1}|$ flows to all terminals in R^{z_1} that do not rejoin pairwise; this is the case in example 2.7 (setting all x-variables to 0.5).

Still, we have to prove two more relations to relate the common flow relaxations to the other relaxations.

Lemma 18 The relaxation LP_{F+FB} is equivalent to $LP_{F^{1,1}}$.

Proof: The relaxations are nearly the same. The only difference is that $LP_{F^{1,1}}$ considers all possible roots. But as we have seen in Lemma 14, the choice of the root does not change the value of the

solution and the transformation of a solution for one root into a solution for another is covered exactly by the constraints (12.2) and (12.3). □

Lemma 19 The relaxation LP_{F^2} is strictly stronger than $LP_{C'}$ and $LP_{C'}$ is strictly stronger than $LP_{F^{1,1}}$.

Proof: It is obvious that LP_{F^2} is stronger than $LP_{C'}$, as $LP_{C'}$ is a restricted and aggregated version of LP_{F^2}. Similarly, $LP_{C'}$ contains a superset of variables and constraints of LP_{C+FB}, which is equivalent to $LP_{F^{1,1}}$. To see that the relations are also strictly stronger, consider the networks $G[1,1]$. If we choose a_1, a_2, or a_3 as root, $LP_{C'}$ has an integrality gap, LP_{F^2} not. And if we contract the edge (v_0, b_1), $LP_{C'}$ gives the integer optimal value, while $LP_{F^{1,1}}$ still has an integrality gap. □

2.8 A Hierarchy of Relaxations

2.8.1 Summary of the Relations

The following Figure 2.8 summarizes the relations stated before. All relaxations in the same box are equivalent. A line between two boxes means that the relaxations in the upper box are strictly stronger than those in the lower box. Notice that the "strictly stronger" relation is transitive. For an overview of the meaning of the abbreviations, see Table 2.1 on page 22.

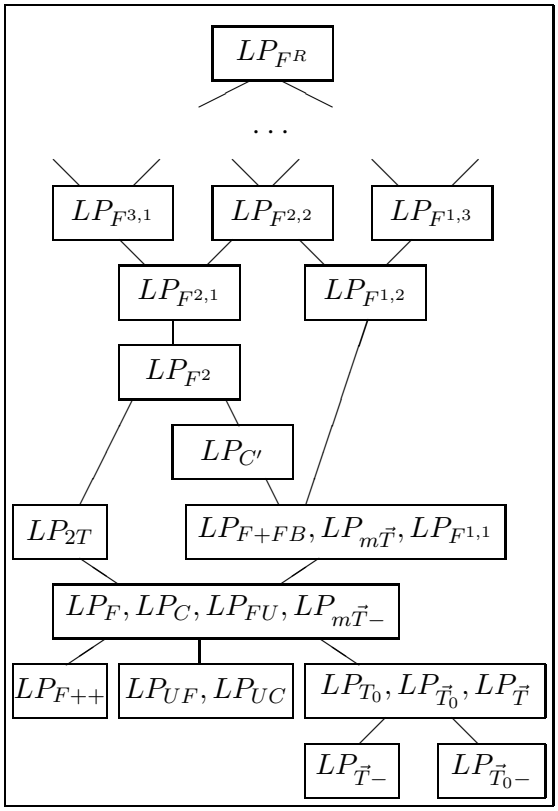


Figure 2.8: Hierarchy of relaxations.

2.8.2 Extensions to Polyhedral Results

It should be mentioned that some of the stated results on the relationship between the optimal values of linear relaxations extend directly to polyhedral results concerning the corresponding feasible sets. This is always the case if optimality is not used in the proofs (e.g., in Lemmas 7 or 8); and hence the feasible set of one relaxation (projected into the x-space) is mapped into the corresponding set of some other. The situation is different in the other cases (e.g., the proofs of Lemma 9 or Theorem 13). Here the assumption of optimality of x can obviously be replaced by the assumption of minimality of x (a feasible x is minimal if there is no feasible $x' \neq x$ with $x' \leq x$). In such cases, the presented results extend directly to polyhedral results in the sense of inclusions between the dominants of the corresponding polyhedra (projected into the x-space). (The dominant of Q is $\{x' \mid x' \geq x \in Q\}$.)

Note also that polyhedral results concerning the facets of the Steiner tree polyhedron (like those in [CR94a, CR94b]) fall into a different category. Our line of approach has been studying linear relaxations of general, explicitly given (and frequently used) integer formulations; not methods for describing facet defining inequalities. Applying such descriptions is typically possible only if the graph has certain properties (e.g., that it contains a special substructure) and involves separation problems that are believed to be difficult.

2.9 Using Relaxations

To actually exploit the relaxations for computing lower bounds, two factors are of more or less equal importance: How good the optimal values of the corresponding linear programs are, and how fast these values can be determined or sufficiently approximated. In the following, we investigate both questions for the stated relaxations.

2.9.1 The Spanning Tree Formulation and Lagrangian Relaxation

A Lagrangian relaxation LaP_{T_0} of the tree formulation P_{T_0} is described in [Bea89], relaxing the degree constraints (6.2). After this, a subgradient optimization of the Lagrangian multiplier problem can be used, which involves calculating a minimum spanning tree in each iteration. Using this approach, the value $v(LP_{T_0})$ can be approximated fairly fast (this relaxation has the integrality property). The problem here is the value $v(LP_{T_0})$ itself. Theorem 13 already indicates theoretically that LP_{T_0} is not a generally tight relaxation. Empirically, we observed that usually the bound $v(LP_{T_0})$ is only satisfactory for instances where the average distance between terminals is not too high in comparison to the average edge length (e.g., random networks with many terminals). A bad situation for this relaxation typically arises from instances modeling points in the plane with respect to a given metric. For instances with Euclidean distances or grid instances with few terminals, gaps of more than 50% are not exceptional. Nevertheless, we have further investigated the mentioned Lagrangian relaxation, since it can be useful for some instances.

We obtained a minor improvement in the speed of the subgradient optimization by applying a sensitivity analysis for the Lagrangian multipliers. Using data structures for efficient handling of tree bottlenecks and alternative chords (see [Tar79, VJ83]) allows fast calculation of the quantities by which each multiplier can be changed without affecting the validity of the calculated minimum spanning tree. Modifying the multipliers by these quantities improves the lower bound immediately.

In [DV87], some modifications for this relaxation are suggested, for example adding (and relaxing) further constraints and using another structure for G_0. In our experiments, these modifications did not

improve the overall results of the lower bound calculation: In situations where LaP_{T_0} leads to a substantial gap, no decisive improvements could be achieved using these modifications.

In [BL98], a relaxation constructed by adding the Steiner cut (and some other) constraints to LP_{T_0} is used. This indeed leads to a stronger relaxation than LP_{T_0}. However, as we have proved in Corollary 13.1 in Section 2.5, LP_C cannot be strengthened (i.e., $v(LP_C)$ does not change) by adding constraints like those present in LP_{T_0}; this motivates concentrating on LP_C itself.

2.9.2 The Cut Formulation, Dual Ascent and Row Generating

Considering the relaxation LP_C, the situation is to some degree the converse of the case LP_{T_0}. It is known that $v(LP_C)$ does not deviate from the optimum by more than 50% [GB93]. All our experimental investigations strongly suggest that LP_F is a very tight relaxation. As an example, for all D-instances of the OR-Library $v(LP_C)$ is equal to $v(P_C)$. Even for the instances where there is a gap, the knowledge of a solution of LP_C has been usually sufficient to solve the instance exactly (without branching) through bound-based reduction techniques (Section 3.4). So, the really interesting problem is how to calculate (or sufficiently approximate) a solution for LP_C.

The direct approach of solving the complete linear program using a standard LP-solver is not practical, even for the equivalent multicommodity flow relaxation [Won84], which has approximately ra variables and $r(a+n)$ constraints: This is still too much for moderate and large instances; and the resulting linear programs are often highly degenerated.

We present those two approaches that have been proven to be very effective, dual ascent and a row generating approach.

Dual Ascent: A fast dual ascent algorithm that generally provides fairly good lower bounds was described in [Won84] for the equivalent multicommodity flow relaxation. Below, we give an alternative description of it as a dual ascent algorithm for LP_C, which we call **DUAL-ASCENT**. First, we state the dual linear program DLP_C, that uses dual variables u_W for each Steiner cut W ($z_1 \in \overline{W}$ and $R^{z_1} \cap W \neq \emptyset$):

$$\boxed{DLP_C} \qquad \sum u_W \;\rightarrow\; \max,$$

$$\sum_{W, a \in \delta^-(W)} u_W \;\leq\; c(a) \quad (a \in A), \tag{15.1}$$

$$u \;\geq\; 0. \tag{15.2}$$

Now, we can describe the algorithm:

- Initialize the reduced costs ($\tilde{c} := c$), the lower bound ($lower := 0$) and assume all dual variables u have been set to zero.
- In each iteration, choose a terminal $z_t \in R^{z_1}$ not reachable from the root by edges of zero reduced cost. Let W, $z_t \in W$, be the smallest set such that $\{\overline{W}, W\}$ is a Steiner cut and $\tilde{c}_{ij} > 0$ for all $[v_i, v_j] \in \delta^-(W)$. Set the dual variable u_W to $\Delta := \min\{\tilde{c}_{ij} \mid [v_i, v_j] \in \delta^-(W)\}$ and let $lower := lower + \Delta$ and $\tilde{c}_{ij} := \tilde{c}_{ij} - \Delta$ (for all $[v_i, v_j] \in \delta^-(W)$).
- Repeat until no such terminal is left.

A good implementation of this algorithm has running time $O(a \cdot \min\{a, rn\})$ (see for example [Dui93]). Empirically, it is usually faster than this time bound would suggest.

The algorithm DUAL-ASCENT already achieves very good results. Out of the 20 D-instances of the OR-Library, a DUAL-ASCENT run yields the optimum (i.e., $v(P_C)$) for 12 instances. The average gap between lower bound and optimum is 0.4% and the average running time is 0.02 seconds.

A critical point in this algorithm is the choice of z_t in each iteration and, for the undirected version, the choice of the root z_1. Although it has been shown [GM93] that $v(LP_C)$ is independent of the choice of the root vertex, the lower bound calculated by DUAL-ASCENT is not. For this reason we start DUAL-ASCENT with different roots if a strengthening of the bound is necessary. Again, for the D-instances, considering up to ten different roots improves the average gap to 0.1%; achieving the optimum for 16 instances. The average running time for this lower bound calculation is 0.08 seconds. Further improvements can be achieved by using the reductions and upper bound calculations, which are done in combination with DUAL-ASCENT (see Sections 3.4.2 and 4.4). Some results on this can be found in [PV01c].

Even using this amplification, there are still cases where DUAL-ASCENT does not reach the value $v(LP_C)$. We tested different criteria for the choice of z_t in each iteration. Our standard criterion is: Choose z_t such that W is smallest. We had some success with the following idea that tries to guide DUAL-ASCENT with the help of a heuristically constructed Steiner tree: Assume that the upper bound is already optimal. DUAL-ASCENT can reach the optimum only if in each set $\delta^-(W)$ there is exactly one edge of the corresponding Steiner tree. Of course this criterion can not always be realized, especially if the best known Steiner tree is not optimal or $v(LP_C) < v(P_C)$. Nevertheless, it is a heuristic criterion that in many cases leads to better lower (and, indirectly, upper) bounds.

A natural alternative for a better approximation of $v(LP_C)$ builds upon a Lagrangian relaxation of the multicommodity flow formulation; an approach already used in in [Bea84] (but with the much weaker undirected relaxation; see also [HRW92]). Relaxing the constraints that bind edge and flow variables together, the problem decomposes into (mainly) $r-1$ single pair shortest path problems, which can be solved in time $O(r(m + n \log n))$. This relaxation has the integrality property, and can be used in combination with subgradient optimization to approximate $v(LP_C)$. In [PV97], we have investigated this approach and presented some improvements, particularly in combination with the algorithm DUAL-ASCENT and with sophisticated reduction techniques. Although this approach is quite effective in many cases, for large instances with many terminals it tends to be too slow. So, it is not used here and is replaced by the following approach.

Row Generating: To get an optimal solution for LP_C, one can begin with a subset of constraints of LP_C as the initial program, and successively solve the current program, find Steiner cut inequalities violated by the current solution x, add them to the program, and iterate this process by reoptimizing the program, until no Steiner cut inequality is violated anymore. This is an approach already used by many authors (see for example [CGR92, BL98, KM98]).

In order to find violated Steiner cut inequalities (or to establish that no such inequality exists), one can compute a minimum capacity cut in each of the $r-1$ flow networks constructed from G by choosing the root (z_1) as the source, a terminal $z_t \in R^{z_1}$ as the sink and the current x_{ij}-value as the capacity of the arc $[v_i, v_j]$. Although there are other (heuristical) ways to find such violated inequalities, using those corresponding to minimum cuts usually leads to better overall results. Indeed, it is even very advantageous to find in each case a minimum capacity cut with a minimum number of cut edges, an idea already used in [KM98]. This can be realized by adding a small ϵ to the capacity of each edge before solving the minimum cut problem. Although this leads to much denser flow networks, the linear programs obtained are easier to (re-)optimize (and the corresponding constraints seem to be much stronger). As a consequence, the overall results (especially the total number of necessary reoptimizations) are clearly superior. It must be mentioned that in our implementation, the time for finding all the $r-1$ minimum cuts is dominated by the time for reoptimizing the linear

programs.

For computing minimum cuts, we implemented the highest-label preflow-push algorithm with several auxiliary heuristics, including the global and the gap relabeling heuristics [CG97]. Although no better time bound than $O(n^2\sqrt{m})$ can be given for this algorithm, using the heuristics mentioned the empirical running times were much better described by $O(n^{1.5})$. As long as only minimum cuts from the sink-side are to be computed, only the first stage of the algorithm has to be performed. Besides, in this context several additional heuristics can be used to improve the empirical times further; for example, sinks which are reachable from the root (or another terminal) by paths of capacity no less than 1 need not be considered.

For (re-)optimizing the linear programs, we use the dual simplex routine in the callable library of CPLEX 8.0. Here, the warm-start ability of the simplex algorithm can be particularly utilized.

We have achieved considerable speedups by inserting the cuts generated by the algorithm DUAL-ASCENT into the initial linear program. In this case the lower bound provided by DUAL-ASCENT (which is often very close to $v(LP_C)$) is already reached in the first iteration; and the number of necessary reoptimizations and the time needed per reoptimization are comparable to the case without these cuts *after* reaching this lower bound value. As a consequence, the overall times are clearly improved.

As mentioned earlier, the flow balance constraints (10.1) can used to strengthen the linear programs. Empirically, we found it advantageous in terms of running times to insert all the flow-balance inequalities into the initial program. Although the other additional constraints used in [KM98] cannot enhance the value of the relaxation (see the corollaries of Theorem 13), a group of them can speed the process up if its violated members are added to the current program. These are constraints of the form of equations (8.2) of $LP_{\vec{T}}$.

To save time and space, we do some garbage collection every ten iterations, purging the constraints that have had large positive slack values in all the iterations since the last garbage collection. Further we make sure that no constraint is present in a linear program more than once.

Another idea, which is promising at first sight, is pricing: To achieve further speedups one can begin with a subset of variables as active variables and at certain stages (especially when a correct lower bound is necessary) add variables that do not price out correctly (have negative reduced costs) to the program (activate them); a correct lower bound is given when all non-active variables have non-negative reduced costs with respect to the current dual solution. We have tried several schemes for using this idea, but could not achieve decisive *additional* improvements through these schemes. The main reason is that because of our massive usage of reduction techniques (see the next chapter), most variables that could be priced out are eliminated anyway. It seems that the information provided by the linear relaxations (like reduced costs) are more effectively used in bound-based reduction techniques (see Section 3.4).

2.9.3 Using Tighter Relaxations

In this section, we will outline approaches for actually using tighter relaxations than LP_C algorithmically.

The complete common flow relaxation LP_{FR} is not a good starting point for building algorithms, because its exponential size makes it difficult (if not impossible) to come up with a practical algorithm using it. Simply writing down the linear program and starting an LP-solver fails already on problems of toy size. But also a branch-cut-price approach fails, because too many variables and constraints have to be included and each iteration of the column and row generation method can take too much time without making any substantial progress.

Although a more efficient realization cannot be ruled out, turning to the restricted version $LP_{C'}$ (see Section 2.7.5) of the common flow relaxation is a by far more appealing approach. Here one can start with the x variables and the flow-balance constraints (14.3), and then use a column and row generation approach. We work in three levels. In each iteration, we process a level only if the previous levels could not find a new row or column.

1. We look for violated Steiner cut constraints (14.1) as described in the previous section.

2. For those $y^{R\setminus\{z_r,z_i\}}$ that have already been inserted, we look for violated Steiner cut constraints (14.2), again with the same procedure as previously described.

3. Now, we are looking for violated constraints (14.4) of the form $y^{R\setminus\{z_r,z_i\}}(\delta^-(z_i)) \leq x(\delta^+(z_i))$ for some z_i. Here, one has to assume that all $y^{R\setminus\{z_r,z_i\}}$ that are not already included in the linear program are equal to x if $x(\delta^+(z_i)) > 0$. Otherwise, a lot of unnecessary y-variables would have to be included because of negative reduced costs. If we find violated constraints (14.4), we also include the necessary variables $y^{R\setminus\{z_r,z_i\}}$.

With this procedure the linear program will reach the value of $v(LP_{C'})$, which is in some cases significantly above the value of $v(LP_{F+FB})$ (see next section). Some other practical approaches for improving the lower bound are mentioned in Section 2.11.

2.10 Some Experimental Results

In this section, we present summarized results of some different approaches for computing lower bounds on bechmark instances from [SteinLib], see Table 2.2, (see Section 1.2 for a description of the instance groups). We measured running times and the average gap between the lower bound (rounded up to the next integer) and the known optimum. We give results for all groups of instances from SteinLib where all optimal values are known. A stroke in the table means that we were not able to compute the lower bound for some of the instances of this class, because of running time constraints.

The main purpose of this table is to provide a rough overview of the power of the different approaches. Additionally, a comparison of our implementation of the different approaches with those of others is now possible.

To ensure comparability we did not use reductions before starting the lower bound computation. Note that the computation of a lower bound on an unreduced instance is a very artificial setting in our context. Obviously, it cannot reveal the true value of a technique as part of exact algorithm or upper bound computation. Usually, one would first try all reduction techniques (even those using lower bound calculations themselves) before starting time consuming lower bound calculations. Reduction techniques influence the lower bound calculation in different ways:

1. As the graph may be smaller, the computation may be faster. Also the structure of the graph may change: Typically, the graph becomes sparser, which may be an advantage or a disadvantage.

2. In the case of approximative approaches for the solution of a relaxation (e.g., DUAL-ASCENT), the quality of the approximation of the lower bound is typically improved.

3. Additionally, the value of the optimal solution of the relaxation can improve.

On the other hand, measuring running times and lower bound gaps on reduced instances would make a comparison with other implementations impossible.

instance group	DUAL-ASCENT time (s)	DUAL-ASCENT gap (%)	DUAL-ASCENT, 10 roots time (s)	DUAL-ASCENT, 10 roots gap (%)	LP_C, row generation time (s)	LP_C, row generation gap (%)
1R	0.01	3.49	0.02	2.03	—	—
2R	0.01	5.87	0.04	4.41	—	—
D	0.02	0.40	0.08	0.10	662.29	0.00
E	0.12	0.26	0.62	0.25	5714.26	0.00
ES10000FST	257.30	1.19	2434.40	1.18	—	—
ES1000FST	0.31	1.20	2.58	1.16	264.53	0.008
I080	0.01	1.25	0.05	0.76	10.81	0.14
I160	0.04	1.10	0.31	0.83	4726.18	0.22
I320	0.17	1.24	1.55	1.05	—	—
LIN	0.39	2.49	3.39	1.88	—	—
MC	0.01	2.51	0.04	2.32	1941.06	0.54
TSPFST	1.72	0.67	16.95	0.60	5897.02	0.007
VLSI	0.23	2.00	2.16	1.51	—	—
WRP3	0.03	0.0006	0.20	0.0005	—	—
WRP4	0.01	0.0008	0.10	0.0006	929.24	0.000003
X	1.02	0.06	6.04	0.06	—	—

Table 2.2: Average results for lower bound calculations on unreduced instances.

We have not included values for LP_{F+FB} and $LP_{C'}$ in this table. The reason for this is that on those instance groups, where the improvements due to LP_{F+FB} and $LP_{C'}$ can show their strength, the computation of $v(LP_C)$ is already too time consuming on the weakly reduced instances. The only exception are the ES1000FST instances, where $LP_{C'}$ reduces the average gap from 0.008% of LP_C to 0.001%, which is a drastic improvement in the context of an exact algorithm.

2.11 Concluding Remarks on Lower Bounds

We have established a hierarchy of relaxations, providing very clear relations between relaxations. Furthermore, we introduced a collection of very tight linear relaxations for the Steiner problem. There are some polynomial variants in this collection that are stronger than any previously known relaxation of polynomial size. One of these relaxations can even be applied in practical algorithms and improve the performance of the current best approach for problem instances of the types mentioned.

Future work can be done in different directions: A theoretically important open question is whether a better worst-case guarantee for the integrality gap for any relaxation is possible. Practically, an improved scheme for using the presented formulations could lead to faster solutions of problem instances of SteinLib, where a small gap in linear relaxation is the crucial problem.

Also a constraint generation approach that is not based on some integer formulation, but on the "local cuts" approach as presented for the TSP by Applegate, Bixby, Chvátal, and Cook [ABCC01], could be applicable in this context. We made some steps in this direction by deriving and implementing some exact and heuristic projection methods, but we could apply them successfully only in rare cases. Our current version is able to improve the value of $v(LP_{C'})$ in the cases it does not reach $v(P_{C'})$. But either the problem instances could be solved faster by using bound-based reductions or branching, or neither of the approaches is able to solve the instance (this holds for those problems that are insightfully constructed to fool the currently known methods). We conjecture that one reason why we could improve our results only rarely with the "local cuts" approach is our partitioning-based reduction techniques (see Section 3.6) that exploit locality even more effectively than the "local cuts" approach.

Recently, we have developed a new approach for improving the lower bound for the Steiner problem that works very well in practice. The main idea is to expand the network such that the value of the

linear relaxation is improved. This idea is inspired by the column replacement techniques that were introduced by Balas and Padberg [BP75] and generalized by Haus et. al. [HKW01] and Gentile et. al. [GHK+02]. In these and other papers a general technique for solving integer programs is developed. However, these techniques are mainly viewed as primal algorithms, and extensions for combinatorial optimization problems are presented for the Stable Set problem only. Furthermore, these extensions are not yet part of a practical algorithm (the general integer programming techniques have been applied successfully). Thus, we are the first to apply this basic idea in a practical algorithm for a concrete combinatorial optimization problem.

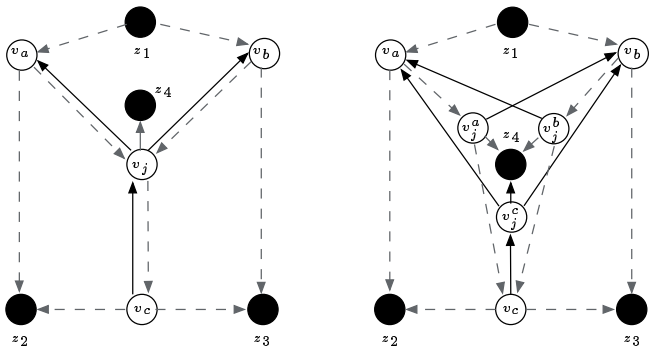


Figure 2.9: Splitting of vertex v_j. The filled circles are terminals, z_1 is the root, all arcs have cost 1. An optimal Steiner arborescence has value 6 in each network. In the left network $v(LP_{C+FB})$ is 5.5 (set the x-values of the dashed arcs to 0.5 and of $[v_j, z_4]$ to 1), but 6 in the right network (again, set the x-values of the dashed arcs and of $[v_j^a, z_4]$ and $[v_j^b, z_4]$ to 0.5). The difference is that in the left network, there is a situation that is called "rejoining of flows": Flows from z_1 to z_2 and from z_1 to z_3 enter v_j on different arcs, but leave on the same arc, so they are accounted in the x variables only once. Before splitting, the x-value corresponding to the arc $[v_j, v_c]$ is 0.5, after splitting the corresponding x-values sum up to 1.

As the algorithm is still under development, we will only sketch the basic operation here. This operation is the splitting of a vertex as depicted in Figure 2.9. A simplified version of the splitting operation is described by the following pseudocode:

SPLIT-VERTEX(G, v_j) : (assuming $v_j \in V \setminus R$)
1 **forall** $[v_i, v_j] \in \delta^-(v_j)$:
2 *insert a new vertex* v_j^i *into* G
3 *insert an arc* $[v_i, v_j^i]$ *with cost* $c(v_i, v_j)$ *into* G
4 **forall** $[v_j, v_k] \in \delta^+(v_j)$:
5 *insert an arc* $[v_j^i, v_k]$ *with cost* $c(v_j, v_k)$ *into* G
6 *delete* v_j

The transformation is valid, as any optimal Steiner tree of the original network can be transformed into a feasible Steiner tree in the new network with the same cost and vice versa. The reasoning behind this transformation is that in an optimal directed Steiner tree, each vertex has at most indegree 1.

Details on the two methods are presented in a technical report [APV03].

Another area for further development are the polynomial approximation algorithms for LP_F and LP_C by Garg and K¨onemann [GK98] and Garg and Khandekar [GK02]. We did some preliminary tests with these methods, but we could not make them as effective as other approaches.

Chapter 3

Simplifying Problem Instances Using Reduction Techniques

3.1 Introduction

Intuitively spoken, reduction techniques reduce the size of an original problem instance without changing the optimal solution. It has been known for some time that reductions can play an important role as a preprocessing step when one wants to deal with $\mathcal{NP}$-hard problems algorithmically. In particular, the importance of reduction techniques for the Steiner problem has been widely recognized and a large number of techniques were developed, with the PhD thesis of Cees Duin [Dui93] being a milestone.

We will not give a formal definition of what a reduction is, an intuitive understanding is sufficient:

1. Some *reduction condition* of a *reduction test* is satisfied (e.g., an edge (v_i, v_j) is longer than the length of a shortest path between its endpoints v_i and v_j).

2. By this, we get some knowledge about the optimal solutions of the problem. In the example, we know that an optimal solution exists that does not contain (v_i, v_j), in fact no optimal solution contains (v_i, v_j).

3. We transform the problem instance into one that is believed to be simpler, but we know that there is some efficient way to transform any optimal solution of the reduced problem instance into an optimal solution of the original instance. In the example, we can simply delete the edge (v_i, v_j), a transformation procedure is not necessary.

We distinguish some major classes of reduction tests:

- The **alternative-based** tests (Section 3.3) use the existence of alternative solutions. For example in case of exclusion tests, it is shown that for any solution containing a certain part of the graph (e.g., a vertex or an edge) there is an alternative solution of no greater cost without this part; the inclusion tests use the converse argument.

- The **bound-based** tests (Section 3.4) use a lower bound for the value of an optimal solution with the additional constraint that a certain part of the graph is contained (in case of exclusion tests) or is not contained (in case of inclusion tests) in the solution; these tests can reduce the problem if such a lower bound exceeds a known upper bound. This can also be interpreted as a kind of an implicit branch-and-bound approach.

- The **extended** tests (Section 3.5) use backtracking to extend the scope of inspection of the reduction tests, here bound-based and alternative-based approaches can be combined.

- The **partitioning-based** tests (Section 3.6) exploit a partitioning of the network to derive reductions using the (optimal) solutions of subnetworks.

Our work in this field is threefold:

1. We design efficient realizations of some classical reduction tests, which would have been too time-consuming for large instances in their original form, especially for application in heuristics.

2. We design some new tests, filling some of the gaps the classical tests had left. Notice that each test is specially effective on a certain type of instances, having less (or even no) effect on others, so it is important to have a large arsenal of tests at one's disposal.

3. We integrated these tests into a program packet. It should be emphasized that the most impressive achievements of reductions are mainly due to the interaction of different tests, achieving results that are incomparable to those each single test could achieve on the same instance on its own.

3.2 Additional Definitions for Reductions

The **bottleneck distance** $b(v_i, v_j)$ between v_i and v_j is the length of the shortest edge among the longest edges taken over all paths from v_i to v_j in G. The **restricted bottleneck distance** $\bar{b}(v_i, v_j)$ between v_i and v_j is the bottleneck distance between v_i and v_j in G excluding (v_i, v_j). An **elementary path** is a path that contains terminals only at the endpoints. Any path P between two vertices v_i and v_j in a network G can be broken at inner terminals into one or more elementary paths. The **Steiner distance** between v_i and v_j along P is the length of the longest elementary path in P. The **bottleneck Steiner distance** (sometimes also called "special distance") $s(v_i, v_j)$ between v_i and v_j is the minimum Steiner distance taken over all paths between v_i and v_j in G. The **restricted bottleneck Steiner distance** $\bar{s}(v_i, v_j)$ between v_i and v_j is the bottleneck Steiner distance between v_i and v_j in G excluding (v_i, v_j). Note that if $R = V$, then $b_{ij} = s_{ij}$ and $\bar{b}_{ij} = \bar{s}_{ij}$.

A **key node** in a tree T is a node that is either a nonterminal of degree at least 3 or a terminal. The unique path in T between two vertices $v_i \in T$ and $v_j \in T$ is called the **fundamental path**. A **tree bottleneck** between $v_i \in T$ and $v_j \in T$ is a longest subpath on the fundamental path between v_i and v_j in T in which only the endpoints may be key nodes; and t_{ij} denotes the length of such a tree bottleneck.

3.3 Alternative-based Reductions

In this subsection we present a collection of alternative-based tests, including some new versions of classical tests and some new tests, which can all be realized in time $O(m + n \log n)$.

3.3.1 PT_m and Related Tests

The test PT_m (Paths with many Terminals) was introduced in [DV89]:

PT_m test: Every edge (v_i, v_j) with $c(v_i, v_j) > s(v_i, v_j)$ can be removed from G. [DV89]

Proof: Suppose all Steiner minimal trees contain an edge (v_i, v_j) with $c_{ij} > s_{ij}$. Let T be such a tree. Removing (v_i, v_j) from T divides it into two components. Let P be a path between v_i and v_j with the Steiner distance s_{ij}. On this path there is at least one elementary path P' that connects the two components of T. The tree T becomes shorter, if (v_i, v_j) is replaced by P', a contradiction. □

The PT_m test is one of the most effective classical exclusion tests, but in its original form it is too time-consuming for large instances. Here we consider a fast realization of this test, which also uses inaccurate information. The modifications follow the same principal ideas as in [Dui93]. Later we will simply refer to this modified version as the PT_m test. Experimentally, one generally observes only a marginal difference in the effectiveness of the original test and its modified version.

For two terminals z_i and z_j, one observes that the bottleneck Steiner distance $s(z_i, z_j)$ can be computed by determining a bottleneck on the fundamental path between z_i and z_j in the spanning tree $T'_D(R)$, which can be constructed in time $\text{O}(m + n \log n)$ [Meh88]. Each such bottleneck can be trivially computed in time $O(r)$, leading to a total time $O(qr)$ for q inquiries ($q \in O(\min\{m, r^2\})$). Observing that one actually has a static-tree variant of the bottleneck problem, one can use a strategy

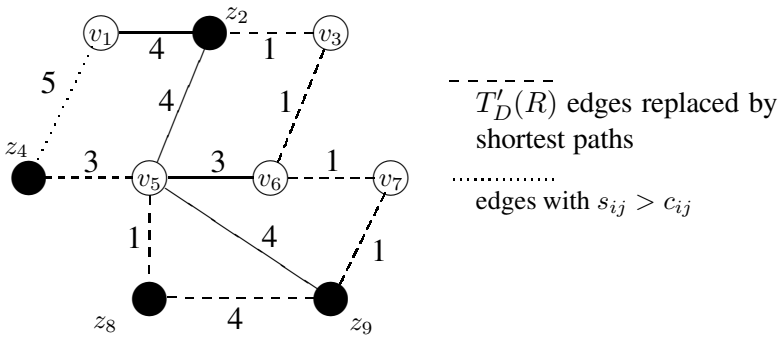


Figure 3.1: Example for the PT$_m$ test.

based on depth-first search (as described in [VJ83]) to achieve time $\Theta(r^2)$ for all inquiries. One can go further and solve the problem as an off-line variant for all q inquiries in time $O(q\alpha(m,r))$ using the Eval-Link-Update data structure [Tar79]. But this data structure is rather complex and leads to relatively large constant factors, and this bound is dominated by the worst-case time of other test operations anyway. So we suggest another method to achieve the desired worst-case time $O(m+n\log n)$ for the whole test: One can sort the edges of $T'_D(R)$ and then process them as links in increasing cost order, building a binary tree (whose internal nodes represent the edges of $T'_D(R)$) using a suitable auxiliary union-find data structure. This transforms the problem to an instance of the off-line nearest-common-ancestor problem, which is solvable, for example, in $O(q)$ using a depth-first search strategy [Tar79]. This leads to a total time $O(q + r\log r)$ for all q inquiries.

For non-terminals v_i and v_j, one can use an upper bound for the bottleneck Steiner distance $s(v_i,v_j)$ considering only paths of the form $v_i - z_{i,a} - z_{j,b} - v_j$, where $z_{i,a}$ and $z_{j,b}$ are the a-th respectively b-th nearest terminals to v_i and v_j. The k (k constant, say 3) nearest terminals to all non-terminals (forbidding intermediary terminals on the corresponding paths) can be computed using a modification of the algorithm of Dijkstra in time $O(m + n\log n)$, as described in [Dui93]. After that, one works with the upper bound $\hat{s}(v_i,v_j) := \min_{a,b\in\{1,\ldots,k\}}\{\max\{\ d(v_i,z_{i,a})\ ,\ s(z_{i,a},z_{j,b})\ ,\ d(z_{j,b},v_j)\}\}$ instead of $s(v_i,v_j)$. But we do not precompute the $\hat{s}$-values, because very often not all the k^2 combinations have to be checked; for example if the test condition turns out to be already satisfied during the computation (or, of course, if v_i or v_j were a terminal). More importantly, many additional observations can be used to do without $\hat{s}(v_i,v_j)$ altogether. For example the lower bound $\tilde{s}(v_i,v_j) := \max\{d(v_i,\mathit{base}(v_i)), d(v_j,\mathit{base}(v_j))\}$ (which is readily available) is often helpful: If both vertices v_i and v_j belong to the same Voronoi region, then we simply have $\tilde{s}(v_i,v_j) = \hat{s}(v_i,v_j)$. If v_i and v_j belong to different Voronoi regions and $c(v_i,v_j) < \tilde{s}(v_i,v_j)$, then the test cannot be successful for (v_i,v_j). Furthermore, precomputing the $\hat{s}$-values (which can need time $\Theta(n^2)$) would destroy the total time $O(m + n\log n)$ for performing this test on all edges.

An additional lemma leads to a simple, very fast test, which is sometimes very powerful:

Lemma 20 Let $\hat{S}$ be the length of the longest edge in $T'_D(R)$. Every edge (v_i,v_j) with $c(v_i,v_j) > \hat{S}$ can be removed from the network.

Proof: Suppose there is a Steiner minimal tree T containing an edge (v_i,v_j) with $c_{ij} > \hat{S}$. Removing this edge from T divides it into two components: C_i containing v_i and C_j containing v_j. In each component, there is at least one terminal. Let z_k and z_l be two arbitrary terminals in C_i respectively C_j. In G, there is a path between z_k and z_l, corresponding to the fundamental path in $T'_D(R)$, with

Steiner distance at most $\hat{S}$. This path contains an elementary path P connecting C_i and C_j, whose length is at most $\hat{S}$. Reconnecting C_i and C_j by P yields a graph spanning all terminals and shorter than T, a contradiction. □

Note that using this test, one can eliminate some edges that could not be eliminated by the PT$_m$ test (even in its original form).

Since the tests above only consider paths with at least one terminal, they miss some of the edges the simple test LE (Long Edges) [Bea84, HRW92] would eliminate. On the other hand, after execution of other tests the graph is often sparse. So a weakened version of LE, which simply searches for shorter paths from both ends of an edge, can be effective. With the additional restriction that during the examination of each edge not more than a constant number of edges are visited in search for an alternative path, one gets the total time $\Theta(m)$ for this modified test, which we call **Triangle**. This test is sometimes a nice complement to the PT$_m$ test (as described above), especially if the proportion of terminals to all vertices is not high.

Improving the Test in the Case of Equality

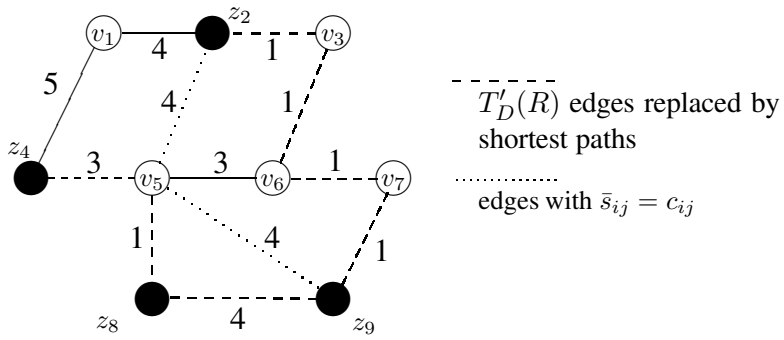


Figure 3.2: Improvement for $\bar{s}_{ij} = c_{ij}$.

The test PT$_m$ can actually be extended to the case of equality with the restricted bottleneck Steiner distance: An edge (v_i, v_j) can be removed from G if $c_{ij} \geq \bar{s}_{ij}$ (remember that $\bar{s}_{ij}$ is s_{ij} in G after removing (v_i, v_j)). Figure 3.2 shows an example of edges that can be removed using this stronger test condition.

But removing edges with this test condition can change the (restricted) bottleneck Steiner distances, which makes a recalculation of these distances after each deletion necessary. In Figure 3.3, this case is shown: The edge (z_2, v_4) can be removed, because the restricted bottleneck Steiner distance is $c(z_2, v_4) = \bar{s}(z_2, v_4) = s(z_2, v_4) = 3$. Similarly, the edge (v_4, z_5) can be removed, because $c(v_4, z_5) = \bar{s}(v_4, z_5)$. But after removing the edge (z_2, v_4) the restricted bottleneck Steiner distances change, in particular $\bar{s}(v_4, z_5) = \infty$. Thus, only one of the two edges (z_2, v_4) and (v_4, z_5) can be deleted. These problems were the reason why in the literature on the Steiner problem it was assumed that the test PT$_m$ cannot be performed for the case of equality without recalculation of the necessary information.

We have observed that the problematic cases can be efficiently identified, such that in all other cases the test actions can be performed even in case of equality (without recalculation) and using the approximated Steiner bottleneck distances $\hat{s}$ (which are computed in advance) in the test condition $c_{ij} \geq \hat{s}_{ij}$. First, we list the problematic cases for $\hat{s}_{ij} = c_{ij}$ and show how to deal with them. After that, we prove that other problematic cases cannot exist.

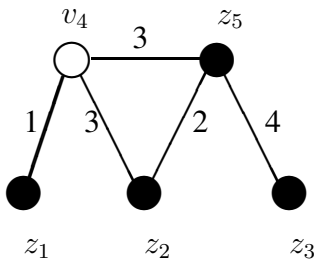


Figure 3.3: Critical cases with $\bar{s}_{ij} = c_{ij}$.

I) Edges from $T'_D(R)$: Obviously, $\hat{s}_{ij} = c_{ij}$ holds for every edge $(z_i, z_j) \in E$, $z_i \in R$, $z_j \in R$ that is contained in the tree $T'_D(R)$. In this case, we do not want the edge to be deleted, as we do not know the value of $\bar{s}_{ij}$. Thus, edges $(z_i, z_j) \in E$ with both endpoints being terminals will only be deleted if $(c_{ij} > \hat{s}_{ij}) \vee (c_{ij} = \hat{s}_{ij} \wedge (z_i, z_j) \notin T'_D(R))$.

As an example take the edges (z_2, z_5) and (z_3, z_5) in Figure 3.3. Although $c_{ij} \geq \hat{s}_{ij}$, the edges will not be deleted, as they are contained in $T'_D(R)$.

II) Edges between a terminal and a non-terminal: For an edge $(z_i, v_j) \in E, z_i \in R, v_j \in V \backslash R$, $\hat{s}_{ij}$ can be equal to c_{ij}, arguing with the path consisting of the edge (z_i, v_j). In Figure 3.3, this is shown for the edge (z_1, v_4).

Therefore, an edge (z_i, v_j) will only be deleted, if $c_{ij} < \hat{s}_{ij}$ or if $c_{ij} = \hat{s}_{ij}$ and the path from v_j to z_i with Steiner distance $\hat{s}_{ij}$ that was used in the calculation of $\hat{s}_{ij}$ does not contain the edge. The last condition can be checked easily by maintaining "parent pointers" in the shortest path calculation.

III) Multiple edges to terminals with equal length: If two edges $(v_i, z_j), (v_i, z_k) \in E$ have the same length and the bottleneck Steiner distance between z_j and z_k is not longer than the length of the edges, then any of the edges may be deleted (arguing with a path containing the other edge). But both edges may not be deleted, because after the deletion of the first edge, the argument for the deletion of the other is no longer valid. In Figure 3.3, this holds for the edges (v_4, z_2) and (v_4, z_5) as described above.

We have solved this problem in the following way: If an edge is deleted under the circumstances described (e.g., (v_4, z_2)), the other edge (in this case (v_4, z_5)) is marked. A marked edge (v_i, v_j) will only be deleted if $c_{ij} > \hat{s}_{ij}$.

Now, we look at the problem from a different point of view. Under which circumstances does the following situation arise: An edge $e_1 \in E$ has been deleted arguing with a path $P_1, e_1 \notin P_1$, with Steiner distance $\hat{s}(e_1) \leq c(e_1)$. Then, an edge $e_2 \in E, e_2 \neq e_1$ is about to be deleted, arguing with a path P_2 of the original network with Steiner distance $\hat{s}(e_2) \leq c(e_2)$. But e_2 may not be deleted, because some part of P_2 has been deleted and it is not possible to find a replacement for it in the current network G_2.

Lemma 21 The three cases mentioned – denoted by the numbers I), II), and III) – are the only cases where $c(e_2) \geq \hat{s}(e_2)$ holds for some edge e_2, but in the current network G_2 it holds $c(e_2) < \bar{s}(e_2)$.

Proof: We have to distinguish several cases:
1.: $e_2 \in P_2$. This is either case I) or II).
2.: $e_1, e_2 \notin P_2$. As P_2 is in G_2, $\hat{s}(e_2) \geq \bar{s}(e_2)$ holds, a contradiction.
3.: $e_2 \notin P_2, e_1 \in P_2 \qquad (\Rightarrow c(e_1) \leq \hat{s}(e_2))$.
3.1.: $e_2 \notin P_1$. We build a new path P_2' with the same Steiner distance $\hat{s}(e_2)$ that is contained in G_2, by replacing e_1 by the path P_1. It follows that $\hat{s}(e_2) \geq \bar{s}(e_2)$, a contradiction.
3.2.: $e_2 \in P_1 \qquad (\Rightarrow c(e_2) \leq \hat{s}(e_1))$.
3.2.1.: $c(e_1) < \hat{s}(e_2)$. It follows that $c(e_1) < \hat{s}(e_2) \leq c(e_2) \leq \hat{s}(e_1) \leq c(e_1)$, a contradiction.
3.2.2.: $c(e_1) = \hat{s}(e_2)$. It follows that $c(e_1) = \hat{s}(e_1) = c(e_2) = \hat{s}(e_2)$.
3.2.2.1.: Both endpoints of e_1 are terminals. Together with $e_1 \in P_2$ it follows that $e_1 \in T_D'(R)$. This is case I) and e_1 has not been deleted.
3.2.2.2.: Exactly one endpoint of e_1 is a terminal. As $e_1 \in P_2$ and P_2 has Steiner distance $\hat{s}(e_2) = c(e_1)$, one endpoint of e_2 must be the non-terminal endpoint of e_1. This is case III).
3.2.2.3.: No endpoint of e_1 is a terminal. As $e_2 \neq e_1$, the path P_2 must consist of more edges than just e_1. But then P_2 must have a Steiner distance greater than $c(e_1)$, a contradiction. □

It is easy to see that this proof also extends to the cases that no edge or more than one edge have been deleted before e_2 is inspected.

It must be mentioned that this strengthening of the reduction test has a greater impact than one would assume, because in some cases the reduction process is blocked in face of many alternatives with equal weights.

3.3.2 NTD$_k$

The test NTD$_k$ (Non-Terminals of Degree k) was introduced in [DV89]:

NTD$_k$ test: A non-terminal v_i has degree at most 2 in at least one Steiner minimal tree if for each set Δ, $|\Delta| \geq 3$, of vertices adjacent to v_i the following holds: The sum of the lengths of the edges between v_i and vertices in Δ is not less than the weight of a minimum spanning tree for the network $(\Delta, \Delta \times \Delta, s)$.
If this condition is satisfied, one can remove v_i and incident edges, introducing for each two vertices v_j and v_k adjacent to v_i an edge (v_j, v_k) with length $c_{ij} + c_{ik}$ (and keeping only the shortest edge between each two vertices).

The special cases with k (degree of v_i) in $\{1, 2\}$ can be implemented with total time $O(n)$ (for examination of all non-terminals). For $k \in \{3, \ldots, 7\}$ we use the $\hat{s}$-values instead of the exact bottleneck Steiner distances, as described in Section 3.3.1. Again, empirically only a marginal difference in effectiveness is observed between the original and the modified version. As before, we do not precompute the $\hat{s}$-values, so the modified version has total time $O(m + n \log n)$.

Because the addition of new edges can be a nontrivial matter and the necessary $\hat{s}$-values are already available, it is a good idea to check whether each new edge could be eliminated using the PT$_m$ test. In this case it need not be inserted in the first place.

3.3.3 NV and Related Tests

The test NV (Nearest Vertex) is a classical inclusion test [Bea84, HRW92]:

NV test: Let z_i be a terminal with degree at least 2, and let (z_i, v_i') and (z_i, v_i'') be the shortest and second shortest edges incident to z_i. The edge (z_i, v_i') belongs to at least one Steiner minimal tree, if there is a terminal $z_j, z_j \neq z_i$, with $c(z_i, v_i'') \geq c(z_i, v_i') + d(v_i', z_j)$.

The original version of the test NV requires the computation of distances, which is too time-consuming for large instances. But one can accelerate this test without making it less powerful, using the lemma given below. For this purpose, we use Voronoi regions again, saving some extra information while computing the regions. Let $distance(z_i)$ be the length of a shortest path from z_i to another terminal z_j over the edge (z_i, v_i'), computed as follows: Each time an edge (v_i, v_j) with $v_i \in N(z_i), v_j \in N(z_j), z_j \neq z_i$ is visited, it is checked whether v_i is a successor of v_i' in the shortest paths tree with root z_i (simply done through marking the successors of v_i'). In such a case $distance(z_i)$ is updated to $\min\{distance(z_i), d(z_i, v_i) + c(v_i, v_j) + d(v_j, z_j)\}$. Now we have:

Lemma 22 The condition of the test NV is satisfied if and only if:
$c(z_i, v_i'') \geq c(z_i, v_i') + d(v_i', base(v_i'))$, if $v_i' \notin N(z_i)$, and
$c(z_i, v_i'') \geq distance(z_i)$, if $v_i' \in N(z_i)$.

Proof: Assume the condition formulated in the lemma is satisfied for a vertex z_i: If $v_i' \notin N(z_i)$, the NV test condition is satisfied for $z_j = base(v_i')$. If $v_i' \in N(z_i)$, then a terminal z_j exists with $c(z_i, v_i') + d(v_i', z_j) = distance(z_i) \leq c(z_i, v_i'')$. Hence, the NV test condition is satisfied.
Now assume that the condition of the test NV, $c(z_i, v_i'') \geq c(z_i, v_i') + d(v_i', z_j)$, is satisfied: If $v_i' \notin N(z_i)$, it follows from $d(v_i', z_j) \geq d(v_i', base(v_i'))$ that $c(z_i, v_i'') \geq c(z_i, v_i') + d(v_i', base(v_i'))$. If $v_i' \in N(z_i)$, we could get $c(z_i, v_i') + d(v_i', z_j) \geq distance(z_i)$, assuming that v_i' is on a shortest path between z_i and z_j. But the latter must be true, because otherwise we have $c(z_i, v_i'') \geq c(z_i, v_i') + d(v_i', z_j) > d(z_i, z_j) \geq c(z_i, v_i'')$, a contradiction. □

Using this lemma, the test NV can be performed for all terminals in time $O(m + n \log n)$. Note that in inclusion tests, each included edge is contracted into a terminal.

The Voronoi regions can also be used to perform a related inclusion test, which we call **SL** (standing for Short Links):

Lemma 23 Let z_i be a terminal, and (v_1, v_1') and (v_2, v_2') the shortest and second shortest edges that leave the Voronoi region of z_i ($v_1, v_2 \in N(z_i)$, $v_1', v_2' \notin N(z_i)$; we call such edges **links**). The edge (v_1, v_1') belongs to at least one Steiner minimal tree, if $c(v_2, v_2') \geq d(z_i, v_1) + c(v_1, v_1') + d(v_1', z_j)$, where $z_j = base(v_1')$.

Proof: Suppose that the edge (v_1, v_1') is not in any Steiner minimal tree. Consider such a tree T and the path between z_i and z_j in T. An edge on this path must leave the Voronoi region of z_i. Removing this edge and inserting (v_1, v_1') and two shortest paths to z_i and to z_j, we get a subgraph that includes (v_1, v_1'), spans all terminals and is no longer than T, a contradiction. □

This test can also be performed for all terminals in total time $O(m + n \log n)$.

The classical test SE (Short Edges) [DV89, HRW92] is a more powerful inclusion test. We have observed that even this test can be implemented with time $O(m + n \log n)$. But although this test is more effective than NV and SL in a single application, the difference almost vanishes when the reduction tests are iterated. Therefore, we only use the much simpler, empirically faster tests NV and SL in our actual implementations.

3.3.4 Path Substitution (PS)

We have designed another alternative-based reduction test that is more general than the previous tests in two ways: The test PS examines several edges along a path, instead of examining elementary graph objects (like single edges and vertices). If the test is successful, some of these edges can be deleted at the same time. The other more general aspect is a consequence of the first: Searching for alternatives

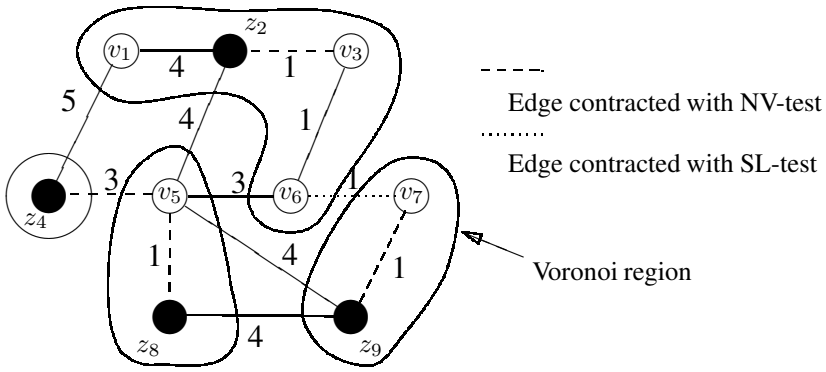


Figure 3.4: Example for the application of the NV- and SL-tests.

for a path, it is no longer sufficient to find *one* alternative, because the edges of the path can be involved in many different ways in a Steiner tree. As a consequence, such a test can only be efficient if it has a rather restricted test condition.

The basic idea is to start with a single edge as the path and then try to find alternative paths for the vertices adjacent to those on the path. If this is not possible for exactly one adjacent vertex, the path is extended by the edge to this vertex and the search for alternative paths is restarted. Such successive extensions could finally lead to the desired situation.

We describe the lemma that leads to the formal specification of the test in a simplified way: We give only the description for deleting *one* edge of the path and define it only for the special case that the starting vertex v_0 has degree 3. The extensions to deleting many edges on the path and to vertices with higher degree are not too complicated. (For $degree(v_0) > 3$ the additional condition $d_P(v_0, v_i) \geq d_0(v_0^{k_0}, v_i^k) + c(v_i, v_i^k)$ is required in the last line of the definition of the test condition.)

Lemma 24 Let P be a path $(v_0, \ldots, v_l)$ with $degree(v_0) = 3$ and $v_i \in V \backslash R$ for all $i \in \{0, \ldots, l\}$. We denote by $v_i^1, v_i^2, \ldots$ the vertices adjacent to each v_i on P that are not contained in P. Let $d_0(v_i, v_j)$ be the length of a shortest path between v_i and v_j that does not contain (v_0, v_1), and $d_P(v_i, v_j)$ (for v_i and v_j in P) the length of the subpath of P between v_i and v_j.
The edge (v_0, v_1) can be deleted if for all $i \in \{1, \ldots, l\}$ there are functions f^i and g^i such that:
I) for all v_i^k adjacent to v_i and for $k_0 = f^i(k)$: $d_P(v_0, v_i) \geq d_0(v_0^{k_0}, v_i^k)$,
II) for all $v_0^{k_0}$ adjacent to v_0 and for $k = g^i(k_0)$: $d_P(v_0, v_i) \geq d_0(v_0^{k_0}, v_i^k),\ c(v_0, v_0^{k_0}) \geq c(v_i, v_i^k)$.

Proof: Suppose all Steiner minimal trees contain the edge (v_0, v_1). Consider such a tree T, and let $t \geq 1$ be the smallest index such that there is an edge (v_t^k, v_t) in T. Notice that the degree of v_0 in T must be greater than 1 and that all edges between v_0 and v_t on P must be in T. There are two cases:
1) In T, v_0 has degree three. Choose k such that (v_t^k, v_t) is in T. Let $k_0 = f^t(k)$. Remove the edges on the path $(v_0, v_1, \ldots, v_{t-1}, v_t)$ from T. The resulting components can be reconnected without reinserting (v_0, v_1) by a path between $v_0^{k_0}$ and v_t^k which is not longer.
2) In T, v_0 has degree two. Choose k_0 such that $(v_0^{k_0}, v_0)$ is in T. Let $k = g^t(k_0)$. Remove the edges on the path $(v_0^{k_0}, v_0, v_1, \ldots, v_{t-1}, v_t)$ from T. The resulting components can be reconnected without reinserting (v_0, v_1) by a path between $v_0^{k_0}$ and v_t^k and the edge (v_t^k, v_t). Again the inserted edges together are not longer than the removed edges.
In both cases, we have a subgraph that does not contain (v_0, v_1), spans all terminals and is not longer than T, a contradiction. □

One problem for an efficient implementation of this test is the calculation of the distances $d_0(v_i, v_j)$. Since we do not want to have running times like $\Theta(n^3)$ for the calculation of shortest paths, we work with a weakened version: To determine an upper bound for $d_0(v_0^{k_0}, v_t^k)$, we examine only those paths that contain only vertices in $\{v_{t'}^{k'} \mid 0 \leq t' \leq t\}$. This makes it easy to maintain shortest paths trees for each $v_0^i, i \in \{1, \ldots, \mathit{degree}(v_0) - 1\}$, during the successive extensions of P. It is also possible to determine up to which vertex v_s in P the edge (v_s, v_{s+1}) can be deleted, under the assumption that all edges between v_0 and v_s have been deleted. If finally a situation is reached in which – according to the lemma above – (v_0, v_1) can be deleted, then all edges of P between v_0 and v_{s+1} can be removed. Our implementation assures that each edge is considered as a part of P not more than twice (once in each direction). We have observed that if the test is successful, all involved vertices have low degrees. If one fixes a small constant g, e.g. $g = 10$, and aborts the successive extension of P each time a vertex with degree larger than g is visited, a total running time (for the whole network) of $O(m)$ can be guaranteed, without impeding its reduction potential noticeably.

This version of the test is usually effective only for some sparse graphs (including some VLSI-instances). For such instances, 5-10% of edges could frequently be removed using this test alone.

3.4 Bound-based Reductions

Since one cannot expect to solve all instances of an $\mathcal{NP}$-hard problem like the Steiner problem only through reduction tests with a (low order) polynomial worst-case time (like the tests in the previous subsection), the computation of (sharp) lower bounds is a common part of the standard algorithms for the exact solution of such a problem. But the information gained during such computations can be used to reduce the instance further; and sometimes small worst-case running times can be guaranteed even for this kind of tests.

3.4.1 Using Voronoi Regions

The Voronoi regions can be used to determine a lower bound for the value of an optimal solution with additional constraints (for example, that the solution contains a certain non-terminal). For any terminal z, we define $\mathit{radius}(z)$ as the length of a shortest path from z leaving its Voronoi region $N(z)$. These values can be easily determined while computing the Voronoi regions. For convenience, we assume here that the terminals are numbered according to non-decreasing radius-values. For each non-terminal v_i, let $z_{i,1}, z_{i,2}$ and $z_{i,3}$ be the three terminals next to v_i, as described in Section 3.3.1. The following lemma can be used to eliminate a non-terminal.

Lemma 25 Let T be a Steiner minimal tree and assume that v_i is a Steiner node in T. Then $d(v_i, z_{i,1}) + d(v_i, z_{i,2}) + \sum_{t=1}^{r-2} \mathit{radius}(z_t)$ is a lower bound for the weight of T.

Proof: For each terminal z_l, we denote the path between z_l and v_i in T with P_l. Among such paths, there must be at least two (edge-)disjoint ones. For any path P, define $\Delta(P)$ as the number of edges on P that have their vertices in two different Voronoi regions. Let P_j and P_k be two disjoint paths such that $\Delta(P_j) + \Delta(P_k)$ is minimal. Note that no path P_l can have edges in common with both P_j and P_k. For each terminal $z_l \notin \{z_j, z_k\}$, let P_l' be the part of P_l from z_l up to the first vertex not in $N(z_l)$; P_l' is well-defined, because otherwise P_l would be the only path with $\Delta(P_l) = 0$ (namely for $z_l = \mathit{base}(v_i)$) and would have been chosen as P_j or P_k. Obviously, all P_l' are disjoint. Now suppose that P_j has an edge in common with some P_l'. Let v_l be a vertex of this edge with $v_l \in N(z_l)$. The part of P_j between z_j and v_l contains an edge with only one vertex in $N(z_l)$, so $\Delta(P_l) < \Delta(P_j)$,

which contradicts the choice of P_j. So P_j (or, similarly, P_k) has no edge in common with a path P'_l. Since P_j, P_k and the $r-2$ paths P'_l are all disjoint, the sum of their lengths cannot be larger than the weight of T. The sum of the lengths of P_j and P_k is at least $d(v_i, z_{i,1}) + d(v_i, z_{i,2})$. The sum of the lengths of the $r-2$ paths P'_l is at least $\sum_{t=1}^{r-2} radius(z_t)$. □

A non-terminal v_i can be eliminated if this lower bound exceeds a known upper bound. This method can be extended for eliminating edges:

Lemma 26 Let T be a Steiner minimal tree and assume that T contains an edge (v_i, v_j). Then $c(v_i, v_j) + d(v_i, z_{i,1}) + d(v_j, z_{j,1}) + \sum_{t=1}^{r-2} radius(z_t)$ is a lower bound for the weight of T.

Proof: Analogous to the proof of Lemma 25. □

One can also define a test performing the same actions as NTD$_k$ when it is successful, using the following lemma:

Lemma 27 Let T be a Steiner minimal tree and assume that v_i is a Steiner node whose degree in T is at least three. Then $d(v_i, z_{i,1}) + d(v_i, z_{i,2}) + d(v_i, z_{i,3}) + \sum_{t=1}^{r-3} radius(z_t)$ is a lower bound for the weight of T.

Proof: Analogous to the proof of Lemma 25. □

Intuitively, one expects that an even better lower bound should be achievable through this line of argument, because the paths between the terminals in a Steiner tree not only leave the corresponding Voronoi regions, but also span all terminals. Indeed, one can use this idea:

Lemma 28 Consider the auxiliary network $G' = (R, E', d')$, in which two terminals are adjacent if and only if they are neighbors in the original network, defining:
$d'(z_i, z_j) := \min\{\min\{d(z_i, v_i), d(z_j, v_j)\} + c(v_i, v_j) \quad | \quad v_i \in N(z_i), v_j \in N(z_j)\}$.
The weight of a minimum spanning tree for G' is a lower bound for the weight of any Steiner tree for the original instance (G, R).

Proof: We will prove the lemma by transforming a Steiner minimal tree $T_G(R)$ into a spanning tree T' in G' without increasing the cost. For guiding this transformation we construct an auxiliary tree T'' by contracting all edges of $T_G(R)$ that are entirely in one Voronoi region. We consider T'' as a rooted tree with an arbitrary root z_r. Beginning with isolated terminals as T', each step of the transformation removes the path from one leaf of T'' to its parent and inserts an edge of G' into T'. Throughout the transformation the following invariant (†) holds: In each component of T', there is exactly one terminal that has not been removed from T''. In the beginning (†) holds trivially.
Each step of the transformation is performed as follows: Choose any leaf z_i of T'' such that all vertices $v_i \in N(z_i) \backslash \{z_i\}$ have at most one successor in T''. Notice that there is always such a z_i in T'', because the number of leaves is greater than the number of non-terminals with more than one successor. From z_i we move in the direction of the root until we reach a terminal z_l. The path from z_i to z_l in T'' is denoted by P''_i. The corresponding path in $T_G(R)$ is denoted by P_i. Now we look at the bases of the vertices on P_i. Let v_j be the last vertex on P_i whose base z_j is connected to z_i in T', and v_k the first whose base z_k is not connected to z_i in T'. The invariant (†) guarantees that such vertices v_j and v_k exist, because not all bases $z_i, \ldots, z_l$ of the vertices of P_i can belong to the same component of T', since z_i and z_l have not been removed from T''. We denote with P'_i the part of P_i between z_i and v_k. The length of P'_i is at least $d'(z_j, z_k)$, because $d'(z_j, z_k) \leq d(z_j, v_j) + c(v_j, v_k) \leq d(z_i, v_j) + c(v_j, v_k)$. Remove the subpath of P''_i beginning from z_i until a vertex in T'' with degree greater than 2 or z_l is reached. The edge (z_j, z_k) is inserted in T', so (†) remains valid.

After all terminals except z_r have been removed from T'', $r-1$ edges have been inserted into T' without creating a cycle, so T' is a spanning tree at the end.
We show now that each two paths P'_a and P'_b corresponding to terminals z_a and z_b are edge-disjoint (in the following simply denoted as disjoint). There are two cases:
I) z_a is a successor of z_b in T'' (or vice versa): If there are common edges in P'_a and P'_b, there must be common edges or (if they are contracted) at least common vertices in P''_a and P''_b. But the paths P''_a and P''_b are disjoint and have at most one vertex in common, namely z_b. Hence, common edges of P'_a and P'_b must lie entirely inside the Voronoi region of z_b and have been contracted in T'' into z_b. When z_a is chosen, neither z_a nor z_b has been removed from T''. Because of $(\dagger)$, at this time z_a and z_b are not connected in T'. So, for P_a the vertex v_j (as defined above) must be outside $N(z_b)$, and there is no edge in P'_a that is entirely inside $N(z_b)$. Therefore, the paths are disjoint.
II) z_a and z_b are successors of a terminal z_c in T'': Assume that z_a is chosen before z_b. There are disjoint paths in $T_G(R)$ from a vertex v_d to z_a, z_b and z_c ($v_d = z_c$ is possible). Let $z_d := base(v_d)$. Suppose that the paths P'_a and P'_b are not disjoint. So, they must both contain edges of the path between v_d and z_c. Thus, when z_a is chosen, z_d must be connected to z_a in T'. Because z_a has not been removed from T'', it follows from $(\dagger)$ that z_d was removed from T'' before. This is only possible if $v_d \neq z_d$. When z_d was chosen, v_d had only one successor in T''. Thus, the path between z_d and z_b has been removed even sooner. This means that z_b had to be chosen before z_a, a contradiction.
Since each edge of T' corresponds to a path in $T_G(R)$ with at least the same length and all these paths are disjoint, the spanning tree T' is not longer than $T_G(R)$, and the lemma follows.[1] □

This lemma can be extended to a test condition; for example, for any non-terminal v_i, the weight of such a spanning tree minus the length of its longest edge plus $d(v_i, z_{i,1}) + d(v_i, z_{i,2})$ is a lower bound for the weight of any Steiner minimal tree that contains v_i. The resulting test is very fast: The network G' can be determined without much extra work while computing the Voronoi regions, and a minimum spanning tree for it can be computed in time $O(m + r \log r)$.

For computing upper bounds in this context, we use a modified path heuristic with time $O(m + n \log n)$, which is described in Section 4.2. So, all these tests can be performed in time $O(m+n \log n)$; we call this combined test **VR** (standing for Voronoi Regions). With a heuristic solution available, all these tests can be easily extended to the case of equality of lower and upper bound. As intuition suggests, the VR test is most effective for sparse networks with relatively few terminals; in this sense, it is a nice complement to the alternative-based tests, which are often especially successful if the proportion of terminals to all vertices is high. Additionally, this test was the basis for the development of the strong PRUNE-heuristics, which are presented in Section 4.3.

3.4.2 Using Dual Ascent

The information provided by the algorithm DUAL-ASCENT (section 2.9.2), namely the lower bound with value $lower$ and the reduced costs can be used to design another bound-based reduction test. Here we use a simple, yet very helpful lemma, which we will exploit frequently later on:

Lemma 29 Let $G = (V, A, c)$ be a (directed) network (with a given set of terminals) and $\tilde{c} \leq c$. Let $lower'$ be a lower bound for the value of any (directed) Steiner tree in $G' = (V, A, c')$ with $c' := c - \tilde{c}$. For each $\tilde{x}$ representing a feasible Steiner tree for G with cost $c \cdot \tilde{x}$, it holds that $lower' + \tilde{c} \cdot \tilde{x} \leq c \cdot \tilde{x}$.

Proof: $c \cdot \tilde{x} = c' \cdot \tilde{x} + \tilde{c} \cdot \tilde{x} \geq lower' + \tilde{c} \cdot \tilde{x}$. □

[1] In [PV00], we gave an easier proof that used the equivalence between this lower bound and the lower bound produced by some primal dual algorithm for LP_C.

Now consider the reduced costs provided by DUAL-ASCENT as $\tilde{c}$: One can observe that the lower bound $lower'$ provided by DUAL-ASCENT in G' is the same as $lower$. So for any $\tilde{x}$ representing a feasible (directed) Steiner tree $\vec{T}$, $lower + \tilde{c} \cdot \tilde{x}$ represents a lower bound on the weight of $\vec{T}$.

This lemma can be used to compute lower bounds for the value of an optimal Steiner tree with additional constraints, for example, that the tree contains a certain non-terminal. The resulting tests are basically identical to the tests IRA and IRAe, which are introduced in [Dui93], using a somewhat more tedious argumentation.

Let v_k be a non-terminal, and $\vec{T}$ any optimal (directed) Steiner tree containing v_k, represented by $\tilde{x}$. The lower bound $\tilde{c} \cdot \tilde{x}$ on the weight of $\vec{T}$ minus $lower$ can be further estimated from below by the length of a shortest path (with respect to the costs $\tilde{c}$) from the root to v_k plus the length of an (arc-disjoint) shortest path from v_k to another terminal; and the last value can be again estimated from below by the distance of v_k to its nearest terminal, as described in Section 3.3.1. The non-terminal v_k can be eliminated if this lower bound exceeds a known upper bound. Similar tests can be developed for the elimination of edges and for the elimination of vertices after replacing incident edges (as in NTD$_k$). All these tests can be performed in time $O(m + n \log n)$ after a run of DUAL-ASCENT (and computation of an upper bound). With a heuristic solution available, these tests can be easily extended to the case of equality of lower and upper bound. We call this collection of tests **DA** (standing for Dual Ascent).

When dealing with the Steiner problem in undirected networks, it is a good idea to try different terminals as the root. Although the optimal value DLP_C is independent of this choice, the value of the lower bound provided by DUAL-ASCENT is not, and, much more important, different roots can lead to the elimination of different parts of the network, even if the value of the lower bound does not change. Trying a constant number (at most 10) of terminals as roots, we have substantially improved the effectiveness of this test. Notice also that each repetition profits from the reductions achieved by the previous ones.

The test DA is very effective, and usually it is fast empirically. But the time bound $O(a \cdot \min\{a, rn\})$ (resulting from DUAL-ASCENT) is, in comparison to the time $O(m + n \log n)$ of the other tests hitherto presented, somewhat unsatisfactory, especially because the other parts of the test can indeed be performed in time $O(m + n \log n)$.

One can try to achieve a better time bound by using a faster dual ascent algorithm, even if the provided lower bounds are worse: The tests described above use both the reduced costs and the lower bound, and a worse lower bound can be compensated to some degree by larger reduced costs.

One successful variant with running time $O(m + n \log n)$ uses the observation that it is possible to increase many dual variables around a terminal simultaneously.

Lemma 30 Choose a terminal $z_t \in R^{z_1}$. Define $d'(v_i) := \min\{d(v_i, z_t), d(z_1, z_t)\}$. For all Steiner cuts $\{\bar{W}, W\}$ set the dual variable $u_W := \max\{0, \min_{v_j \notin W}\{d'(v_j)\} - \max_{v_j \in W}\{d'(v_j)\}\}$. Then $\sum_{W,[v_a,v_b]\in\delta^-(W)} u_W = \max\{0, d'(v_a) - d'(v_b)\} \leq c_{ab}$ for all edges $[v_a, v_b] \in A$.

Proof: Let $v_1, v_2, \ldots, v_n$ be the vertices of V sorted by their distances to z_t in ascending order. Consider a Steiner cut $\{\bar{W}, W\}$. Obviously $u_W = \max\{0, d'(v_h) - d'(v_i)\}$ for $h = \min\{j \mid v_j \notin W\}$, $i = \max\{j \mid v_j \in W\}$. If there are two vertices v_h and v_i with $h < i, v_h \notin W$, and $v_i \in W$, then $u_W = 0$. So if $u_W > 0$, there must be a vertex v_k with $v_l \in W$ for all $l \leq k$ and $v_l \notin W$ for all $l > k$; so we can denote W by W_k: $W_k = \{v_1, \ldots, v_k\}$; $u_{W_k} = d'(v_{k+1}) - d'(v_k)$. For any edge $[v_a, v_b]$ we have:

$\sum_{W,[v_a,v_b]\in\delta^-(W)} u_W = \sum_{b\leq k<a} u_{W_k} =$
$\sum_{b\leq k<a}(d'(v_{k+1}) - d'(v_k)) = \max\{0, d'(v_a) - d'(v_b)\} =$

$\max\{0, \min\{d(v_a, z_t), d(z_1, z_t)\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} \leq$
$\max\{0, \min\{c_{ab} + d(v_b, z_t), d(z_1, z_t) + c_{ab}\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} = c_{ab}.$ □

It follows immediately that u is feasible for DLP_C. Since the dual variables u are not used explicitly in the reduction process, it is sufficient to work with the reduced costs and the calculated lower bound; so the updating process for one terminal can be performed very quickly, because we just need a shortest paths tree rooted at z_t that spans z_1. Then the reduced costs for an edge $[v_a, v_b]$ are decreased by $\max\{0, d'(v_a) - d'(v_b)\}$ and the lower bound is increased appropriately. After each such updating there may still be terminals that are not reachable from the root by edges of zero reduced cost, so the updating can be repeated with other terminals, but then with respect to the remaining reduced costs. We guide this calculation by the structure of a heuristic solution: The terminals are sorted according to non-decreasing distances from the root in this solution and considered one at a time.

Note that using this method, an edge can be visited by several terminals. To limit the effort, we simply abort the calculation of a shortest paths tree if it reaches a vertex that has already been visited by a constant number of terminals (e.g., 5). This leads to a worst-case running time for the calculation of a lower bound and reduced costs of $O(m + n \log n)$. The other operations of the test can be performed in the same time, as previously described. To construct a heuristic solution, we use a heuristic described in Section 4.2, which has the same running time. So the whole test can be performed in total time $O(m + n \log n)$. We call this test **LDA** (Limited Dual Ascent). Despite its low running time, it is fairly effective, especially if the proportion of terminals to all vertices is not very high.

3.4.3 Using the Row Generation Strategy

The modification above aimed at making the reduction technique based on reduced costs faster. A legitimate question is if it is possible to make that technique stronger. For this, we use the row generation method described in Section 2.9.2.

Every iteration of the row generation method provides a dual feasible solution for LP_C (or $LP_{C'}$) and appropriate reduced costs. Using this information, the same reduction techniques as described in Section 3.4.2 can be used. The only enhancement here is that edges are allowed to be deleted only in one direction during the row generation process (remember that the relaxations LP_C and $LP_{C'}$ use directed networks). This can amplify the effect of subsequent reductions considerably. In the linear program itself, the deletion of edges is realized by fixing the corresponding variables to zero.

In many cases the mentioned reductions during the row generation make further alternative-based reductions possible. But it would be a bad idea to delay these reductions until the row generation terminates, because they could possibly accelerate the computation and raise the optimal value of the relaxation. On the other hand, it would be problematic to abort the row generation, do the alternative-based reductions and then start it again, because the constraints generated in the meantime could not be used anymore, at least not directly. Our approach for dealing with this problem is to perform alternative-based reductions in an undirected copy of the current directed instance (which is not necessarily bidirected). After that, the reduced undirected instance is translated back into a directed instance, with the performed reductions translated into fixing of variables. We call the whole reduction method **RG** (for Row Generation).

The row generation approach can be exploited for even stronger reductions in combination with the extended reduction techniques that will be described in the next sections.

Note that using a lower bound and reduced costs produced by an LP-solver in the context of reduction techniques is a delicate thing. As the LP-solver works with floating point arithmetic and gives hardly any guarantee for the quality of the returned solution, the output of the LP-solver cannot

be used directly. One reliable and very efficient solution is the use of fix point integer arithmetic. The floating point numbers are scaled by some factor and rounded to integers.

In this process, we distinguish between the primal and the dual variables. The primal variables are less critical, as they are only used for the computation of new constraints, e.g., by minimum cut computations. Note that the preflow-push algorithm used only works reliably using exact arithmetic. As we always round up while transforming the primal variables, we make it less probable, yet not impossible, to find an already satisfied cut constraint that has a capacity less then one according to the rounded capacity values. Also, it is possible that we miss some violated cut constraints with a capacity very close to one. Neither of the two situations is really harmful. As primal variables are always between zero and one, we can choose a very high factor for the scaling.

The dual side is more interesting. The basic idea is to round the dual variables to fixed point numbers and compute a valid lower bound and reduced costs in exact arithmetic. A minor issue is that we have to store the integer constraint matrix A of the linear program. The problem is that the dual variables may not be feasible any more. Here we can use a nice extension of Lemma 29.

Lemma 31 For any linear program

$$\min\{c \cdot x \mid Ax \geq b;\ 1 \geq x \geq 0\} = \max\{b \cdot \lambda - 1 \cdot \mu \mid A^T\lambda - \mu \leq c;\ \lambda \geq 0;\ \mu \geq 0\},$$

any (feasible or infeasible) dual values $\bar{\lambda} \geq 0,\ \bar{\mu} \geq 0$ and any feasible primal values $\tilde{x}$ it holds that:

$$b \cdot \bar{\lambda} - 1 \cdot \bar{\mu} + (c + \bar{\mu} - A^T\bar{\lambda}) \cdot \tilde{x} \leq c \cdot \tilde{x}.$$

Proof: Let x^+ be an optimal solution for the linear program $\min\{c^+ \cdot x \mid Ax \geq b;\ x \geq 0\}$ with $c^+ = A^T\bar{\lambda}$. Note that $\tilde{x}$ is feasible for this linear program, and $\bar{\lambda}$ is feasible for the dual of the program. It follows that $b \cdot \bar{\lambda} \leq c^+ \cdot x^+ \leq c^+ \cdot \tilde{x}$. As $\tilde{x} \leq 1$, it follows that $\bar{\mu} \cdot \tilde{x} \leq 1 \cdot \bar{\mu}$. Both inequalities together prove the claim. □

Corollary 31.1 If $\bar{\mu} \geq A^T\bar{\lambda} - c$, it follows that for any primal feasible $\tilde{x}$, $b \cdot \bar{\lambda} - 1 \cdot \bar{\mu}$ is a lower bound for $c \cdot \tilde{x}$.

To use the inexact dual variables in the context of bound-based reductions, we do the following:

1. We round the dual variables to fixed point numbers $\bar{\lambda}$. We choose the scaling factor such that the current difference between upper and lower bound corresponds to a fixed point number $rest$ that uses the nearly full bit length of the computer's integers. We can ensure that all numbers that are computed in the bound-based reductions are not greater than $2\ rest + 1$. On the one hand, this presents overflows. On the other, the accuracy is increased in the row generation iterations automatically, as the difference between upper and lower bound decreases.

2. We set any negative $\bar{\lambda}_i$ to zero.

3. We try a "test and repair" strategy: If $A^T\bar{\lambda} \leq c$ is violated for some arc $[v_i, v_j]$, we try to shift the dual variables associated to constraints containing x_{ij} a little so that the violation gets smaller. This done in a heuristical way.

4. Using exact arithmetic, we compute $\bar{\mu} := \max\{0, A^T\bar{\lambda} - c\}$. From Corollary 31.1 it follows that $lower := b \cdot \bar{\lambda} - 1 \cdot \bar{\mu}$ is an exact lower bound for the value of an optimal Steiner tree.

5. We compute reduced costs $\tilde{c} := c + \bar{\mu} - A^T\bar{\lambda}$. Note that $\tilde{c} \geq 0$.

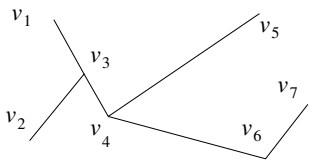
6. Now, we can perform bound-based reductions as described in the previous section with $lower$ and $\tilde{c}$: $lower + \tilde{c} \cdot \tilde{x}$ is a lower bound for the value of a Steiner tree represented by $\tilde{x}$.

3.5 Extended Reduction Techniques: Combining Alternative- and Bound-based Approaches

The classical reduction tests just consider single vertices or edges. Recent and more sophisticated tests extend the scope of inspection to more general patterns. In the next sections, we present such an extended reduction test, which generalizes various tests from the literature. We use the new approach of combining alternative- and bound-based methods, which substantially improves the impact of the tests. We also present several algorithmic contributions. The experimental results show a large improvement over previous methods using the idea of extension, leading to a drastic speed-up in the optimal solution process and the solution of several previously unsolved benchmark instances.

3.5.1 Additional Definitions for Extended Reduction Techniques

For every tree T in G, we denote by $V(T)$ the vertices of T, by $L(T)$ the leaves of T, and by $c(T)$ the sum of the costs of edges in T. Let T' be a subtree of T. The **linking set** between T and T' is the set of vertices $v_i \in V(T')$ such that there is a fundamental path from v_i to a leaf of T not containing any edge of T'. Note that the paths can have zero length and if a leaf of T' is also a leaf of T it will be in the linking set. If the linking set between T and T' is equal to $L(T')$, T' is said to be **peripherally contained** in T (Figure 3.5). This means that for every leave v_j of T the fundamental path connecting v_j to T' ends in a leaf of T'. A set $L' \subseteq V(T), |L'| > 1$, induces a subtree $T_{L'}$ of T containing for every two vertices $v_i, v_j \in L'$ the fundamental path between v_i and v_j in T. We define L' to be a **pruning set** if L' contains the linking set between T and $T_{L'}$.



For the depicted tree T, let T' be the subtree of T after removing the edges (v_4, v_6) and (v_6, v_7). The linking set between T and T' is $\{v_1, v_2, v_4, v_5\}$, and therefore T' is not peripherally contained in T. But if we add (v_4, v_6) to T', it is.

Figure 3.5: Depiction of some central notions for extended reductions.

3.5.2 Extending Reduction Tests

The classical reduction tests for the Steiner problem inspect only simple patterns (a single vertex or a single edge). There have been some approaches in the literature for extending the scope of inspection [Dui00, UdAR99, Win95]. The following function $EXTENDED\text{-}TEST$ describes in pseudocode a general framework for many of these approaches. The argument of $EXTENDED\text{-}TEST$ is a tree T that is expanded recursively. For example, to eliminate an edge e, T is initialized with e. The function returns 1 if the test is successful, i.e., it is established that there is an optimal Steiner tree that does not peripherally contain T.

In the pseudocode, the function $RULE\text{-}OUT(T, L)$ contains the specific test conditions (see Section 3.5.3): $RULE\text{-}OUT(T, L)$ returns 1 if it is established that T is not contained with linking set L in at least one optimal Steiner tree. The function $TRUNCATE$ checks some criterion to truncate the recursive expansion, and $PROMISING$ tries to identify promising candidates for expansion.

```
EXTENDED-TEST(T) :
    (returns 1 only if T is not contained peripherally in an optimal Steiner tree)
1     if RULE-OUT(T, L(T)) :
2         return 1            (test successful)
3     if TRUNCATE(T) :
4         return 0            (test truncated)
5     forall leaves v_i of T :
6         if v_i ∉ R and PROMISING(v_i) :
7           success := 1
8           forall nonempty extension ⊆ {(v_i, v_j) :
                    not RULE-OUT(T ∪ {(v_i, v_j)}, L(T) ∪ {v_j})} :
9               if not EXTENDED-TEST(T ∪ extension) :
10                success := 0
11          if success :
12              return 1            (no acceptable extension at v_i)
13    return 0          (in all inspected cases, there was an acceptable extension)
```

Assuming that $RULE\text{-}OUT$ is correct, the correctness of $EXTENDED\text{-}TEST$ can be proven easily by induction, using the fact that if T' is a subtree of an optimal Steiner tree T^* and contains no inner terminals, all leaves of T' are connected to some terminal by paths in $T^* \setminus T'$.

Clearly the decisive factor for the performance of this algorithm is the realization of the functions $RULE\text{-}OUT$, $TRUNCATE$ and $PROMISING$.

Using this framework, previous extension approaches can be outlined easily:

- In [Win95] the idea of expansion was introduced for the rectilinear Steiner problem.
- In [UdAR99] this idea was adopted to the Steiner problem in networks. This variant of the test tries to replace vertices with degree three; if this is successful, the newly introduced edges are tested again with an expansion test. The expansion is performed only if there is a single possible extension at a vertex, thus eliminating the need for backtracking.
- In [Dui00] backtracking was explicitly introduced, together with a number of new test conditions to rule out subnetworks, dominating those mentioned in [UdAR99].
- In Section 3.3.4, we used a different test that tries to eliminate edges. Expansion is performed only if there is at most one possible extension (thus inspecting a path) and only if the elimination of one edge implies the elimination of all edges of the path.

All previous approaches use only alternative-based methods. We present an expansion test that explicitly combines the alternative-based and bound-based methods. This combination is far more effective than previous tests, because the two approaches have complementary strengths. Intuitively speaking, the alternative-based method is especially effective if there are terminals in the vicinity of the currently inspected subgraph T, because it uses the bottleneck Steiner distances. On the other hand, the bound-based method is especially effective if there are no close terminals, because it uses the distances (with respect to reduced costs) to terminals. Furthermore, for the expansion test to be successful, usually many possible extensions must be considered and it is often the case that not all of them can be ruled out using exclusively the alternative- or the bound-based methods, whereas an explicit combination of both methods can do the job.

Although the pseudocode of $EXTENDED\text{-}TEST$ is simple, designing an efficient and effective implementation requires many algorithmic ideas and has to be done carefully, taking the interaction between different actions into account, which is highly nontrivial. Since writing down many pages of pseudocode would be less instructive, we prefer to explain the main building blocks. In the following, we first describe the test conditions for ruling out trees (the function $RULE\text{-}OUT$), using the results of [Dui00] and introducing new ideas. Then we explain the criteria used for truncation and choice of the leaves for expansion (the functions $TRUNCATE$ and $PROMISING$). Finally, we will address some implementation issues, particularly data structures for querying different types of distances.

3.5.3 Test Conditions

For the following test conditions we always consider a tree T where terminals may appear only as leaves of the tree, i.e., $V(T) \cap R \subseteq L(T)$. A very general formulation of the alternative-based test condition is the following:

Lemma 32 Consider a pruning set L' for T. If $c(T_{L'})$ is larger than the cost of a Steiner tree T' in $G' = (V, V \times V, s)$ with L' as terminals, then there is an optimal Steiner tree that does not peripherally contain T. This test can be strengthened to the case of equality if there is a vertex v in $T_{L'}$ that is not in any of the paths used for defining the s-values of the edges of T'.

Proof: Assume that T is peripherally contained in an (optimal) Steiner tree T^* in G. As L' is a pruning set for T and the leaves of T are a pruning set for T^*, L' is also a pruning set for T^*. It follows that after removing the edges of $T_{L'}$ from T^*, each of the remaining subtrees contains one vertex of L'. The plan is to reconnect these subtrees to a new Steiner tree by replacing each necessary edge of T' with a path in G of no larger cost. Consider the forest F consisting of these subtrees together with the remaining nodes of T' (i.e., nodes that are not in any of these subtrees). Merge all vertices of T' that are in one component of F, breaking emerging cycles by deleting an arbitrary edge of each cycle. This operation does not increase the cost of T'. Now, each component C_i of F corresponds to one vertex t_i of T'. We will ensure this invariant during the whole process of updating T' and F.

Choose a shortest edge (t_i, t_j) of T'. Let P_{ij} be a path of bottleneck Steiner distance s_{ij} between v_i and v_j, vertices of V corresponding to t_i and t_j (before merging). Let P_{kl} be a subpath of P_{ij} in which only the endpoints v_k and v_l are in $R \cup \{v_i, v_j\}$, and v_k and v_l are in different components C_k and C_l of F. Remove an arbitrary edge on the fundamental path in T' between t_k and t_l and merge t_k and t_l in T'. Finally, connect C_k and C_l in F by adding the necessary edges from P_{kl}. The sum of the costs of these edges is not larger than s_{ij}. Because (t_i, t_j) was a shortest edge of T', the added cost in F is also not larger than the cost of the edge that was removed from T'.

Repeating this procedure leads to a new network that connects all terminals of G and has cost at most $c(T^*) - c(T_{L'}) + c(T')$. If $c(T') < c(T_{L'})$ or $c(T') = c(T_{L'})$ and there is at least one vertex in $V(T_{L'})$ that is not in the new tree (because it was not in any of the paths that were used for defining the s-values of the edges in T'), we have a Steiner tree of cost not larger then $c(T^*)$ that does not peripherally contain T.[2] □

A typical choice for L' is $L(T)$, often with some leaves replaced by vertices added to T in the first steps of the expansion. If computing an optimal Steiner tree T' is considered too expensive, the cost of a minimum spanning tree for L' with respect to s can be used as a valid upper bound.

A relaxed test condition compares bottleneck Steiner distances with tree bottlenecks:

[2] There are proofs in [Dui00, HRW92] for similar (but weaker) conditions, but they are not complete.

Lemma 33 If $s_{ij} < t_{ij}$ for any $v_i, v_j \in T$, there is an optimal Steiner tree that does not peripherally contain T. Again, the test can be strengthened to the case of equality if a path corresponding to s_{ij} does not contain a tree bottleneck of T between v_i and v_j.

Proof: Consider v_i, v_j and all key nodes on the fundamental path P_{ij} between v_i and v_j in T as the pruning set L' in the previous lemma. The induced subtree $T_{L'}$ is the path P_{ij} itself. Removing a tree bottleneck from P_{ij}, inserting an edge (v_i, v_j) of cost s_{ij} and substituting the c-values for the other edges with the (not larger) s-values leads to a Steiner tree for L' in G' with cost at most $c(P_{ij}) + s_{ij} - t_{ij}$. □

The bound-based test condition uses a dual feasible solution for LP_C of value $lower'$ and corresponding reduced costs c'' (with resulting distances d''):

Lemma 34 Let $\{l_1, l_2, \ldots, l_k\} = L(T)$ be the leaves of T. Then $lower_{constrained} := lower' + min_i\{d''(z_1, l_i) + c''(\vec{T}_i) + \sum_{j \neq i} min_{z_p \in R^{z_1}} d''(l_j, z_p)\}$ defines a lower bound for the cost of any Steiner tree with the additional constraint that it peripherally contains T, where $\vec{T}_i$ denotes the directed version of T when rooted at l_i.

Proof: If T is peripherally contained in an optimal Steiner tree T^*, then there is a path in T^* from the root terminal z_1 to a leaf l_i of T. After rooting T from l_i, each (possibly single-vertex) subtree of T^* corresponding to other leaves l_j contains a terminal. Now the lemma follows directly using Lemma 29. □

In the context of replacement of edges, one can use the following lemma.

Lemma 35 Let e_1 and e_2 be two edges of T in a reduced network. If both edges originate from a common edge e_3 by a series of replacements, then no optimal Steiner tree for the reduced network that corresponds to an optimal Steiner tree in the unreduced network contains T.

Proof: Assume that there is an optimal Steiner tree T^* for the reduced network containing both e_1 and e_2. Back-substituting the edges of T^* leads to a solution in the original network in which e_3 is used twice. This means that the solution value in the unreduced and consequently in the reduced network can be decreased by $c(e_3)$, which contradicts the optimality of T^*. □

The conditions above cover the calls $RULE\text{-}OUT(T, L)$ with $L = L(T)$. In case other vertices than the leaves need to be considered in the linking set (as in line 8 of the pseudocode), one can easily establish that all lemmas above remain valid if we treat all vertices of L as leaves.

3.5.4 Criteria for Expansion and Truncation

The basic truncation criterion is the number of backtracking steps, where there is an obvious tradeoff between the running time and the effectiveness of the test. A typical number of backtracking steps in our implementations is five.

Additionally, there are other criteria that guide and limit the expansion:

1. If a leaf is a terminal, we cannot easily expand over this leaf, because we cannot assume anymore that an optimal Steiner tree must connect this leaf to a terminal by edges not in the current tree. However, if all leaves are terminals (a situation in which no expansion is possible for the original test), we know that at least one leaf is connected by an edge-disjoint path to another terminal (as long as not all terminals are spanned by the current tree). This can be built into the test by another level of backtracking and some modifications of the test conditions. But we do

not describe the modifications in detail, because the additional cost did not pay off in terms of significantly more reductions.

2. If the degree d of a leaf is large, considering all $2^{(d-1)} - 1$ possible extensions would be too costly and the desired outcome, namely that we can rule out all of these extended subtrees, is less likely. Therefore, we limit the degree of possible candidates for expansion by a small constant, e.g. 8.

3. It has turned out that a depth-first realization of backtracking is quite successful. In each step, we consider only those leaves for expansion that have maximum depth in T when rooted at the starting point. In this way, the bookkeeping of the inspected subtrees becomes much easier and the whole procedure can be implemented without recursion. A similar idea was already mentioned (but not explicitly used) in [Dui00].

4. In case we do not choose the depth-first strategy, a tree T could be inspected more than once. As an example, consider a tree T resulting from an expansion of T' at leaf v_i and then at v_j. If T cannot be ruled out, it is possible that we return to T', expand it at v_j and then at v_i, arriving at T again. This problem can be avoided by using a (hashing-based) dictionary.

3.5.5 Implementation Issues

Precomputing (Steiner) Distances: A crucial issue for the implementation of the test is the calculation of bottleneck Steiner distances as defined in Section 3.2. An exact calculation of all s_{ij} needs time $O(n(m + n \log n))$ [Dui00] and space $\Theta(n^2)$, which would make the test impractical even for mid-size instances. So we need a good approximation of these distances and some appropriate data structures for retrieving them. Building upon a result of Mehlhorn [Meh88], Duin [Dui93] gave a nice suggestion for the approximation of the bottleneck Steiner distances, which needs preprocessing time $O(m + n \log n)$ and a small running time for each query. As described in Section 3.3.1 the query time can even made constant if all necessary queries are known in advance. Although the resulting approximate values $\hat{s}_{ij}$ produce quite satisfactory results for the original s-test (PT_m in Section 3.3.1), for the extended test the results are unfortunately much worse than with the exact values. But we observed that $\tilde{s} = \min\{\hat{s}, d\}$ is almost always equal to the exact s-values, and therefore can be used in the extended test as well. Still there remains the problem of computing the d-values: For each vertex v_i we compute and store in a neighbor list the distances to a constant number (e.g. 50) of nearest vertices. But we consider only vertices v_j with $d_{ij} < \hat{s}_{ij}$. This is justified by the observation that in case $d_{ij} \geq \hat{s}_{ij}$, for all descendants v_k of v_j in the shortest paths tree with the root v_i there is a path P_{ik} of bottleneck Steiner distance s_{ik} containing at least one terminal, and in such cases s_{ik} is usually quite well approximated by $\hat{s}_{ik}$.

Now, we use different methods for different variants of the test:

1. If we only replace vertices in an expansion test, the d- and s-values do not increase and we can use the precomputed neighbor lists during the whole test using binary search.

2. If we limit the number of (backtracking) steps, then we can confine the set of all possible queries in advance. When a vertex is considered for replacement by the expansion test, we first compute the set of possibly visited neighbors (adjacent vertices, and vertices adjacent to them, and so on, up to the limited depth). Then we compute a distance matrix for this set according to the $\tilde{s}$-values. Using this matrix, each query can be answered in constant time.

3. If we also want to delete edges, we have to store for each vertex in which neighbor list computations it was used. When an edge is deleted, we can redo the computation of the $\hat{s}$-value (or at least those parts that may have changed due to the edge deletion) and restore the affected neighbor lists. This can even be improved by a lazy calculation of the neighbor lists.

Tree Bottlenecks: The tree bottleneck test of Lemma 33 can be very helpful, because every distance between tree vertices calculated for a minimum spanning tree or a Steiner minimal tree computation can be tested against the tree bottleneck; and in many of the cases where a tree can be ruled out, already an intermediate bottleneck test can rule out this tree, leading to a shortcut in the computation. This is especially the case if there are long chains of nodes with degree two in the tree. We promote the building of such chains while choosing a leaf for extension: We first check whether there is a leaf at which the tree can be expanded by only one edge. In this case we immediately perform this expansion, without creating a new key node and without the need of backtracking through all possible combinations of expansion edges.

The tree bottleneck test can be sped up by storing for each node of the tree the length of a tree bottleneck on the path to the starting vertex. For each two nodes v_i and v_j in the tree, the maximum of these values gives an upper bound for the actual tree bottleneck length t_{ij}. Only if this upper bound is greater than the (approximated) bottleneck Steiner distance, an exact tree bottleneck computation is performed.

Computations for the Bound-Based Tests: An efficient method for generating the dual feasible solution needed for the bound-based test of Lemma 34 is the DUAL-ASCENT algorithm described in Section 3.4.2. We improve the test by calculating a lower bound and reduced costs for different roots. Although the optimal value of the directed cut relaxation LP_C does not change with the choice of the root, this is not true concerning the value of the dual feasible solution generated by DUAL-ASCENT and, more importantly, the resulting reduced costs can have significantly different patterns, leading to a greater potential for reductions.

Even more reductions can be achieved by using stronger lower bounds, as computed with a row generating algorithm, see Section 3.4.3. Concerning the tests for the replacement of vertices, we use only the result of the final iteration, which provides an optimal dual solution of the underlying linear relaxation. The dual feasible solutions of the intermediate iterations are used only for the tests dealing with the deletion of edges, because the positive effect of the replacement of a vertex (see the NTD$_k$ test in Section 3.3.2) cannot be translated easily into linear programs.

Replacement History: Our program package can transform a tree in a reduced network back into a tree in the original instance. For this purpose, we assign a unique ID number to each edge. When a vertex is replaced, we store for each newly inserted edge a triple with the new ID and the two old IDs of the replaced edges. We use this information to implement the test described in Lemma 35. First we do some preprocessing, determining for each ID the edges it possibly originates from (here called ancestors); this can be done in time and space linear in the number of IDs. Later, a test for a conflict between two edges (i.e., they originate from the same edge) can be performed by marking the IDs of the ancestors of one edge and then checking the IDs of ancestors of the other edge; so each such test can be done in time linear in the number of ancestors. We perform this test each time the current tree T is to be extended over a leaf v_i (with (v_k, v_i) in T) by an edge (v_i, v_j). Then we check for a conflict between (v_i, v_j) and (v_k, v_i). This procedure implements an idea briefly mentioned in [Dui00], where

a coloring scheme was suggested for a similar purpose. Our scheme has the advantage that it may even discover conflicts in situations where an edge is the result of a series of replacements.

3.5.6 Variants of the Test

A general principle for the application of reduction tests is to perform the faster tests first so that the stronger (and more expensive) tests are applied to (we hope) sufficiently reduced graphs. In the present context, different design decisions (e.g., trying to delete edges or replace vertices) lead to different consequences for an appropriate implementation and quite different versions of the test, some faster and some stronger.

We have implemented four versions of expansion tests and integrated them into the reduction process. Some details of the corresponding implementations were already given in Section 3.5.5.

1. For a fast preprocessing we use the linear time expansion test that eliminates paths, as described in Section 3.3.4.
2. A stronger variant tries to replace vertices, but only expands at leaves that are the most number of edges away from the starting vertex.
3. Even stronger but more time-consuming is a version that performs full backtracking.
4. The most time-consuming variant tries to eliminate edges.

3.6 Partitioning as a Reduction Technique

Partitioning is a basic principle in many algorithms. Although the basic ideas behind the approach presented in the following are relatively simple, the way we exploit them is new, and our approach could also be useful for other problems.

There are several reasons that motivate the use of partitioning in the present or similar contexts:

Efficiency: For any algorithm with superlinear running time, a suitable partitioning of the instance leads to a superlinear speedup. Note that because exact algorithms in this context use very time-consuming components (like LP-solvers) and are even exponential in the worst case, the speedup for solving subproblems can be highly superlinear.

Effectiveness: Sometimes, a method (a component of an exact algorithm) works well on some group of instances; but it fails on larger instances of the same type. Methods that are based on LP-relaxations of the problem are good examples, because any LP-relaxation of polynomial size is bound to have some (integrality) errors in this context. In larger instances, such errors can accumulate and become more and more relevant. Partitioning can help against this accumulation of errors, as we will show in Section 3.6.3.

Implementation: A reasonable partitioning offers a direct path to a distributed implementation, because different (reasonably independent) subproblems can be processed on different processors.

However, for applying the idea of partitioning to problems like the Steiner problem, classical approaches are not very helpful. Divide-and-conquer techniques are not generally applicable, because one usually cannot find independent subproblems. Dynamic programming techniques can indeed be applied, but these techniques (at least in their classical form) do not lead to empirically efficient algorithms.

In this section, we introduce the new approach of using partitioning to design reduction methods. We partition instances by finding (small) terminal separators, i.e., after removing these terminals from the network it is no longer connected. This allows us to keep the dependence between the resulting subinstances manageable.

3.6.1 Partitioning on the Basis of Terminal Separators

Although one cannot assume that a typical instance of the Steiner problem has small terminal separators, the situation often changes in the process of solving an instance, as described in the following.

Reduced Instances

There are several reduction methods which, when successful, tend to transform instances without useful terminal separators into instances with them.

Figure 3.6 shows a VLSI-instance from the library [SteinLib]. The terminals (black squares) are to be connected on the grid. Note that there are holes in the grid (corresponding to obstacles on the chip), so such instances are not geometric (rectilinear). In this figure, one does not detect any useful terminal separator. But the situation changes after applying some reduction methods: Figure 3.7 shows the reduced instance, which is produced after a couple of seconds: the black edges are chosen (and contracted); the grey edges remain as the reduced instance; this reduced instance is redrawn more compactly in Figure 3.8. Note that the visualization used is not geometric; edges which appear relatively long may actually have relatively small cost. (Our algorithm is a pure graph algorithm, which does not use the coordinates of the points anyway.) In this reduced instance, one easily detects many small terminal separators and corresponding components.

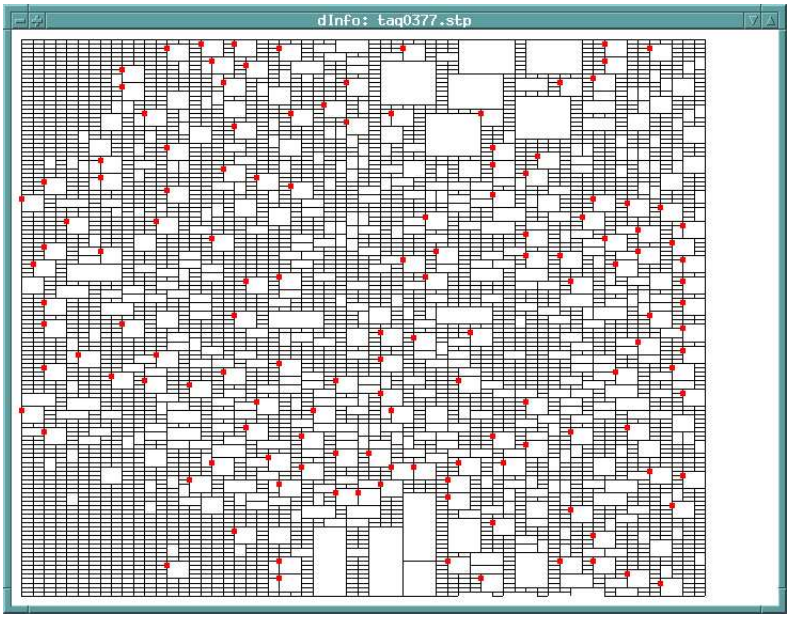


Figure 3.6: VLSI-instance taq0377 ($|V| = 6836,\ |E| = 11715,\ |R| = 136$).

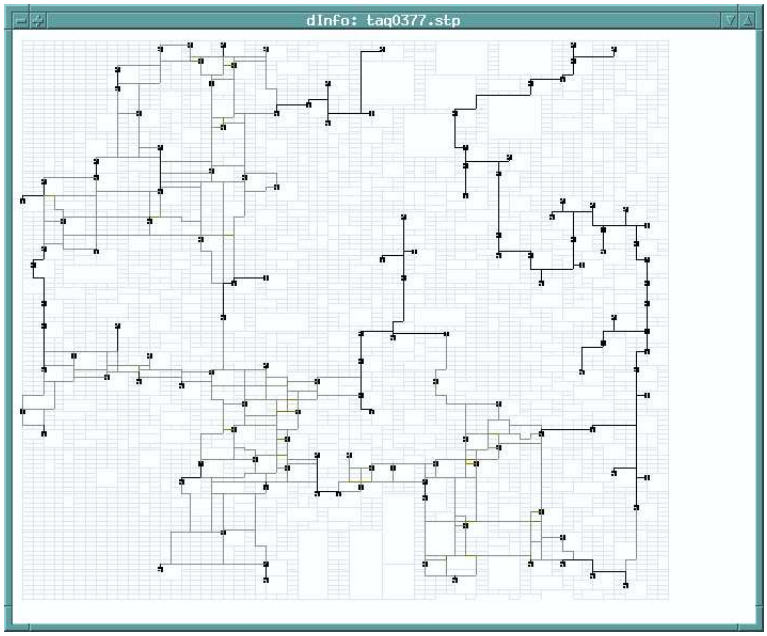


Figure 3.7: taq0377, reduced ($|V| = 193,\ |E| = 312,\ |R| = 67$).

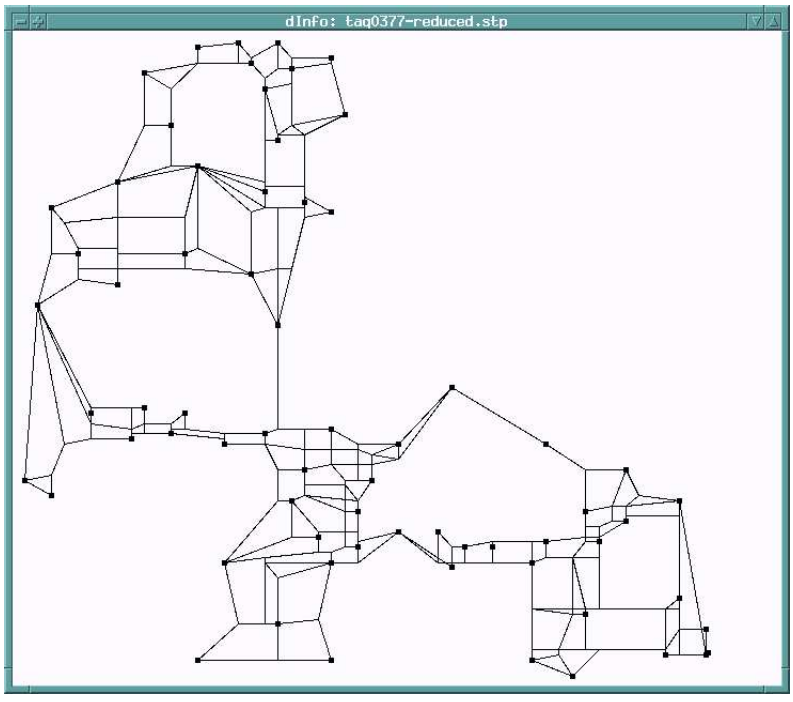


Figure 3.8: taq0377, reduced, redrawn ($|V| = 193,\ |E| = 312,\ |R| = 67$).

Geometric Steiner Problems

For geometric Steiner problems, an approach based on full Steiner trees has been empirically successful [WWZ00]: In geometric Steiner problems, a set of points (in the plane) is to be connected at minimum cost according to some geometric distance metric. The resulting interconnection, a Steiner minimal tree (SMT), can be decomposed into FSTs (full Steiner trees) by splitting its inner terminals. The FST-approach consists of two phases. In the first phase, the FST generation phase, a set of FSTs is generated that is guaranteed to contain an SMT. In the second phase, the FST concatenation phase, one chooses a subset of the generated FSTs whose concatenation yields an SMT. Although there are point sets that give rise to an exponential number of FSTs in the first phase, usually only a linear number (in $|R|$) of FSTs are generated, and empirically the bottleneck of this approach has usually been the second phase, where methods like backtracking or dynamic programming have originally been used. A breakthrough occured when Warme [War98] used the fact that FST concatenation can be reduced to finding a minimum spanning tree in a hypergraph whose vertices are the terminals and whose (hyper-)edges correspond to the generated FSTs. Although the minimum spanning tree in hypergraph (MSTH) problem is $\mathcal{NP}$-hard, a branch-and-cut approach based on the linear relaxation of an integer programming formulation of this problem has been empirically successful.

Note that by building the union of (the edge sets of) the FSTs generated in the first phase, we get a normal graph and the FST-concatenation problem is reduced to solving the classical Steiner problem in this graph. In Section 5.4.1, we report that our general graph algorithm often performs the second phase significantly faster than the MSTH-based method, especially for the more time-consuming instances. This is mainly due to the use of reduction techniques, where the methods presented here also play an important role, because usually the instances have many useful terminal separators with corresponding small components already at the beginning of the second phase, and this is even amplified after application of other reduction methods.

Combination of Steiner Trees

During the solution process, our program generates many heuristical solutions (Steiner trees). It would not be the best idea to just keep the best of them and throw the others away, because one could possibly combine parts of them to construct a new, even better solution. A simple but effective method is to build the instance corresponding to the union of the edges of different Steiner trees and find a (heuristical or even exact) solution in that instance (see Section 4.5). Such instances often have small terminal separators and the methods described here can be applied.

3.6.2 Finding Terminal Separators

It is well known that the vertex connectivity problem can be solved by network flow techniques in the so-called split graph, which is generated by splitting each vertex into two vertices and connecting them by edges of low capacity; original edges have high (infinite) capacity. In this way, k-connectedness (finding a vertex separator of size less than k or verifying that no such separator exists) can be decided in time $O(\min\{kmn, (k^3 + n)m\})$ [HRG00] (this bound comes from a combination of augmenting path and preflow-push methods). In case of undirected graphs (as in the present application), the job can be done in a sparse graph with $O(kn)$ edges, which can be constructed in time $O(m)$ [NI92], so m can be replaced in the above bound by kn.

However, the application here is less general: we search for vertex separators consisting of terminals only, so only terminals need to be split. Besides, we are interested in only small separators, where k is a very small constant (usually less than 5, say at most 10), so we can concentrate on the

(easier) augmenting flow methods. More importantly, we are not searching for a single separator of minimum size, but for many separators of small (not necessarily minimum) size. These observations have lead to the following implementation: we build the (modified) split graph (as described above), fix a random terminal as source, and try different terminals as sinks, each time solving a minimum cut problem using augmenting path methods. In this way, up to $\Theta(r)$ terminal separators can be found in time $O(rm)$. We accelerate the process by using some heuristics. A simple observation is that vertices that are reachable from the source by paths of non-terminals need not be considered as sinks. Similar arguments can be used to heuristically discard vertices that are reachable from already considered sinks by paths of non-terminals.

Empirically, this method is quite effective (it finds enough terminal separators if they do exist) and reasonably fast, so a more stringent method (e.g., trying to find all separators of at most a given size) would not pay off. Note that the running time is within the bound given above for the k-connectedness problem, which is mainly the time for finding a single vertex separator.

3.6.3 Reduction Methods

In this section, we describe two methods that exploit a terminal separator $S \subset R$ and the corresponding partitioning to reduce a given instance.

Case Differentiation

warm-up ($|S| = 1$): The case $|S| = 1$ corresponds to biconnected components and articulation points (i.e., if one of the articulation points is removed from the network, it is no longer connected) and biconnected components, which can be found in linear time $O(m)$. It is well known [HRW92] that the subinstances corresponding to the biconnected components can be solved independently. An example can be found in Figure 3.18: There is an articulation point in the middle; the upper and the lower components can be solved independently and the small component in the middle can be discarded (see Lemma 42 in Section 5.3.2).

base case ($|S| = 2$): The case $|S| = 2$ corresponds to separation pairs (and triconnected components). All (non-trivial) triconnected components can be found in linear time $O(m)$ [HT73]. Consider Figure 3.9, where a separator S of size 2 and the corresponding components G_1 and G_2 are shown (black edges for G_2). Note that the two subinstances are no longer independent. Now, for any Steiner minimal tree T, two cases are possible :

I) The terminals in S are connected by T inside G_2. A corresponding Steiner tree can be found by solving the subinstance corresponding to G_2.

II) The terminals in S are connected by T inside G_1. Now there are two subtrees of T inside S, and we do not know in advance how the terminals of G_2 are divided between them. But one can observe that the problem can be solved by merging the terminals in S and solving the resulting subinstance.

Since we do not know T in advance, for a direct solution we must also consider both cases for its complement G_1. But if G_2 is relatively small, the solution of the complementary subinstance can be almost as time-consuming as the solution of the original instance, meaning that not much is gained (or time may even be lost, because now we have to solve it twice). A classical approach would search for components of almost equal size, but we choose a different approach. The idea is to solve only the small component twice, and then take edges that are common to

both solutions and discard edges that are included in neither. The result is shown in Figure 3.10, where only one edge from G_2 is left undecided (the other edges can be either contracted (black) or deleted (light grey)). In Figure 3.11, the reduced instance is redrawn; as one sees, the processed component has almost disappeared.

general case ($|S| = k$): As described in Section 3.6.2, we can find up to $\Theta(r)$ separators of size at most k in time $O(krm)$. The basic approach is the same as for the case $|S| = 2$; but a larger number of cases must be considered now. We put each subset of terminals in S that are connected by one subtree of T in G_1 into one group. There can be $i = 1, \ldots, k$ such groups. For each i, we must count the number of ways of partitioning a set of k elements into i non-empty subsets, which is a Stirling number of the second kind $\{{k \atop i}\}$. So there are $\sum_{i=1}^{k} \{{k \atop i}\}$ cases. Table 1 contains the concrete numbers for small k.

k	2	3	4	5	6	7	8
cases	2	5	15	52	203	877	4140

Table 1: Possible cases for a terminal separator of size k

As the numbers in Table 1 suggest, this method can be used profitably usually only for $k \leq 4$ (and for $k \geq 3$ only if the processed component is reasonably small). Actually, not all these cases must always be considered explicitly, because many of them can be ruled out at little extra cost using some heuristics. A basic idea for such heuristics uses the following lemma:

Lemma 36 Let z_i and z_j be two terminals in the separator S and let b^1_{ij} and s^2_{ij} be the bottleneck distance in G_1 and bottleneck Steiner distance in G_2 between z_i and z_j, respectively. Then the cases in which z_i and z_j are connected in G_1 can be discarded if $b^1_{ij} \geq s^2_{ij}$.

Proof: Consider a Steiner tree T connecting z_i and z_j in G_1. A bottleneck on the fundamental path between z_i and z_j has at least cost b^1_{ij}. Removing such a bottleneck and reconnecting the two resulting subtrees of T with the subpath corresponding to s^2_{ij}, we get again a feasible solution of no larger cost in which z_i and z_j are connected in G_2. □

Note also that for the cases in which we assume that z_i and z_j are connected in G_1, we do not merge z_i and z_j while solving the subinstance corresponding to G_2, but connect them with an edge of weight b^1_{ij}. In case this edge is not used in the solution of the subinstance, this can lead to more reductions.

This and similar observations can be used to rule out many cases in advance. Nevertheless, a question naturally arises: Can we find an alternative method that does not need explicit case differentiation? We will introduce such an alternative in the following section.

Local Bounds

Recall that the general principle of bound-based reduction methods is to compute an upper bound $upper$ and a lower bound under some constraint $lower_{constrained}$. The constraint cannot be satisfied by any optimal solution if $lower_{constrained} > upper$. The constraint is usually that the solution must contain some pattern (e.g., a vertex or an edge, or even more complex patterns like paths and trees, as in Section 3.5 on extended reduction techniques). But it is usually too costly to recompute a (strong) lower bound from scratch for each constraint. Here one can use an approach based on Linear Programming presented in Section 3.4.2: Any linear relaxation can provide a dual feasible solution of value $lower$ and reduced costs c'. We can use a fast method to compute a constrained lower bound

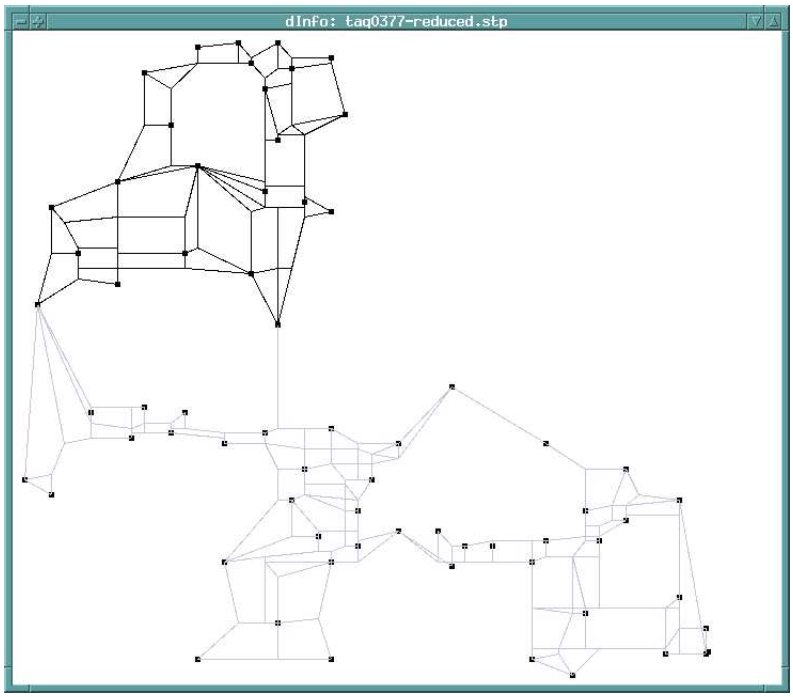


Figure 3.9: $|S| = 2$.

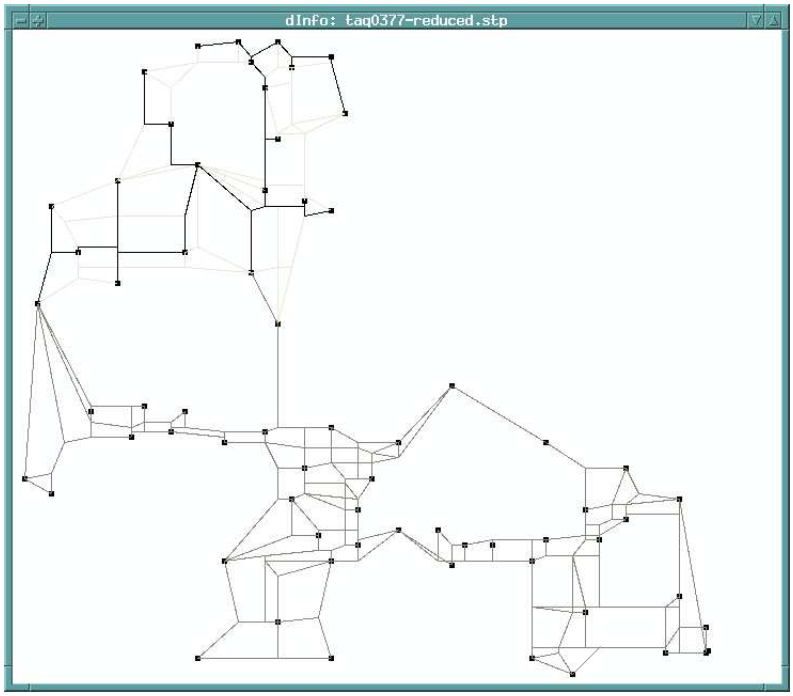


Figure 3.10: ($|V| = 133,\ |E| = 214,\ |R| = 45$).

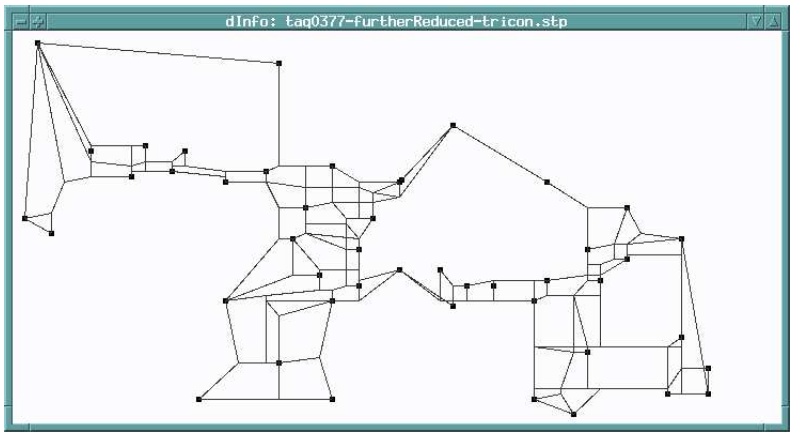


Figure 3.11: ($|V| = 133,\ |E| = 214,\ |R| = 45$).

$lower'$ with respect to c'. From Lemma 29 follows that $lower_{constrained} := lower + lower'$ is a lower bound for the value of any solution satisfying the constraint.

As an example for such a relaxation, consider the (directed) cut formulation P_C [Won84]. It has been described in Section 2.3.1. Here, we give an informal description: The Steiner problem is formulated as an integer program by introducing a binary x-variable for each arc (in case of undirected graphs, the bidirected counterpart is used, fixing a $z_1 \in R$ as the root). For each cut separating the root from another terminal, there is a constraint requiring that the sum of x-values of the cut arcs is at least 1. In this way, every feasible binary solution represents a feasible solution for the Steiner problem and each minimum solution (with value $v(P_C)$) a Steiner minimal tree (Figure 3.12). But in the optimal solution of the LP-relaxation LP_C variables can have fractional values. In Figure 3.13, the grey arcs have value 1/2 (the black arcs have value 1, the other arcs have value 0). One observes that a flow of one unit can still be sent from the root to each terminal, so all cut constraints are satisfied; and if the costs are adverse, an integrality gap can occur. This is in fact the case in this example, where the linear relaxation has optimum solution value $v(LP_C) = 6392.5$. As such gaps accumulate (e.g., in larger instances), the difference between the bounds grows, eventually causing the bound-based reductions to fail.
In the following, we show how to use locally computed bounds for bound-based reduction. This approach has two main advantages: The bounds can be computed faster; and there is less chance of accumulation of errors. The main difficulty is that the bounds must somehow take the dependence on the rest of the graph into account.

Let S be a terminal separator in G and G_1 and G_2 the corresponding subgraphs (Figure 3.14). The bounds will be computed locally in supplemented versions of G_2. Let C be a clique over S. We denote with (C, b) the weighted version of C with weights equal to bottleneck distances in G_1; similarly for (C, s) with weights equal to bottleneck Steiner distances in G. Let G_2' and G_2'' be the instances of the Steiner problem created by supplementing G_2 with (C, s) and (C, b), respectively. We compute a lower bound $lower_{constrained}(G_2'')$ for any Steiner tree satisfying a given constraint in G_2'' and an upper bound $upper(G_2')$ corresponding to an (unrestricted) Steiner tree in G_2'. The test condition is: $upper(G_2') < lower_{constrained}(G_2'')$.

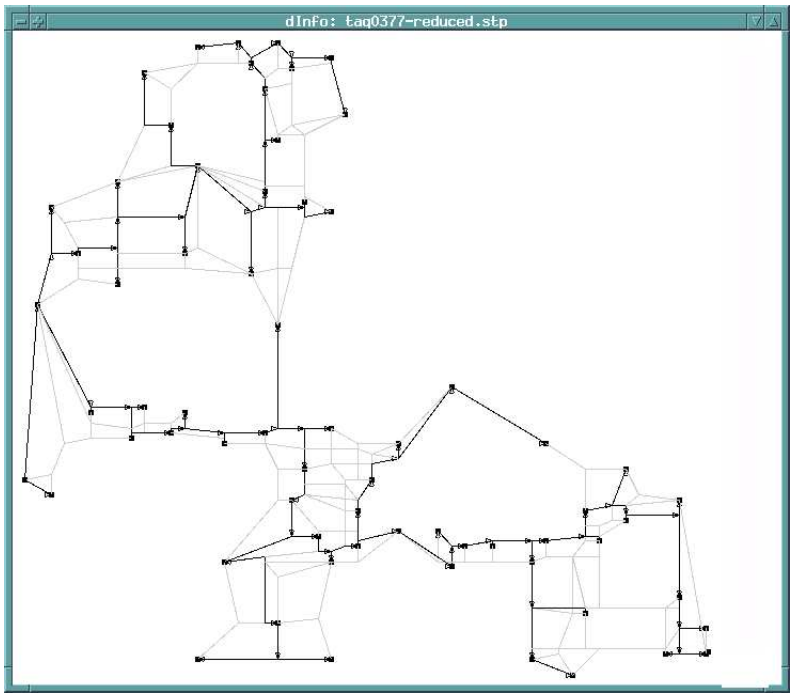


Figure 3.12: $v(P_C) = 6393$.

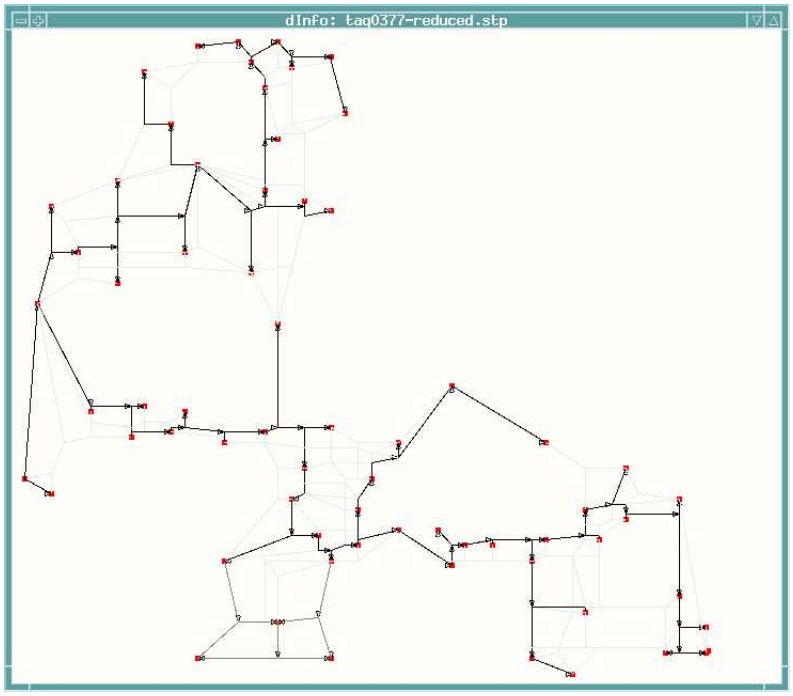


Figure 3.13: $v(LP_C) = 6392.5$.

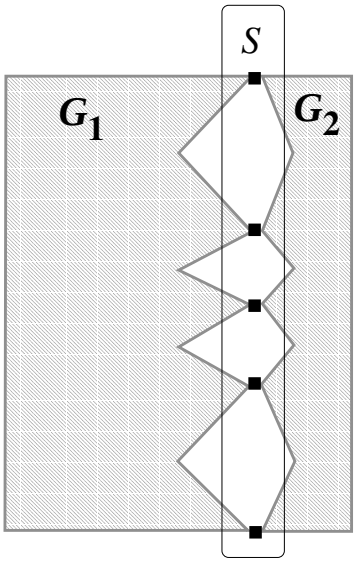


Figure 3.14: A terminal separator S and corresponding subgraphs G_1 and G_2.

Lemma 37 The test condition is valid, i.e., no Steiner minimal tree in G satisfies the constraint if $upper(G'_2) < lower_{constrained}(G''_2)$.

Proof: Consider $T^{opt}_{con}(G)$, an (unknown) optimum Steiner tree satisfying the constraint. The subtrees of this tree restricted to subgraphs G_1 and G_2 build two forests F_1 (with connected components T_i) and F_2 (Figure 3.15, left). Removing F_2 and reconnecting F_1 with $T^{upper}(G'_2)$ we get a feasible solution again, which is not necessarily a tree (Figure 3.15, middle). Let S_i be the subset of S in T_i. Consider two terminals of S_i: Removing a bottleneck on the corresponding fundamental path disconnects T_i into two connected components. Repeating this step until all terminals in S_i are disconnected in T_i, we have removed $|S_i| - 1$ bottlenecks, which together build a spanning tree $spanT_i$ for S_i (Figure 3.15, right). Repeating this for all T_i, we get again a feasible Steiner tree $T^{upper}(G')$ for the graph G', which is created by adding the edges of (C, s) to G. Note that the optimum solution values in G' and G are the same (see Section 1.3). Let $upper(G')$ be the weight of $T^{upper}(G')$. By construction of $T^{upper}(G')$, we have:

$$upper(G') = opt_{con}(G) + upper(G'_2) - c(F_2) - \sum_i c(spanT_i)$$

The edge weights of the trees $spanT_i$ correspond to bottlenecks in F_1, so by definition they cannot be smaller than the corresponding bottleneck distances in G_1. By construction of G''_2, all these edges (with the latter weights) are available in G''_2. Since the trees $spanT_i$ reconnect the forest F_2, together with F_2 they build a feasible solution for G''_2, which even satisfies the constraint (because F_2 did), so it has at least the cost $opt_{con}(G''_2)$. This means:

$$\begin{aligned} upper(G') &\leq opt_{con}(G) + upper(G'_2) - opt_{con}(G''_2) \\ &< opt_{con}(G) + lower_{constrained}(G''_2) - opt_{con}(G''_2) \quad \text{(because of the test condition)} \\ &\leq opt_{con}(G), \end{aligned}$$

thus $opt_{con}(G) > upper(G') \geq opt(G') = opt(G)$, meaning that the constraint cannot be satisfied without deteriorating the optimal solution value. □

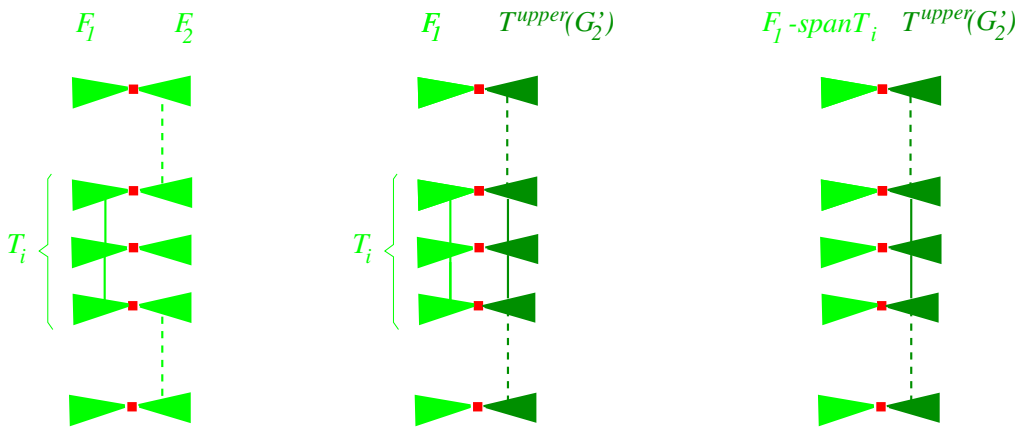


Figure 3.15: Construction of $T^{upper}(G')$ from $T^{opt}_{con}(G)$.

Studying the test condition, one detects a weakness of the lower bound used relative to the upper bound: bottleneck distances used in the lower bound correspond to single edges, whereas the bottleneck Steiner distances used in the upper bound correspond to whole paths and can be much larger. The attempt to use some paths in F_1 instead of bottlenecks fails because the tree $T^{opt}_{con}(G)$ and consequently the forest F_1 are unknown. But going through the proof, one observes that we can use the fact that the tree $T^{upper}(G'_2)$ is available: Instead of removing a single edge in F_1, we can remove a (longest) key path on the corresponding fundamental path in $T^{upper}(G'_2)$. This leads to the following improvement of the test: While choosing the edge weights for constructing G''_2, we can use the length of a key path in $T^{upper}(G'_2)$ whenever it is larger than the corresponding bottleneck distance in G_1. However, care must be taken to keep the key paths used disjoint.

An application example for this test is given in Figure 3.16, where a separator of size 4 and the corresponding component G_2 are shown (black edges). Figure 3.17 shows the result of application of the reduction method presented (black edges contracted, grey edges remaining); in Figure 3.18 the reduced instance is redrawn.

This is also an example of how reduction methods based on partitioning can reduce the errors in an LP-relaxation: As shown in Figure 3.19, the relaxation LP_C has now an integer optimal solution. It is perhaps interesting that this improvement is mainly achieved by applying the same relaxation locally.

3.7 Integration and Implementation of Tests

To study the effect of different combinations and orderings of the tests, we designed an interpreter for command-lines, where each test is encoded by a character. We also implemented a direct control of loops (through parentheses), their termination criteria, switching of parameters, etc. The main observation is that the (alternative-based) tests are not very sensitive to the order in which they are executed. On the other hand, the ordering often has an impact on the total time for reductions; in this sense the ordering cited in [HRW92] is a suitable one (although not necessarily the only one, as long as a fast version of PT_m is performed first).

For the implementation, we have chosen a kind of adjacency-list representation of networks (with all edges in a single array), but we sometimes switch to other auxiliary representations (all linear in

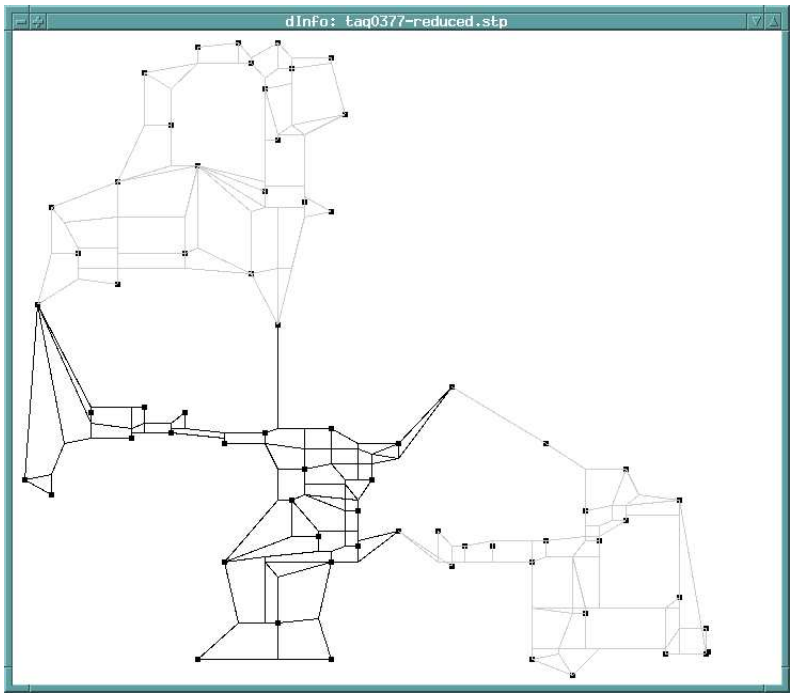


Figure 3.16: $|S| = 4$.

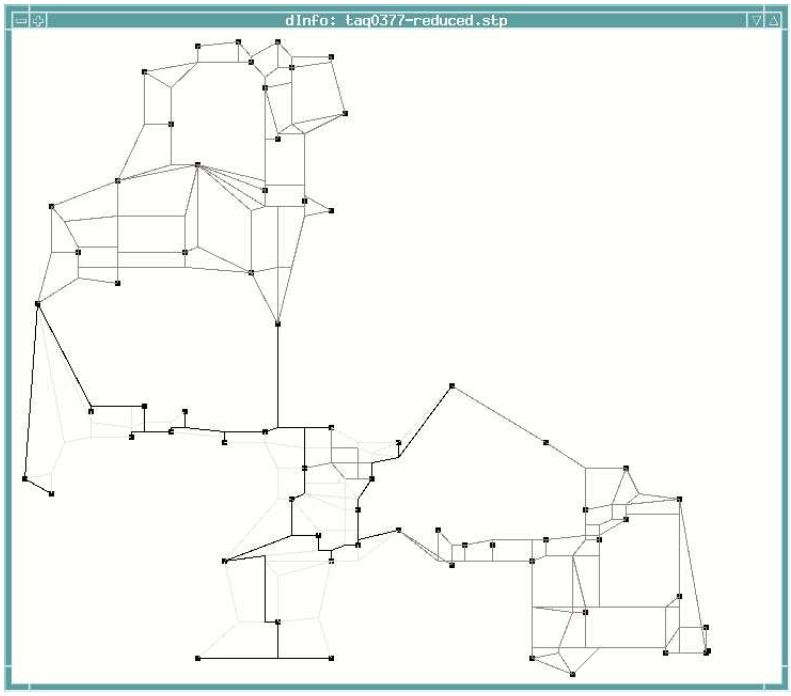


Figure 3.17: ($|V| = 121,\ |E| = 200,\ |R| = 41$).

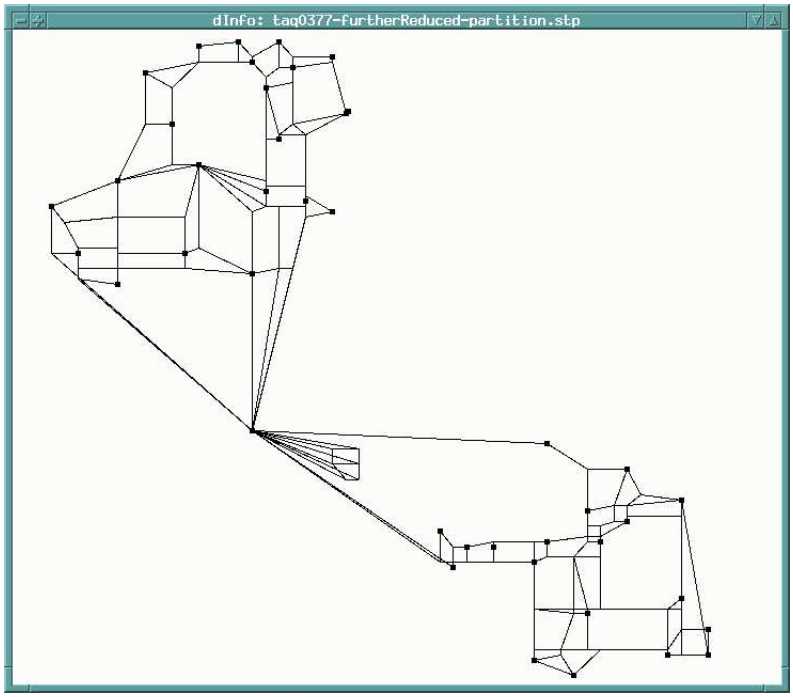


Figure 3.18: ($|V| = 121,\ |E| = 200,\ |R| = 41$).

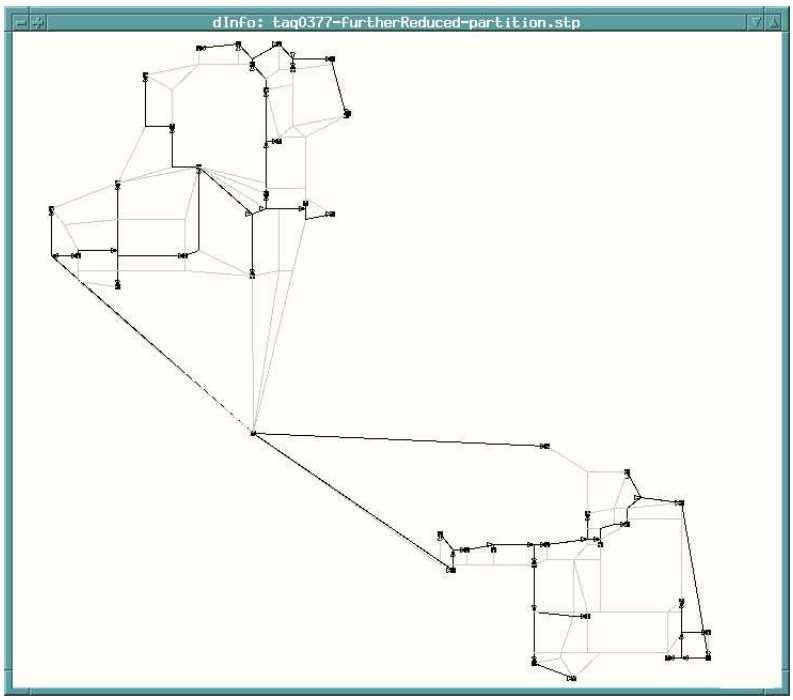


Figure 3.19: $v(LP_C) = 6393$.

the number of edges) for certain operations. For each test, we perform all actions in a single pass (and do not, for example, delete an edge and start the test from scratch). The details of the realization of the various actions are very technical and are omitted here; we merely mention that all actions following each test can be realized in a time dominated by the worst-case time $O(m + n \log n)$ of the fast tests.

With the additional requirement that in each loop of the selected tests a constant proportion (say 5%) of vertices and edges must be eliminated and that instances of trivially small size are solved directly (by enumeration), one gets the same asymptotic time bound for the whole reduction process as for the first iteration ($O(m + n \log n)$, if one confines oneself to the fast tests).

Another technical aspect is the efficient reconstruction of a solution for the original instance out of a solution for the reduced instance (which often consists of a single terminal). Saving appropriate information during the reduction process, this can be done in time $O(m)$. We always perform such a transformation after each run of the program, checking the feasibility and value of the solution in the original instance.

3.8 Some Experimental Results

In this section, we present some summarized results of the reduction package on instance groups of SteinLib (see Section 1.2 for a description of the problem instances). To give some impression of the effect of the different approaches, we present results for different subsets of the reduction techniques:

$O(m + n \log n)$**:** In these runs, we have only included the fast versions of the reduction tests PT_m, Triangle, NTD_k, NV, SL, PS, VR, and LDA, which all can be implemented with a worst-case running time of $O(m + n \log n)$. For computing an upper bound, we used a fast path heuristic with the same running time, which will be described in Section 4.2.

+ DUAL-ASCENT: These runs additionally use the DUAL-ASCENT algorithm, for generating a lower bound, the calculation of an auxiliary graph for ASCEND-AND-PRUNE (see Section 4.4), and for bound-based reductions.

+ Extended Reduction Techniques: In this column, the extended reduction tests are also used. Note that using extended reductions, there is an obvious tradeoff between reduction results and running time, which can be configured by increasing or decreasing the parameter for the maximum backtracking depth. A more detailed study of the results using extended reduction techniques is given in [PV01b].

+ Partitioning: Here, we additionally use the partitioning based reductions, presented in Section 3.6. As they require an exact solution of small subinstances, we also had to include the RG-test and branching, but only for the solution of the subinstances. Results including the RG-test are presented in the context of the exact algorithm, in Section 5.4.

We know of only a few other works where the reported reduction results reach a quality that is at least comparable to ours. One is the PhD thesis of Cees Duin [Dui93] who developed together with Ton Volgenant several of the most important classical tests. He modified these tests so that they are applicable at least to medium sized instances. Our results on these instances are similar, but in general we have better results: On the D-instances of SteinLib our fastest reductions already leave less than 2% of the edges, while his leave 24%. (Duin did not give results on larger benchmark instances.) The best other results on these instances that were published in a journal are from Koch and Martin [KM98]. For the D-instances they achieve reduction to 38% of the edges of the original network.

For VLSI instances, Uchoa, Poggi de Aragão, and Ribeiro [UdAR02] presented the best other results: With their reduction techniques only 6.2% of the edges remain, and the reductions took 367 seconds on a Sun Ultra Sparc with 167 MHz. With our reductions, on the average only 0.05% of the edges remain after 4 seconds on a Sparc-III CPU with 900 MHz. A more detailed comparison of the two reduction packages is given in [PV01b, PV02a].

instance	$O(m + n \log n)$		+ DUAL-ASCENT		+ Extended Red.		+ Partitioning	
group	time	remaining	time	remaining	time	remaining	time	remaining
	in sec.	edges in %	in sec.	edges in %	in sec.	edges in %	in sec.	edges in %
1R	0.08	57.41	0.12	45.60	0.32	0.00	0.20	0.00
2R	0.19	69.99	0.22	64.03	23.46	6.95	20.64	6.24
D	0.08	1.46	0.07	0.52	0.64	0.35	0.31	0.14
E	0.30	3.03	0.27	0.93	4.27	0.55	3.65	0.50
ES10000FST	9.42	62.35	92.68	62.39	874.23	46.05	1048.04	17.41
ES1000FST	0.58	62.67	0.80	62.79	4.82	46.83	9.63	20.32
I080	0.02	43.62	0.02	26.77	0.06	20.63	0.09	15.95
I160	0.05	63.98	0.06	33.61	0.31	28.77	0.43	25.60
I320	0.19	71.82	0.31	47.51	2.48	40.07	3.42	35.72
I640	0.96	79.88	1.89	53.17	15.32	45.12	25.33	43.29
LIN	2.31	31.84	1.72	27.12	207.55	9.29	194.75	6.08
MC	0.03	9.68	0.03	7.05	0.10	6.34	0.17	6.34
PUC	0.34	99.33	0.89	99.33	7.67	99.31	12.77	99.31
SP	0.17	37.50	1.39	37.50	5.75	37.50	7.81	37.50
TSPFST	0.24	30.71	0.80	29.93	7.53	20.37	15.48	7.84
VLSI	0.36	15.70	0.64	10.78	11.00	0.60	4.37	0.05
WRP3	0.15	92.68	0.24	91.41	42.24	58.01	44.26	52.28
WRP4	0.09	96.16	0.20	87.65	14.61	57.63	12.46	44.15
X	0.34	0.39	0.31	0.00	0.28	0.00	0.32	0.00

Table 3.1: Some experimental results for reductions.

Chapter 4

Fast Computation of Short Steiner Trees

4.1 Introduction

In this chapter, we focus on methods for computing upper bounds, in particular computing short (but not necessary optimal) Steiner trees in short running times.

The number of papers on this topic is enormous, there are many heuristics and every popular meta-heuristic has been tried (typically in many variants). A very comprehensive overview of articles published before 1992 can be found in [HRW92]. But the best results have been published after that (or have not yet even been published in a journal). In [PV97] we gave a comprehensive overview of the most important publications before 1997. Important recent developments can be found in [RUW02, PW02].

We have developed a variety of heuristics for obtaining upper bounds. Especially in the context of exact algorithms very sharp upper bounds are highly desired. So, our main concern was achieving very strong bounds, reaching the optimum as often as possible. On the other hand, the goal of obtaining short total empirical running times prohibited us from using heuristics that achieve good solution values only after long runs. In the following, we describe some of the methods we used in our attempt to achieve both goals simultaneously.

4.2 Path Heuristics

The repetitive shortest paths heuristic (SPH) is one of the empirically most successful classical heuristics for the Steiner problem in networks [TM80, HRW92, WS92, Voß92, PV01c, PW02]. The basic idea is to build a Steiner tree similar to Prim's algorithm for the minimum spanning tree problem: Start with a subtree that initially consists of one vertex. In each step, connect the current subtree with a shortest path to the next terminal. Repeat this until all terminals are connected. A final pruning step computes a minimum spanning tree of the vertices of the heuristic tree and removes non-terminals that have degree one in this tree. The worst-case running time for such a tree construction is $O(r(m + n \log n))$. To improve the quality of the solution, one usually repeats this process with different start vertices (repetitive SPH).

For the computation of shortest paths in each iteration the algorithm of Dijkstra is used, which uses tentative distances π_i for every vertex $v_i \in V$. A tentative distance π_i represents an upper bound to the distance between the current subtree and v_i. In [PW02] Poggi de Aragão and Werneck made a simple, but very effective observation: One does not have to reset tentative distances when a terminal is connected to the current subtree. The reason for this is that the old tentative distances are still valid upper bounds for the distances from the extended subtree. All one has to do is reinsert every vertex v_i into the priority queue of Dijkstra's algorithm (with $\pi_i = 0$) as soon as it becomes part of the current subtree. Although the empirical running time is improved drastically, it is unsatisfying that the worst-case running time is still $O(r(m + n \log n))$. Furthermore, as SPH is usually used in the repetitive variant, one can observe that very similar information is computed over and over without taking advantage of the previous computations.

Studying the repetitive shortest paths heuristic one observes that the actions can be divided into two phases (see [Dui93, DV97]): In the first phase, one can compute shortest paths from each terminal to all vertices; this can be done in time $O(r(m+n \log n))$. Using the information from the first phase, each run of the SPH in the second phase (constructing a Steiner tree by successively connecting the current tree (a single vertex at the beginning) to the closest terminal not in the tree by a shortest path) can easily be realized in time $O(rn)$.

In [PV97, PV01c], we presented an efficient implementation of this approach and also a version

with a worst-case running time of $O(m + n \log n)$. We will describe these heuristics in the next two sections. Note that we use these variants only as components of our other algorithms, not as stand-alone heuristics.

4.2.1 Faster Preprocessing for the Repetitive Shortest Path Heuristic

Our concern here is achieving empirical acceleration of the two-phase version of repetitive SPH. Regarding the first phase, we observe that the shortest paths need not be always computed completely:

Lemma 38 Let P be a shortest path between a terminal z and a vertex v, such that there is a vertex v' on P with $z' := base(v') \neq z$ and $d(z, z') \leq d(z, v)$. If v, but not z, belongs to the current tree T in the second phase, there exists at least one other path connecting T to a terminal not in T which is not longer than P. So, when computing shortest paths from z, we need consider neither v nor any vertex that would become a successor of v in the shortest paths tree.

Proof: There are two cases:
I) $z' \in T$: Since $d(z, z') \leq d(z, v)$, we can choose the path between z' and z.
II) $z' \notin T$: Since $d(z', v) \leq d(z', v') + d(v', v) \leq d(z, v') + d(v', v) = d(z, v)$, we can choose the path between v and z'. □

As a consequence, one can stop computing the shortest paths tree from a terminal z in the first phase as soon as the Voronoi region of z and the neighboring terminals (as defined in 1.3) have been spanned, because the shortest path between z and every vertex v visited afterwards contains a vertex $v' \in N(z')$ with z' a neighbor of z and $d(z, z') \leq d(z, v)$, since z' has already been spanned by the shortest paths tree. Furthermore, no shortest path via an intermediary terminal needs to be considered. These observations often lead to a considerable reduction in the empirical times, especially if the graph has many terminals and is not dense (the latter is almost always the case after reductions). Note that for graphs with few terminals, repetitive SPH is fast anyway.

For building the Steiner trees in the second phase, we prefer a realization that uses the concept of neighborhoods: Using the information from the first phase, we manage for each vertex v a list of neighboring terminals, sorted by (increasing) distances to v. A priority queue manages candidates for expansion of the tree, using the distance to the nearest terminal not in the tree as the key for insertion. Each time a vertex v is extracted from the queue, two cases can arise: Either the terminal corresponding to the key is not yet in the tree, in which case the tree is expanded by the corresponding shortest path (and the queue is updated); or it is already in the tree, in which case the neighbor list of v is scanned further until either a terminal not in the tree is visited (which delivers the key for reinsertion of v into the queue) or the end of the list is reached (meaning that v can be ignored). Although the worst-case time of this implementation ($O(rn \log n)$) is slightly worse than $O(rn)$ of the straightforward implementations, it is usually much faster, and the worst-case time is dominated by the first phase anyway.

4.2.2 A Path Heuristic with Good Worst Case Running Time

In situations where the worst-case time is the primary concern, we used a strengthening of the ideas above to design a heuristic with time $O(m + n \log n)$. Motivated by the fact that the vicinity of each terminal relevant for SPH often gets smaller with a growing number of terminals, one can simply force the first phase not to perform more than $O(m+n \log n)$ operations. But then it is no longer guaranteed that the relevant neighborhood of each terminal is captured. To remedy this defect, we simultaneously use the graph G' of Mehlhorn's fast implementation of $T'_D(R)$ [HRW92, Meh88], which we also

compute in the first phase, see Section 1.3. In addition to the priority queue described above, a second priority queue, offering expansion of the current tree through edges of G', is managed in the second phase. For each expansion, the better offer is accepted and both queues are updated appropriately.

The information gained in the first phase can be used more economically if not only one, but a (constant) number of Steiner trees are computed in the second phase, using different terminals or vertices as starting points.

This heuristic can be implemented with time $O(m + n \log n)$ and guarantees a performance ratio of 2.

4.2.3 Some Experimental Results Of Path Heuristics

Although the path heuristics mentioned before were designed only to be used as a component of other algorithms (especially in combination with reductions), they yield reasonable results even on their own. In Table 4.1, we give average running time and average gap from the optimal solution for instances from SteinLib (see Section 1.2 for a description of the instance groups). The four algorithms compared are the following:

one-phase SPH: One iteration of the simple shortest path heuristic, using the improvement of [PW02].

one-phase R-SPH: Repetitive variant of the above: Start from up to 100 different starting points (terminals are preferred as starting points).

two-phase R-SPH: Repetitive shortest path heuristic (up to 100 different starting points, terminals are preferred), using the restricted preprocessing of distances and $T'_D(R)$ to achieve a worst-case running time of $O(m + n \log n)$.

$T'_D(R)$**:** The distance network heuristic, for comparison with the two phase R-SPH.

All variants use the final minimum spanning tree pruning step, described above in the beginning of Section 4.2.

One observes that due to the improvement of [PW02], the one phase SPH variant is very fast, but the quality of the results is relatively weak. For this reason, the repetitive variant is usually used. Here, we see that using the fast two-phase variant can sometimes improve the running time. Furthermore, we can give an $O(m + n \log n)$ running time guarantee. The solutions quality produced by this two-phase variant is often the same as for the one-phase variant. This is due to the fact that in many cases the computation is focussed very effectively by the application of Lemma 38, and in spite of the rigid running time guarantee, all necessary distances can be computed in the first phase. In these cases, differences in the quality of the solutions of the one-phase and the two-phase variant reflect only different decisions in situations when ties of paths of equal length are broken. A different situation happens for example for the WRP3 and WRP4 instances, where many vertices are similarly close to many terminals. In the one-phase variant this leads to comparably high running times, as distance values have to be rechecked again and again. In the two-phase $O(m + n \log n)$ variant, many computations in the first phase have to be aborted. As a consequence, for many paths, a precomputed shortest path is not available and one has two use paths from $T'_D(R)$ and the solution quality deteriorates.

As a comparison we included the minimum spanning tree in the distance network $T'_D(R)$. Here one sees that quality of the solutions is drastically improved with the two-phase variant, although the same asymptotic running time guarantee can be given.

instance group	one phase SPH $O(r(m+n\log n))$		one phase R-SPH $O(r(m+n\log n))$		two phase R-SPH $O(m+n\log n)$		$T'_D(R)$ $O(m+n\log n)$	
	time	gap in %	time	gap in %	time	gap in %	time	gap in %
1R	0.001	4.59	0.054	1.40	0.017	3.19	0.001	8.13
2R	0.002	5.82	0.114	2.36	0.032	4.42	0.002	8.98
D	0.002	3.25	0.176	1.27	0.128	1.43	0.004	5.16
E	0.007	4.58	0.521	1.60	0.382	1.58	0.009	7.77
ES10000FST	0.065	2.32	5.682	2.27	6.489	2.23	0.071	3.18
ES1000FST	0.005	2.41	0.446	2.22	0.452	2.13	0.005	3.26
I080	0.001	14.66	0.011	1.35	0.004	1.41	0.001	18.56
I160	0.001	17.52	0.047	1.83	0.013	1.92	0.001	22.36
I320	0.004	19.11	0.193	2.51	0.038	3.80	0.004	25.54
LIN	0.015	3.65	1.176	2.27	0.235	2.14	0.015	5.74
MC	0.001	3.75	0.072	2.39	0.069	2.44	0.002	5.94
TSPFST	0.003	1.39	0.267	1.05	0.323	1.08	0.003	1.95
VLSI	0.004	2.63	0.314	1.12	0.118	1.59	0.004	5.45
WRP3	0.007	0.0007	0.572	0.0003	0.059	0.08	0.001	49.08
WRP4	0.003	0.002	0.282	0.001	0.040	0.12	0.001	42.09
X	0.017	1.05	0.778	0.34	0.285	0.34	0.053	1.05

Table 4.1: Experimental results of path heuristics.

Our implementation compares quite well with the implementation given in [PW02]. Although they use a faster computer (1.7 GHz Pentium 4), the running times for one iteration of the one-phase SPH are comparable (e.g., on the VLSI instances they need 8 milliseconds on average while we need only 4). The quality of the solutions is also similar. Note that one cannot expect exactly the same values, as there is some freedom in the choice of the starting vertex and in choosing a shortest path in the case of multiple shortest paths. For the two-phase variant they only implemented a simpler version that is much slower and runs into memory problems on larger instances (although they cite our paper where we also presented the fast variant).

4.3 Reduction-based Heuristics

Working with reductions, one often gets the impression that some of the tests are too cautious. Sometimes nice ideas for strengthening a test turn out to be not universally valid. Of course even the strangest exception is enough to make a reduction test completely useless for (direct) integration into an exact algorithm. But with respect to heuristics, the situation is fundamentally different: Here a much stronger orientation towards the frequent case can be adopted.

The idea used here is to support the normal (exact) reduction tests through some heuristic ones. It must be emphasized that the goal is not reducing the graphs by brute force, but only giving an impulse in situations where the exact reduction process is blocked, in order to activate it again. In this context, it is particularly advantageous if it can be assumed that the actions performed could have been carried out by a more powerful, but unknown exact test anyway.

A natural basis for such an approach is given by the test VR (see Section 3.4.1). This test is kept very cautious to make a comprehensible proof possible. Furthermore, one observes readily that in case the upper bound used is not optimal, the test could potentially perform more (exact) reductions if a

better upper bound was available. The idea is now to perform the usual actions of this test without an upper bound each time the other tests are blocked. At each application, a certain proportion of vertices is eliminated (directly or after replacing of incident edges) according to the same criteria as in the exact version of the test (sum of distances to the next two or three terminals). Motivated by the fact that for a large ratio r/n the alternative-based reductions are very successful anyway and the test VR is usually effective only for small r/n, the proportion of the vertices being eliminated is a function of n and r, getting smaller with growing r/n. With the additional postulation that during each application of the tests a constant percentage (say 5%) of vertices and edges is eliminated, the asymptotic time for all iterations together is the same as for the first one, namely $O(m + n \log n)$. To make sure that the instance is not made infeasible by the heuristic reductions, we further forbid direct elimination of vertices in the current tree $T'_D(R)$. The computation of $T'_D(R)$ also yields as a side effect a guaranteed performance ratio of 2. We call this whole procedure **PRUNE**.

The idea of not eliminating the nodes of a Steiner tree can be further utilized by using a (good) heuristic solution instead of $T'_D(R)$ for guiding the heuristic reductions. We use the implementation of repetitive SPH described in Section 4.2 (with a constant upper bound for the number of repetitions) for this purpose, but any other good solution would do, too. On the other hand, we make the actions of the heuristic reductions somewhat bolder, eliminating vertices only directly (without replacing of incident edges). Note also that even Steiner nodes of the guiding heuristic solution may be eliminated, but only by the exact tests; these tests are guaranteed not to deteriorate the optimum. We call this variant of the PRUNE heuristic **GUIDED-PRUNE**.

4.4 Relaxations and Upper Bounds

Computing lower bounds is not the only motivation for dealing with relaxations; the information obtained can also be used (among other things) to obtain upper bounds.

Consider the (directed) cut formulation P_C of the Steiner problem: Given an optimal solution $\hat{x}$ of its linear relaxation LP_C, the complementary slackness conditions state that each edge $[v_i, v_j]$ with $\hat{x}_{ij} > 0$ has zero reduced cost. Assuming that there is some similarity between some optimal solutions of the integer program P_C and its linear relaxation LP_C, it is well motivated to search an (optimal) solution in a subgraph containing the edges with reduced cost zero.

The algorithm DUAL-ASCENT, attempting to construct an optimal solution for DLP_C, adjusts the reduced costs favorably. So it is very natural to search for a solution in the set of edges whose reduced costs are set to zero by this algorithm, an idea already used in [Won84, Voß92]. The auxiliary graph to be searched for a good solution need not contain all these edges; we have experimented with several schemes and gained the best overall results with a subgraph containing the (undirected edges corresponding to) edges on zero-cost ways (with respect to reduced costs) from the root to another terminal, although other variants are not inappropriate either.

Having chosen such an auxiliary graph, the key question is how to obtain an (optimal) solution for the corresponding instance. The structure of such instances is very suitable for the application of our PRUNE heuristics; in particular, there are often long chains of vertices that are replaced by long edges through the NTD_2 test, making other alternative-based reductions very effective; and the heuristic reductions do the rest of the job. We call the whole procedure of doing fast reductions, calling DUAL-ASCENT, determining a subgraph and performing a PRUNE heuristic in the subgraph **ASCEND-AND-PRUNE**.

Since we are working in a subgraph of G, the time bounds for the PRUNE heuristics (which are dominated by the worst case time of DUAL-ASCENT) are guaranteed in any case. Empirically, since

the PRUNE heuristics run extremely fast on the auxiliary graphs, this kind of computation of upper bounds should be performed after each call to DUAL-ASCENT.

Although the experimental solution quality of this heuristic is impressive, it still sometimes misses the optimum. We found out that in almost all such cases the reason is simply that the auxiliary graph does not contain an optimal solution (and not that the PRUNE heuristics do not find it). This observation suggests a supplementation of this heuristic: The Steiner tree found in the subgraph can be used as the guiding solution for a call to GUIDED-PRUNE in the original graph. In the cases mentioned, this approach often improves the solution value, frequently leading to the optimum.

By applying the idea of the PRUNE heuristics directly to the original graph, one can do without the auxiliary graphs altogether. Let $lower$ and $\tilde{c}$ be the lower bound and the reduced cost vector provided by DUAL-ASCENT and $\hat{x}$ an optimal solution of P_C with value $optimum$. The inequality $\tilde{c} \cdot \hat{x} \leq optimum - lower$ (see Lemma 29) strongly suggests that normally there cannot be many edges with large reduced costs in an optimal solution. This motivates another heuristic, **SLACK-PRUNE**, that basically follows the same scheme as GUIDED-PRUNE, but uses the criterion of the test DA for eliminating vertices. The guiding solution is computed by a call of PRUNE in the auxiliary graph described above, since the necessary information is available after performing DUAL-ASCENT anyway. The running time is dominated by the worst-case time of DUAL-ASCENT. Using the same arguments as in the case of PRUNE, one gets the time bound $O(m \cdot \min\{m, nr\})$. But in combination with reductions, the empirical times are much smaller than the above term suggests.

As in DUAL-ASCENT, dual feasible solutions and corresponding reduced costs for LP_C are calculated during the row generating algorithm (Section 2.9.2). This information can be used to generate auxiliary graphs similar to those in ASCEND-AND-PRUNE. But in this case there are not necessarily paths with reduced cost zero from the root to all terminals. The auxiliary graph in this context contains all vertices with the property that there is a path from the root over this vertex to another terminal not longer (with respect to reduced costs) than the longest shortest path from the root to another terminal. This auxiliary graph can be used as in ASCEND-AND-PRUNE.

A classical method for utilizing the information provided by linear relaxations is to use an ordinary heuristic in the original network with modified edge costs $c'_{ij} = c_{ij}(1 - x_{ij})$ (where x is the primal solution of the current linear program). But this is not generally a good idea, because the structure of the primal solutions does not provide a good guide for a primal heuristic until the most advanced stages of the row generating algorithm.

These latter approaches only work in combination with explicit solution of linear programs and are therefore not suitable for fast, stand alone heuristics. But as a complement to the row generating strategy, they are frequently effective, especially in the advanced stages of the algorithm.

4.5 Combination of Steiner Trees

During the reduction process and especially while solving instances exactly, one usually gets several distinct heuristic solutions. In general, it is not the best idea to simply keep the best solution and forget the others. It is possible that solutions with a worse value are better locally, and one can try to keep the best part of each solution.

We have developed several techniques for realizing the idea above. One simple and effective way is to consider the graph consisting of the union of the edge sets of two (or more) Steiner trees. In this graph, one can call a (powerful) heuristic again or even try an exact solution. Such graphs have frequently several (nontrivial) biconnected components, which makes the (exact) solution considerably faster. Using such schemes, we frequently get improvements in solution values (in case they

were not optimal anyway). Since the instances generated through such combinations (in the following called combination-instances) are almost always solved to optimality through (fast) reductions, these improvements are gained at no significant extra cost.

For the results reported in this section, we simply call a PRUNE heuristic in such combination-instances; in particular, in the context of the heuristic SLACK-PRUNE we call the same heuristic (only without combinations) again in each combination-instance.

4.6 Experimental Results and Evaluation

In this section, we present some experimental results for reductions-based heuristics on instances from [SteinLib] (see Section 1.2 for a description of the instance groups). All of them are heuristics with a worst-case time describable by a polynomial of low order, as explained in the previous sections. Other good heuristics in the literature can be found in [Dui93, DV97, Esb95, Ver96, RUW02].

For comparison, we include the results of Ribeiro, Uchoa, and Werneck [RUW02], which are the best other results we are aware of. They use a combination of several techniques: A greedy randomized adaptive search procedure (GRASP), a weight perturbation strategy (which uses the knowledge of which edges are used frequently in many solutions), a local search procedure, a heuristic for the combination of good solutions, and many reduction techniques. Unfortunately, they include a preprocessing for the D, E, and VLSI-instances in their runs, but did not include the preprocessing time in the reported running times as we did. They report results only for those instances that could not be solved with their preprocessing routine, but we rescaled the values (assuming that instances solved by the preprocessing produced an optimal solution in zero time), so that at least the values for the average gaps are comparable. A stroke in the table means that no results were reported for these instances. The running times for [RUW02] where measured on a 300 MHz AMD and a Pentium II 400 MHz.

instance	PRUNE		ASCEND-&-PRUNE		SLACK-PRUNE		[RUW02]	
group	time	gap in %	time	gap in %	time	gap in %	time	gap in %
1R	0.13	1.36	0.07	1.03	0.22	0.00	—	—
2R	0.27	1.42	0.16	1.59	10.91	0.00	—	—
D	0.10	0.07	0.07	0.02	0.14	0.00	> 11	0.04
E	0.39	0.31	0.25	0.13	1.64	0.00	> 86	0.05
ES10000FST	7.56	1.11	30.88	0.67	2101.89	0.38	—	—
ES1000FST	0.57	1.01	0.38	0.53	18.57	0.19	—	—
I080	0.07	1.15	0.02	1.65	0.43	0.06	> 2	0.01
I160	0.31	1.97	0.07	1.69	1.68	0.10	> 16	0.13
I320	1.63	2.84	0.30	1.81	7.46	0.14	> 108	0.15
LIN	1.51	1.44	1.09	0.76	153.99	0.04	—	—
MC	0.05	1.70	0.04	1.01	0.95	0.42	—	—
TSPFST	0.21	0.42	0.38	0.31	32.01	0.04	—	—
VLSI	0.33	0.39	0.40	0.35	7.17	0.004	> 830	0.01
WRP3	0.15	0.0006	0.14	0.0003	17.79	0.00003	—	—
WRP4	0.10	0.07	0.10	0.0006	6.19	0.00006	—	—
X	0.44	0.17	0.29	0.00	0.23	0.00	—	—

Table 4.2: Some experimental results on upper bound calculation.

From Table 4.2, one can see that SLACK-PRUNE is clearly superior to the approach of Ribeiro, Uchoa, and Werneck. Even taking into account the different speed of the machines, our approach is faster in most cases. Remember also that in [RUW02] the preprocessing time was not recorded. Additionally, the quality of the solutions produced by SLACK-PRUNE is better, even on the larger incidence instances (I160, I320), which were constructed to be difficult for the known reduction techniques. Note that the percentage of pruned vertices is a simple parameter for the tradeoff between running time and solution quality, i.e., better solutions can be achieved simply by decreasing this parameter at the cost of longer running times.

The two heuristics PRUNE and ASCEND-AND-PRUNE are very fast while producing fairly good results. Note that PRUNE, although it has the better running time guarantee of $O(m+n \log n)$, in many cases takes longer than ASCEND-AND-PRUNE. The reason is that the techniques used in ASCEND-AND-PRUNE, in particular DUAL-ASCENT, are on the one hand much faster than the worst-case time bound suggests, and on the other hand reduce a problem instance much more efficiently.

Chapter 5

Solving to Optimality

5.1 Introduction

In this chapter, we present our results concerning the computation of optimal Steiner trees.

As with the upper bound calculations, the number of competitors is huge: For every classical approach (enumeration algorithms, dynamic programming, branch-and-bound, branch-and-cut) there are many published results and the presented algorithms are sometimes quite involved. Again, we point to [HRW92] for a survey and only mention the most important updates:

- Duin [Dui93] improved the preprocessing techniques and invented some heuristics that were integrated into a branch-and-bound algorithm, which was clearly the best at its time.
- Koch and Martin [KM98] used some of the improved preprocessing techniques and developed a very powerful and robust branch-and-cut algorithm.
- Uchoa et al. [UdAR02] contributed some new extended preprocessing techniques (which were further generalized in Section 3.5). Their branch-and-cut algorithm is currently, to the best of our knowledge, the best exact algorithm produced by other authors.

In the next section, we describe a dynamic programming approach that is useful for solving certain small subgraphs. After that we will describe the integration of the previously mentioned methods for reductions and upper and lower bound into an exact algorithm.

5.2 A Dynamic Programming Approach for Subgraphs

In this section, we present a practical algorithm for solving the Steiner problem in graphs with a small width-parameter (a formal definition of width is given in the next section). The running time of the algorithm is linear in the number of vertices when the width is constant, thus it belongs to the category of algorithms exploiting the fixed-parameter (FP) tractability of NP-hard problems. But the applicability of this algorithm is much broader in the context of our work. Due to the use of reduction techniques based on partitioning (Section 3.6) we can already profit from small width in subgraphs of the given instance. These techniques are in turn applicable to a very wide range of instances of the Steiner problem in networks after applying extended reduction methods (Section 3.5) and instances of geometric Steiner problems after FST-generation [PV03, WWZ00].

The width-concept here is closely related to pathwidth. (For an overview of subjects concerning pathwidth and the more general notion of treewidth see [Bod93].) There are already FP-polynomial algorithms for the Steiner problem in graphs; specifically, in [KS90] a linear-time algorithm for graphs with bounded treewidth is described. But this algorithm is more complicated than the one we present here, and its running time grows faster with the (tree-) width (it is given in [KS90] as $O(nf(d))$ with $f(d) = \Omega(d^{4d})$, where d is the treewidth of the graph); so it seems to be not as practical as our algorithm, and no experimental results are reported in [KS90]. In a different context (network reliability), a similar approach using pathwidth is described in [PT01], which is practical for a range of pathwidths similar to the one considered here. We also adapted this approach to the Steiner problem, but the experimental results were not as good as with the our approach.

A theoretically more powerful concept is the branch-width decomposition. It was formalized by Robertson and Seymour [RS91] and applied successfully to the TSP [CS02] and to the Steiner problem by Cook et. al. [Coo02]. We will not describe this approach, because experimental results showed that no gain can be expected from it: Cook was not able to solve many instances of the TSPFST group from [SteinLib], while we were able to solve them using a combination of our reduction techniques,

in particular the partitioning-based reductions (Section 3.6), and the Dynamic Programming approach for subgraph presented in this section.

5.2.1 Additional Definitions

A **path-decomposition** of a graph $G = (V, E)$ is a sequence of subsets of vertices $(X_1, X_2, \ldots, X_r)$, such that

1. $\bigcup_{1 \leq i \leq r} X_i = V$,
2. $\forall (v, w) \in E \quad \exists i \in \{1, \ldots, r\} : \quad v \in X_i \wedge w \in X_i$,
3. $\forall i, j, k \in \{1, \ldots, r\} : \quad i \leq j \leq k \Rightarrow X_i \cap X_k \subseteq X_j$.

The **pathwidth** of a path-decomposition $(X_1, X_2, \ldots, X_r)$ is $\max\{|X_i| \mid 1 \leq i \leq r\} - 1$. The **pathwidth** of a graph G is the minimum pathwidth over all possible path-decompositions of G.

5.2.2 The Algorithm

The outline of the algorithm is as follows: We maintain a set of already visited vertices and a subset of them (the **border**) that are adjacent to some non-visited vertex. In each step, the set of visited vertices is extended by one non-visited vertex adjacent to the border. For all possible partitions in each border, we calculate (the cost of) a forest of minimum cost that contains all visited terminals with the property that each tree in the forest spans just one of the partition sets. We are finished when all vertices have been visited.

The motivation behind this approach is as follows: For any optimal Steiner tree T, the subgraph of T when restricted to the visited vertices is a forest, which also defines a partition in the border. The plan is to calculate these forests in a bottom-up manner, in each step using the values calculated in the previous step. If the size of all borders can be bounded by a constant, the total time can be bounded by the number of steps times another constant.

For an arbitrary fixed ordering $v_1, \ldots, v_n$ of the vertices and any $s \in \{1, \ldots, n\}$, we define $V_s := \{v_1, \ldots, v_s\}$ and denote with G_s the subgraph of G with vertex set V_s. In the following, we assume an ordering of the vertices with the property that all G_s are connected. (For example, a depth-first-search traversal of G delivers such an ordering.)

We denote with B_s the border of V_s, i.e., $B_s := \{v_i \in V_s \mid \exists (v_i, v_j) \in E : v_j \in V \setminus V_s\}$. With L_s we denote the set of vertices that leave the border after step s, i.e., $L_s := (B_{s-1} \cup \{v_s\}) \setminus B_s$. The inclusion of v_s in this definition should cover the case that v_s has no adjacent vertices in $V \setminus V_s$; this simplifies some other definitions.

Consider a set Q, $B_s \cap R \subseteq Q \subseteq B_s$, and a partitioning $\mathcal{P} = \{P_1, \ldots, P_t\}$ of Q into nonempty subsets, i.e., $\dot{\bigcup}_{1 \leq i \leq t} P_i = Q$ and $\emptyset \notin \mathcal{P}$. For a partition $\mathcal{P}$ and a set $L \subseteq V$ we define $\mathcal{P} - L := \{P_i' \mid P_i \in \mathcal{P},\ P_i' = P_i \setminus L\}$. Let $F(s, \mathcal{P})$ be a forest of minimum cost in G_s containing all terminals in V_s and consisting of t (vertex-disjoint) trees $T_1, \ldots, T_t$ such that T_i spans P_i for all $i \in \{1, \ldots, t\}$. With $c(s, \mathcal{P})$ we denote the cost of $F(s, \mathcal{P})$.

Let $V_0 = B_0 = \emptyset$ and set $c(0, \emptyset) = 0$. The value $c(s, \mathcal{P})$ can be calculated recursively using a case distinction:

I) $v_s \in Q$:

$$c(s,\mathcal{P}) = \min\{c(s-1,\mathcal{P}') + C \mid \begin{array}{l} \mathcal{P}' = \{P_1,\ldots,P_r\},\ j \in \{0,\ldots,r\}, \\ \mathcal{P} = (\{\{v_s\} \cup \bigcup_{1\le l\le j} P_l\} \cup \{P_{j+1},\ldots,P_r\}) - L_s, \\ \forall\, 1 \le l \le j : v_l \in P_l, \\ C = \sum_{1\le l\le j} c(v_l, v_s)\}, \end{array}$$

II) $v_s \notin Q$:

$$c(s,\mathcal{P}) = \min\{c(s-1,\mathcal{P}') \mid \mathcal{P} = \mathcal{P}' - L_s\}.$$

The cost of an optimal Steiner tree in G is

$$\min\{c(s,\mathcal{P}) \mid R \subseteq V_s, |\mathcal{P}| = 1\}.$$

Obviously the forests $F(s,\mathcal{P})$ (and an optimal Steiner tree) can be calculated following the same pattern.

5.2.3 Dynamic Programming Implementation

By using the recursive formula above, the necessary values can be calculated in a bottom-up manner by memorizing, for each step s, the values $c(s,\mathcal{P})$. We assume $c(s,\mathcal{P}) = \infty$ if no partition $\mathcal{P}$ is calculated at step s. This leads to the following algorithm, written in pseudocode:

```
1   s = 0; r = 0; opt = ∞;            "r : number of visited terminals"
2   c(s, ∅) = 0;
3   while s < n :
4       s = s + 1; determine v_s, B_s and L_s;
5       if v_s ∈ R : r = r + 1;
6       forall P with c(s − 1, P) ≠ ∞ :
7           oldCost = c(s − 1, P);
8           if v_s ∉ R and ∅ ∉ P − L_s :
9               c(s, P − L_s) = oldCost;
10          Pcandidates = {P_i ∈ P | ∃ v_i ∈ P_i : (v_i, v_s) ∈ E};
11          forall Pconnect ⊆ Pcandidates :
12              connectionCost= Σ_{P_i∈Pconnect} min_{v_i∈P_i,(v_i,v_s)∈E} c(v_i, v_s);
13              Pstay = P\Pconnect; Pnew = ({{v_s} ∪ ⋃_{P_i∈Pconnect} P_i} ∪ Pstay) − L_s;
14              if ∅ ∉ Pnew and c(s,Pnew) > oldCost + connectionCost :
15                  c(s,Pnew) = oldCost + connectionCost;
16                  if r = |R| and |Pnew| = 1 :      "feasible Steiner tree"
17                      opt = min(opt, c(s,Pnew));
```

5.2.4 Running Time

Let p_s denote the number of partitions at step s. We have

$$p_s = \sum_{R\cap B_s \subseteq Q \subseteq B_s} \sum_{i=1}^{|Q|} \left\{ {|Q| \atop i} \right\} = \sum_{R\cap B_s \subseteq Q \subseteq B_s} B(|Q|),$$

where $B(b)$ is the b-th Bell number; so $p_s = O(2^{b_s}B(b_s))$ with $b_s := |B_s|$. We only maintain one global list of partitions, which is updated after each step, keeping for each valid partition a solution of minimum cost. Because of the loop in line 11, this list can grow to at most $l_s := 2^{b_s}p_s = O(2^{2b_s}B(b_s))$ partitions. Eliminating the duplicates can be done by sorting the list: Each partition can be represented as a (lexicographically) sorted string (of length at most $2b_s$) of sorted substrings (of length at most b_s) separated by some extra symbol. Using radix sort, all the individual sortings of l_s strings can be done in total time $O(n + l_s b_s)$. Sorting the resulting list of l_s strings takes again $O(n+l_s b_s)$. We set aside for now a total extra time of $O(|E|)$ for the operations on edges; and assume that an ordering of vertices is given (these points are explained below). The (rest of the) operations in lines $12 - 17$ can be carried out in time $O(b_s)$. This gives the total running time $O(\sum_{s=1}^{n} b_s 2^{2b_s}B(b_S))$. Note that this bound implicitly contains the extra amortized time $O(|E|)$ by the following observation: After a vertex is visited for the first time, it remains in the border as long as it has some non-visited adjacent vertex; so each edge is accounted for by its first-visited endpoint.

Now if we can guarantee an upper bound b for the size of all borders, we have an upper bound of $O(nb2^{2b}B(b))$ for the running time. Using the very rough upper bound $(2b)^b$ for $B(b)$ we get the running time: $O(n2^{b\log b+3b+\log b})$. This means that the algorithm runs in linear time for constant b and, for example, in time $O(n^2)$ for $b = O(\log n/\log\log n)$.

For the actual implementation, some modifications are used. For example, avoiding duplicate partitions is done using hashing techniques, which reduces the amount of necessary memory.

5.2.5 Ordering the Vertices

In Section 5.2.6 we will show that finding an ordering of vertices such that the maximum border size equals b is (up to some easy transformations) equivalent to finding a path-decomposition of pathwidth b. The problem of deciding whether the pathwidth of a given graph is at most b, and if so, finding a path-decomposition of width at most b is $\mathcal{NP}$-hard for general b [ACP87], but for constant b, this problem can be solved in linear time [Bod96]. However, already for $b > 4$ the corresponding algorithm is no longer practical [R¨oh98], and it seems that no practical exact algorithm is known for more general cases [Bod02]. Furthermore, we have a more specific scenario; for example we differentiate between terminals and nonterminals. So for the actual implementation we use a heuristic, which has produced quite satisfactory results for our applications. The heuristic chooses in each step a vertex v_s adjacent to the border using a (ad hoc) priority function of the following parameters:

- size of resulting set L_s,
- number of visited vertices in the adjacency list of v_s,
- is v_s a terminal (1/0),
- number of edges connecting V_s and $V \setminus V_s$.

We select the starting vertex by trying a small number of terminals and performing a sweep through the graph without actually computing the partitions, but by estimating their number using another (ad hoc) function and selecting the one with the smallest overall value.

A straightforward implementation of this heuristic needs time $O(n^2)$ for all choices. This bound could be improved using advanced data structures for priority queues and additional tricks, but the ordering has not been the bottleneck in our applications; and theoretically a better (linear for constant b as in our applications) time bound for path-decomposition is available anyway.

5.2.6 Relation to Pathwidth

In this section, we show that every path-decomposition with pathwidth k delivers a sequence of borders $B = (B_1, \ldots, B_s, \ldots, B_n)$ such that $\max\{|B_s| \mid 1 \leq s < n\} \leq k$ and vice versa.

First, note that the 3rd condition in the definition of path-decomposition can be rewritten as follows: There are functions $start$, end: $|V| \to \{1, \ldots, r\}$ with $v \in X_j \Leftrightarrow start(v) \leq j \leq end(v)$.

We call a path-decomposition **bijective** if the mapping $start$ is a bijection.

Lemma 39 Every path-decomposition $X = (X_1, \ldots, X_r)$ can be transformed to a bijective path-decomposition $(X'_1, \ldots, X'_n)$ of no larger pathwidth.

Proof: We modify the sequence X as follows:
As long as there is some i with $start^{-1}(\{i\}) = \emptyset$, remove X_i from X; adapting the functions $start$ and end.
As long as there are $v \neq v'$ with $start(v) = i = start(v')$, define $X_{j+1} = X_j$ for all $j \geq i$ and remove v' from X_i; adapting the functions $start$ and end.
One easily observes that the final resulting sequence X' satisfies the properties for a path-decomposition, is bijective, and has no greater pathwidth. □

We call a path-decomposition with functions $start$, end **minimal** if it holds:

$$end(v) \geq i \Rightarrow start(v) = i \vee \exists (v, w) \in E : \ start(w) \geq i.$$

(Note that the other direction is satisfied for every path-decomposition.)

Lemma 40 Every (bijective) path-decomposition X can be transformed to a minimal (bijective) path-decomposition of no larger pathwidth.

Proof: For every v, set $end(v) := \max\{start(w) \mid (v, w) \in E \vee v = w\}$. Delete v from all X_i with $i > end(v)$. One easily observes that the resulting sequence satisfies the properties for a path-decomposition, has no greater pathwidth, is minimal, and remains bijective if the original decomposition was bijective. □

Lemma 41 Let $(X_1, \ldots, X_n)$ be a minimal, bijective path-decomposition of G with the functions $start$ and end. Assume that the vertices are ordered according to their $start$ values, i.e., $start(v) = s \Leftrightarrow v \in V_s \setminus V_{s-1}$. For each $s \in \{1, \ldots, n\}$ it holds: $X_s = \{v_s\} \cup B_{s-1}$.

Proof:

$$\begin{aligned}
v \in X_s &\Leftrightarrow start(v) \leq s \wedge end(v) \geq s \\
&\Leftrightarrow (start(v) = s \vee start(v) < s) \wedge (start(v) = s \vee \exists (v, w) \in E : \ start(w) \geq s) \\
&\Leftrightarrow start(v) = s \vee (start(v) < s \wedge \exists (v, w) \in E : \ start(w) \geq s) \\
&\Leftrightarrow v = v_s \vee (v \in V_{s-1} \wedge \exists (v, w) \in E : \ w \in V \setminus V_{s-1}) \\
&\Leftrightarrow v = v_s \vee v \in B_{s-1}
\end{aligned}$$

□

It follows that every path-decomposition of G can be transformed to a path-decomposition $X = (X_1, \ldots, X_n)$ of no larger pathwidth such that for an ordering of vertices according to the $start$ function of X it holds: $X_s = \{v_s\} \cup B_{s-1}$. On the other hand, it is easy to verify that each ordering of vertices and the corresponding sequence of borders $(B_1, \ldots, B_n)$ deliver a (minimal, bijective) path-decomposition X by setting $X_s = \{v_s\} \cup B_{s-1}$. In each case, we have: $\max\{|X_s| \mid 1 \leq s \leq n\} - 1 = \max\{|B_{s-1}| \mid 1 \leq s \leq n\}$.

5.3 Putting the Pieces Together: The Exact Algorithm

In this section we describe the synthesis of an exact algorithm from the components described in the previous sections.

5.3.1 Interaction of the Components

A central feature of our exact algorithm is that the various components (reduction tests, lower bounds and upper bounds) do not act independently of each other, as described in detail in previous sections:

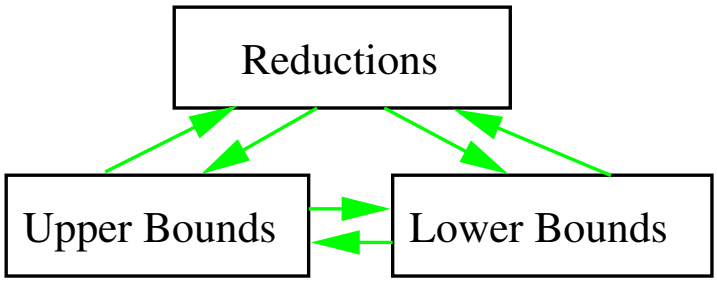


Figure 5.1: Interaction of the different components.

The bound-based reductions depend on upper and lower bounds; and the computation of upper and lower bounds profits from reductions, both in terms of running time and quality of results. The idea behind reduction tests is also the central part of the reduction-based heuristics for computing upper bounds. Further we use the structure of heuristic solutions (corresponding to good upper bounds) to guide the computation of lower bounds; and the information gained during the computation of lower bounds is used to guide the computation of upper bounds. All in all, there is a mutual dependence between the three major components: reductions, upper bounds, and lower bounds. This is not a drawback, but an advantage: The scenario is that performing (alternative-based) reductions accelerates the computation of upper and lower bounds and enhances their qualities; the information gained during the computation of bounds is used to reduce the instance further (using bound-based reductions), and then the whole pattern repeats. We call this whole process the reduction process, beginning with fast reductions and switching to more and more powerful ones as the process advances. This strategy is not only a major reason for the short solution times our algorithm very often achieves, but also enables instances to be solved that we could not solve in a reasonable time otherwise. Note especially that the value of the lower bound corresponding to a certain relaxation can be enhanced through reductions; this helps to solve instances that otherwise could not be solved (without branching) using the same techniques for computing upper and lower bounds.

For the experimental results given in this chapter, we use the following components: For computing lower bounds, we use the relaxation LP_C through the algorithm DUAL-ASCENT and, in advanced stages, row generation with $LP_{C'}$ (Section 2.9.2) or, if the proportion of terminals to all vertices is high, the Lagrangian relaxation LaP_{T_0} (Section 2.9.1). To reduce the instances, we use all described alternative-based techniques (Section 3.3). In addition, we use the bound-based techniques DA (Section 3.4.2) and, in combination with row-generation, the test RG (Section 3.4.3). Furthermore, we use the extended reduction techniques (Section 3.5) and the partitioning-based reductions (Section 3.6). For computing upper bounds we use our PRUNE heuristics (Sections 4.3, 4.4), including the combination of Steiner trees (Section 4.5). Before starting the time-consuming RG reduction

test, we check with the heuristic mentioned in Section 5.2 whether we can solve the (sub-)instance with the pathwidth approach.

As described above, the fast methods are applied first, with switching to more time-consuming ones only if an instance has not already been solved to optimality. Apart from this general principle, the exact ordering of the components has usually not been critical.

5.3.2 Branch-and-Bound

The reduction process described in the previous subsection is an extremely powerful device, but it is not guaranteed to solve every instance of the problem. To get an exact algorithm, we integrate it into a branch-and-bound framework. But one should not be misled by the name *branch*-and-bound: Branching is something we generally (and often successfully) try to avoid, it is only a safety net in case the reduction process is blocked. This also means that we invest a lot of work in each node of the branch-and-bound tree to keep the tree small, and do not try to gain speed by limiting the work in each node.

We use binary, vertex-oriented forward branching [HRW92]. Both depth-first and best-first search strategies are available in our implementation, with depth-first as default. There are usually not many nodes in our branch-and-bound trees anyway; moreover, only the currently processed node needs to be kept in the main memory.

As the branching variable, we choose the non-terminal with the largest degree in the best available Steiner tree. The intuitive motivation for this choice is an intensification of the search in an area where a good solution has been found (in case of inclusion) and a diversification of the search to other areas (in case of exclusion). This strategy also supports the building of several blocks (biconnected components). It is known [HRW92] that in case several blocks exist, the problem can be solved by solving the instances corresponding to each intermediate block separately, which generally reduces the total running time substantially. Although it usually cannot be assumed that the original instance is not biconnected, this often changes later during the reduction process and after branching. We use this fact frequently in our algorithms: Whenever a more time-consuming part is to be performed, we check whether the graph is biconnected. If this is not the case, we solve the corresponding subinstances separately and transform the information gained back to information for the original instance. Here one can use the following observation to identify the blocks that must be considered:

Lemma 42 Let T be a Steiner tree with all leaves being terminals in a network G. A block of G is intermediate if and only if it contains an edge of T.

Proof: For a block B to be intermediate, there must be two terminals z_k and z_l such that every path between z_k and z_l contains an edge in B. Hence, every Steiner tree must contain an edge in B. Conversely, consider a Steiner tree T with all leaves being terminals that contains an edge in a block B. So there are two terminals z_k and z_l such that at least one path between z_k and z_l contains an edge in B. If z_k (or z_l) is in B and it is not an articulation point, B is obviously intermediate. Otherwise there must be two articulation points v_i and v_j of B such that a path between z_k and v_i and a path between v_j and z_l contain no edge in B. Now suppose B is not intermediate. Then there is a path between z_k and z_l that does not contain an edge in B. Hence, there is also a path between v_i and v_j that has no edge in B, which contradicts the definition of B as a biconnected component. □

5.4 Summary on Experimental Results

Here we report on what the components presented achieve together, acting as an "orchestra". As stated in Section 1.2, results for different types of instances from the benchmark SteinLib are presented, including the instances of the OR-Library. Here we give only averages for some groups of instances. In the appendix, we report on the results for each instance separately.

We leave it mainly to the reader to compare the given running times to those of other exact algorithms in the literature (see for example [Bea89, BL98, CGR92, Dui93, KM98, Uch01]). As an orientation, we provide in Table 5.1 the average times (in seconds) for the exact solution of some instances groups, which have frequently been used by other authors. It is true that we used the fastest machine (it is approximately half as fast as the PC we used in Section 5.4.1); however improved machine speed explaines our improved times only to a small extend.

instance-group	[Bea89] Cray X-MP	[BL98] SG Indigo	[CGR92] VAX 8700	[Dui93] i486	[KM98] Sun Sparc 20	[Uch01]* UltraSparc II	here Sunfire
D	556	3545	14260	176	117	> 4	0.2
E	—	—	—	—	4415	> 181	1.7
TAQ†	—	—	—	—	197	4	0.08
TAQ	—	—	—	—	—	162	1.1

Table 5.1: Average running times of different exact algorithms.

We solved all instances that (to our knowledge) have been solved before. Furthermore, we solved 32 instances from different instance classes from SteinLib (geometric (TSPFST), VLSI (LIN), wire routing problems (WRP3), constructed difficult instances (I640, PUC, SP) that have not been solved by other authors. See the appendix for details.

5.4.1 Experimental Results on Geometric Instances

As mentioned already in Sections 1.4.2 and 3.6.1 for geometric Steiner problems, an approach based on full Steiner trees (FSTs) has been empirically successful [WWZ00]: In the first phase, the FST generation phase, a set of FSTs is generated that is guaranteed to contain an SMT. In the second phase, the FST concatenation phase, one chooses a subset of the generated FSTs whose concatenation yields an SMT. Empirically, the second phase was usually the bottleneck, where originally methods like backtracking or dynamic programming have been used. A breakthrough occured when Warme [War98] used the fact that FST concatenation can be reduced to finding a minimum spanning tree in a hypergraph whose vertices are the terminals and whose (hyper-)edges correspond to the generated FSTs. We use a different approach: By building the union of (the edge sets of) the FSTs generated in the first phase, we get a normal graph and the FST-concatenation problem is reduced to solving the classical Steiner problem in this graph. Note that in this way the useful information about the individual FSTs is lost. Still, our approach is much faster. In Table 5.2, we compare for the second phase the running times of our program and of the program package GeoSteiner [WWZ01]. For more detailed results on this, see [PV03, PV01d].

*In [Uch01] only running times for 5 of the 40 D/E instances are reported, we estimated the averages by assuming that all other instances were solved in zero time.

†Excluding instances not solved by [KM98].

For both programs all computation was performed on a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor and 1 GB of main memory, using the operating system Linux 2.4.9. We used the gcc 2.96 compiler and CPLEX 7.0 as LP-solver.

instance group	GeoSteiner time (s)	our program time (s)
ES1000FST	150.6	10.3
TSPFST[‡]	261.8	3.0

Table 5.2: Comparison of GeoSteiner (2nd phase) and our program (average running time for exact solution).

Additionally, we are able to solve some instances (fl1400, fl3795, fnl4461) that have not been solved before. In Table 5.3, we show results for instances not solved by GeoSteiner in one day.

instance	size			optimum	our program time (s)
	$\|R\|$	$\|V\|$	$\|E\|$		
es10000	10000	27019	39407	716174280	758
fl1400	1400	2694	4546	17980523	118
fl3795	3795	4859	6539	25529856	139
fnl4461	4461	17127	27352	182361	6148
pcb3038	3038	5829	7552	131895	2.4
pla7397	7397	8790	9815	22481625	0.1

Table 5.3: Instances not solved by GeoSteiner in one day.

5.5 Concluding Remarks

We have presented several algorithmic contributions for solving the Steiner problem in networks. The experimental results strongly recommend the chosen approach based on reductions and underline the utility of the techniques presented here. In particular, the reduction-based heuristics have proven to be extremely strong and robust. Also, the running times of the exact algorithm are often surprisingly small; and for many instances, there is not much room left for improvements. But this is not always the case:

For some instances, fast reductions come to a halt at a time when the relaxations used are still not strong enough; this is for example the case for some of the MC- and I-instances, which were insightfully constructed to fool the currently available techniques. Here, the algorithm has gone into branching to solve the instances exactly. Until new techniques are developed, instances of this type with thousands of vertices will not be solvable in reasonable time. But the results for the other groups of instances seem to indicate that such cases rarely arise naturally.

With respect to instances that were not artificially constructed, even instances with tens of thousands of vertices can usually be solved in reasonable time. Of course, very large instances are still a challenge. There are two main problems:

First, even though the running time and memory consumption for many components can be described by polynomials of low degree, they can grow to a critical value for very large instances. Especially, the $\Theta(rn)$ memory consumption of the DUAL-ASCENT procedure, which is used relatively

[‡]Excluding instances not solved by GeoSteiner.

early in the reduction process, can be a limiting factor. To overcome this problem, the development of new fast reduction techniques could be a solution. If they combine low memory consumption, fast (empirical) execution times and relevant reduction results, they could prepare the ground for currently available reduction techniques.

A second reason is even more decisive. As already discussed in Section 3.6.3, the LP-based approaches rely on the quality of the Linear Programming relaxation used, but any LP-relaxation of polynomial size is bound to have some (integrality) gap in this context. In larger instances, such gaps can accumulate and become more and more relevant. In particular, the important bound-based reductions lose their impact if the available bounds are too weak. There are several approaches to address this:

Branch-and-cut: Currently, we do not use the branch-and-cut approach to improve the quality of the lower bound calculation. The reason is that we found it more advantageous to use the calculated lower bound to reinforce the reduction process, i.e., we use the lower bound and reduced costs for bound-based reductions and then try all reduction tests on the reduced graph. This procedure is often sufficient to solve problem instances without branching. Even if branching is necessary, the whole reduction processes can try to make use of the vertex branching, which is not the case in a branch-and-cut setting, because for many operations performed by reduction tests (e.g., replacement of a vertex by a clique over its neighbors, see Section 3.3.2) there are no methods available to translate them profitably into the linear program maintained by the branch-and-cut algorithm. On the other hand, our approach has the disadvantage that the generated constraints will be discarded after the row generation test (RG) is finished and for the next RG test, they have to be computed from scratch. For medium sized instances this disadvantage is not important, as the row generation is not too time consuming. For larger instances where most of the time is spent solving linear programs, a better integration of the reductions into a row generation or branch-and-cut approach could be an improvement.

Using tighter relaxations: In Section 2.9.3, we showed how to use a tighter linear relaxation efficiently. Other, even tighter relaxations are available in the hierarchy of relaxations, presented in Section 2.8, but currently no technique is at hand to use them in such an efficient way that they could be applied to larger problem instances.

Improving linear relaxations outside the template paradigm: We have already discussed two advanced techniques, the local cuts approach and the graph expansion approach, in the concluding remarks on lower bounds, Section 2.11. A further development of these techniques could enlarge the range of solvable instances.

For geometric instances, we have shown that a combination of full Steiner tree (FST) approach (see the results in the last section) that exploits the geometric properties of the problem and our algorithm for the Steiner problem in general networks is currently the most successful approach. However, it should be possible to improve on this, as a lot of useful information is discarded when the set of FSTs is converted into an instance for the network Steiner tree problem. Exploiting this information in our approach, or integrating our reduction techniques into the FST approach could be fruitful.

Another algorithmic challenge for Steiner tree algorithms is the development of algorithms for reliable computation [Meh02], i.e., algorithms that come with some kind of evidence that they produce correct results. As a formal verification of the whole program is unlikely, certifying algorithms as components would already be an advantage. Such algorithms return additional output (frequently

called a witness) that enables a (fast) verification of the results. A weak version of a witness is mentioned in Section 3.7: After all reductions are performed, the algorithm can transform a tree in the reduced instance into one for the unreduced instance, thus certifying that a tree with the same value exists in the original instance. Of course, this says nothing about the optimality of this tree. Other paradigms of reliable computation, namely "exact arithmetic" and "test and repair" have been mentioned in the context of using linear programming packages for reductions in Section 3.4.3. We list some of the major problems for certifying algorithms in the context of reduction techniques:

- A difficulty arises from a pattern that is frequently used in the efficient implementation of reduction tests: First, in a preprocessing step, some data structures are set up (e.g., representing a Steiner tree produced by some heuristic, a minimum spanning tree $T'_D(R)$ in the distance network, approximations of distances and bottleneck Steiner distances, a lower bound and reduced costs). Then, reduction tests use these data and modify the network. These modifications are done in such a way that the precomputed information is still valid (e.g., the extension of the PT$_m$-test to the case of equality in Section 3.3.1) or that invalid parts are marked such they do not cause harm (e.g., the neighbor lists used for tagging precomputations as invalid if an edge is deleted in Section 3.5.5).

  Even if it was possible to verify that the data structures were correct for the original network, it could be very difficult to verify that they are used correctly after the network is modified. A simple example is the following: When the first edge is deleted by the PT$_m$-test (Section 3.3.1), it is easy to verify that a path exists with Steiner distances not longer than the length of the edge by following the parent pointers in the shortest path tree and the $T'_D(R)$ edges. For edges that are deleted later, this is no longer possible directly, as edges on this path may have been deleted.

- For many certifying algorithms the running times for producing the witness and verification of the result is (at least asymptotically) dominated by the running time for the algorithm. For the presented reduction tests with running time $O(m + n \log n)$ this is not an easy task. Note that one of the major contributions of this work are the techniques for performing reduction tests so fast that they can be applied extensively. As a consequence, if a certifying reduction test is much slower than the original version, it cannot be applied in the same way as before.

  In the example above, checking the edges of the alternative path can take time $O(n)$, while the test condition of the PT$_m$ test can be checked in nearly constant time.

Finally, it is an interesting open question whether the concepts presented in this work can be adapted to other combinatorial optimization problems. Promising candidates for such a recycling of ideas are the reduction-based heuristics (Section 4.3) and extended reduction techniques (Section 3.5) for problems where reduction techniques are already known, and partitioning as a reduction technique (Section 3.6) and graph expansion for improving lower bounds (sketched in Section 2.11) for general optimization problems in networks.

Appendix A

Experimental Results of the Program Package

Here we report on what the components presented achieve together, acting as an "orchestra". In Tables A.2-A.11 we report on each instance of SteinLib, except for the instance groups I080, B, C, P, and ES10FST–ES500FST, which are too easy. Overall we report on 831 instances.

All results were produced by a single run of the same program with the same parameter values, with the exception of the instance groups I and PUC. These instances were constructed with the aim of being difficult for known techniques, thus currently many of them can only be solved using many branching nodes in a branch-and-bound framework. Therefore, we made the solution faster by omitting the time consuming row generation method (Section 2.9.2) in the reduction process. Note that instances that arise naturally from some application can often be solved without branching by using the sophisticated reduction techniques.

For each instance, we give the value of the optimal solution and the total time until the exact solution of the instance (in seconds). We set a time limit of five hours on each run. Within this time, we have solved most of the considered instances. Only 64 instances have not been solved within the time limit. For these instances we give the upper and the lower bound after five hours (in italics). However, 9 of these instances could be solved using stronger extended reductions (see Section 3.5), and 12 could be solved by longer runs. For them we give the time for the exact solution in Tables A.14 and A.15. In total, there are only 43 unsolved instances left. These instances are from the instance groups SP, PUC, and I640, which where constructed with the aim of being difficult for known techniques.

We solved all instances that (to our knowledge) have been solved before. Furthermore, we solved 32 instances from different instance classes from SteinLib that have not been solved by other authors (see Table A.1).

In many cases our program performs several orders of magnitude faster than programs of other authors, see Section 5.4 for a short comparison with others' results. It is acknowledged to be the strongest program for solving general Steiner problem instances at the time being [CD01, SteinLib].

All results in this work were produced single-threaded on a Sunfire 15000 with 900 MHz SPARC III+ CPUs, using the operating system SunOS 5.9. We always used the GNU g++ 2.95.3 compiler with the -O4 flag. As it is a multi-processor machine with shared memory, it is slower than a single processor system with the same processor. A comparison with the running times reported in Section 5.4.1 shows that the machine is approximately half as fast as a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor.

Instance Group	Table	Instances
LIN	A.5	lin31, lin32, lin33, lin34, lin35, lin36, lin37
WRP	A.4	wrp3-83
TSPFST	A.6	fl1400fst, fl3795fst
PUC	A.12	bipe2p, bipe2u, cc5-3p, cc5-3u, hc8p, hc8u
SP	A.13	w3c571
I640	A.11	i640-211, i640-212, i640-213, i640-214, i640-215, i640-321, i640-322, i640-323, i640-324, i640-325, i640-341, i640-342, i640-343, i640-344, i640-345

Table A.1: Instances solved by us and not solved by others.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
d01	1000	1250	5	106	0.1
d02	1000	1250	10	220	0.1
d03	1000	1250	167	1565	0.1
d04	1000	1250	250	1935	0.1
d05	1000	1250	500	3250	0.1
d06	1000	2000	5	67	0.3
d07	1000	2000	10	103	0.3
d08	1000	2000	167	1072	0.1
d09	1000	2000	250	1448	0.1
d10	1000	2000	500	2110	0.1
d11	1000	5000	5	29	0.2
d12	1000	5000	10	42	0.1
d13	1000	5000	167	500	0.1
d14	1000	5000	250	667	0.1
d15	1000	5000	500	1116	0.1
d16	1000	25000	5	13	0.2
d17	1000	25000	10	23	0.2
d18	1000	25000	167	223	0.7
d19	1000	25000	250	310	0.5
d20	1000	25000	500	537	0.1

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
e01	2500	3125	5	111	0.5
e02	2500	3125	10	214	0.3
e03	2500	3125	417	4013	0.1
e04	2500	3125	625	5101	0.1
e05	2500	3125	1250	8128	0.1
e06	2500	5000	5	73	1.2
e07	2500	5000	10	145	1.1
e08	2500	5000	417	2640	0.2
e09	2500	5000	625	3604	0.1
e10	2500	5000	1250	5600	0.1
e11	2500	12500	5	34	0.8
e12	2500	12500	10	67	0.6
e13	2500	12500	417	1280	1.3
e14	2500	12500	625	1732	0.2
e15	2500	12500	1250	2784	0.2
e16	2500	62500	5	15	0.8
e17	2500	62500	10	25	0.5
e18	2500	62500	417	564	21.6
e19	2500	62500	625	758	4.1
e20	2500	62500	1250	1342	0.2

Table A.2: Results on the D and E-instances. Type: Sparse random with varying graph parameters, OR-Library.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
1r111	625	2352	6	28000	0.2
1r112	625	2352	6	28000	0.1
1r113	625	2352	6	26000	0.1
1r121	625	2340	6	36000	0.1
1r122	625	2342	6	45000	0.7
1r123	625	2343	6	40000	0.3
1r131	625	2336	6	43000	0.4
1r132	625	2340	6	37000	0.1
1r133	625	2326	6	36000	0.1
1r211	625	2352	31	77000	0.5
1r212	625	2352	30	81000	0.5
1r213	625	2352	29	70000	0.2
1r221	625	2341	31	79000	0.3
1r222	625	2343	31	68000	0.1
1r223	625	2340	31	77000	0.2
1r231	625	2331	30	80000	0.2
1r232	625	2335	29	86000	0.2
1r233	625	2327	31	71000	0.1
1r311	625	2352	56	108000	0.2
1r312	625	2352	60	113000	0.2
1r313	625	2352	58	106000	0.2
1r321	625	2338	59	118000	0.2
1r322	625	2336	60	113000	0.2
1r323	625	2341	60	117000	0.3
1r331	625	2319	58	103000	0.1
1r332	625	2333	58	109000	0.1
1r333	625	2331	58	113000	0.1

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
2r111	1000	5800	9	28000	1.0
2r112	1000	5800	9	32000	1.0
2r113	1000	5800	9	28000	0.5
2r121	1000	5766	9	28000	0.2
2r122	1000	5772	9	29000	0.5
2r123	1000	5754	9	25000	0.7
2r131	1000	5726	9	27000	0.4
2r132	1000	5725	9	33000	6.6
2r133	1000	5729	9	29000	0.6
2r211	1000	5800	50	89000	384.5
2r212	1000	5800	49	80000	3.7
2r213	1000	5800	48	76000	45.9
2r221	1000	5764	50	83000	4.5
2r222	1000	5765	50	84000	39.7
2r223	1000	5770	49	84000	74.0
2r231	1000	5737	50	86000	51.0
2r232	1000	5733	49	87000	71.4
2r233	1000	5730	47	83000	18.3
2r311	1000	5800	95	129000	70.6
2r312	1000	5800	92	126000	78.9
2r313	1000	5800	97	128000	41.7
2r321	1000	5771	92	125000	2.1
2r322	1000	5753	92	130000	34.0
2r323	1000	5764	96	142000	92.9
2r331	1000	5736	93	134000	26.2
2r332	1000	5745	95	136000	130.7
2r333	1000	5741	98	143000	100.3

Table A.3: Results on the 1R and 2R-instances. Type: 2D (respectively 3D) cross grid graph [Fre97].

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
wrp3-11	128	227	11	1100361	0.1
wrp3-12	84	149	12	1200237	0.1
wrp3-13	311	613	13	1300497	0.5
wrp3-14	128	247	14	1400250	0.1
wrp3-15	138	257	15	1500422	0.1
wrp3-16	204	374	16	1600208	0.2
wrp3-17	177	354	17	1700442	0.1
wrp3-19	189	353	19	1900439	0.1
wrp3-20	245	454	20	2000271	0.3
wrp3-21	237	444	21	2100522	0.2
wrp3-22	233	431	22	2200557	0.5
wrp3-23	132	230	23	2300245	0.1
wrp3-24	262	487	24	2400623	0.7
wrp3-25	246	468	25	2500540	0.2
wrp3-26	402	780	26	2600484	0.5
wrp3-27	370	721	27	2700502	1.2
wrp3-28	307	559	28	2800379	0.3
wrp3-29	245	436	29	2900479	0.2
wrp3-30	467	896	30	3000569	3.6
wrp3-31	323	592	31	3100635	0.8
wrp3-33	437	838	33	3300513	0.6
wrp3-34	1244	2474	34	3400646	397.0
wrp3-36	435	818	36	3600610	4.2
wrp3-37	1011	2010	37	3700485	181.1
wrp3-38	603	1207	38	3800656	17.7
wrp3-39	703	1616	39	3900450	725.6
wrp3-41	178	307	41	4100466	0.8
wrp3-42	705	1373	42	4200598	46.4
wrp3-43	173	298	43	4300457	0.8
wrp3-45	1414	2813	45	4500860	592.5
wrp3-48	925	1738	48	4800552	24.7
wrp3-49	886	1800	49	4900882	131.3
wrp3-50	1119	2251	50	5000673	2769.9
wrp3-52	701	1352	52	5200825	207.7
wrp3-53	775	1471	53	5300847	7.3
wrp3-55	1645	3186	55	*[5500887—5500890]*	
wrp3-56	853	1590	56	5600872	53.3
wrp3-60	838	1763	60	6001164	262.2
wrp3-62	670	1316	62	6201016	68.5
wrp3-64	1822	3610	64	6400931	1549.1
wrp3-66	2521	4858	66	6600922	4483.6
wrp3-67	987	1923	67	6700776	46.4
wrp3-69	856	1621	69	6900841	21.3
wrp3-70	1468	2931	70	7000890	208.1
wrp3-71	1221	2414	71	7101028	248.3
wrp3-73	1890	3613	73	7301207	7104.5
wrp3-74	1019	1941	74	7400759	263.0
wrp3-75	729	1395	75	7501020	11.5
wrp3-76	1761	3370	76	7601028	865.7
wrp3-78	2346	4656	78	7801094	3306.0
wrp3-79	833	1595	79	7900444	28.0
wrp3-80	1491	2831	80	8000849	212.2
wrp3-83	3168	6220	83	*[8300888—8300906]*	
wrp3-84	2356	4547	84	8401094	1243.2
wrp3-85	528	1017	85	8500739	564.8
wrp3-86	1360	2607	86	86000746	677.7
wrp3-88	743	1409	88	88001175	26.4
wrp3-91	1343	2594	91	91000866	265.1
wrp3-92	1765	3613	92	92000764	518.2
wrp3-94	1976	3836	94	94001181	851.2
wrp3-96	2518	4985	96	96001172	3421.5
wrp3-98	2265	4545	98	98001224	3812.8
wrp3-99	2076	4072	99	99001097	1298.3

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
wrp4-11	123	233	11	1100179	0.1
wrp4-13	110	188	13	1300798	0.1
wrp4-14	145	283	14	1400290	0.1
wrp4-15	193	369	15	1500405	0.2
wrp4-16	311	579	16	1601190	0.2
wrp4-17	223	404	17	1700525	0.4
wrp4-18	211	380	18	1801464	0.2
wrp4-19	119	206	19	1901446	0.1
wrp4-21	529	1032	21	2103283	1.5
wrp4-22	294	568	22	2200394	3.0
wrp4-23	257	515	23	2300376	0.9
wrp4-24	493	963	24	2403332	1.8
wrp4-25	422	808	25	2500828	0.7
wrp4-26	396	781	26	2600443	19.5
wrp4-27	243	497	27	2700441	1.2
wrp4-28	272	545	28	2800466	4.2
wrp4-29	247	505	29	2900484	25.6
wrp4-30	361	724	30	3000526	25.0
wrp4-31	390	786	31	3100526	35.7
wrp4-32	311	632	32	3200554	17.8
wrp4-33	304	571	33	3300655	0.6
wrp4-34	314	650	34	3400525	0.8
wrp4-35	471	954	35	3500601	16.2
wrp4-36	363	750	36	3600596	13.2
wrp4-37	522	1054	37	3700647	50.8
wrp4-38	294	618	38	3800606	2.0
wrp4-39	802	1553	39	3903734	3.0
wrp4-40	538	1088	40	4000758	119.5
wrp4-41	465	955	41	4100695	48.7
wrp4-42	552	1131	42	4200701	119.9
wrp4-43	596	1148	43	4301508	3.2
wrp4-44	398	788	44	4401504	30.0
wrp4-45	388	815	45	4500728	2.9
wrp4-46	632	1287	46	4600756	50.9
wrp4-47	555	1098	47	4701318	10.0
wrp4-48	451	825	48	4802220	3.1
wrp4-49	557	1080	49	4901968	7.0
wrp4-50	564	1112	50	5001625	11.3
wrp4-51	668	1306	51	5101616	10.4
wrp4-52	547	1115	52	5201081	4.0
wrp4-53	615	1232	53	5301351	19.2
wrp4-54	688	1388	54	5401534	14.5
wrp4-55	610	1201	55	5501952	13.3
wrp4-56	839	1617	56	5602299	25.6
wrp4-58	757	1493	58	5801466	27.6
wrp4-59	904	1806	59	5901592	6.6
wrp4-60	693	1370	60	6001782	7.8
wrp4-61	775	1538	61	6102210	2.6
wrp4-62	1283	2493	62	6202100	30.8
wrp4-63	1121	2227	63	6301479	793.4
wrp4-64	632	1281	64	6401996	7.5
wrp4-66	844	1691	66	6602931	18.6
wrp4-67	1518	3060	67	6702800	82.3
wrp4-68	917	1850	68	6801753	40.0
wrp4-69	574	1165	69	6902328	7.2
wrp4-70	637	1269	70	7003022	2.0
wrp4-71	802	1609	71	7102320	4.2
wrp4-72	1151	2274	72	7202807	73.0
wrp4-73	1898	3616	73	7302643	284.5
wrp4-74	802	1620	74	7402046	38.4
wrp4-75	938	1869	75	7501712	25.6
wrp4-76	766	1535	76	7602040	21.1

Table A.4: Results on the WRP-instances. Type: Wire routing problems from industry [ZR00]. Instances not solved here could be solved using stronger reductions, see Table A.14.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
alue2087	1244	1971	34	1049	0.1
alue2105	1220	1858	34	1032	0.1
alue3146	3626	5869	64	2240	0.4
alue5067	3524	5560	68	2586	0.9
alue5345	5179	8165	68	3507	3.9
alue5623	4472	6938	68	3413	1.9
alue5901	11543	18429	68	3912	3.3
alue6179	3372	5213	67	2452	0.8
alue6457	3932	6137	68	3057	1.1
alue6735	4119	6696	68	2696	0.9
alue6951	2818	4419	67	2386	0.8
alue7065	34046	54841	544	23881	94.5
alue7066	6405	10454	16	2256	7.3
alue7080	34479	55494	2344	62449	68.3
alue7229	940	1474	34	824	0.1
alut0787	1160	2089	34	982	0.1
alut0805	966	1666	34	958	0.1
alut1181	3041	5693	64	2353	0.5
alut2010	6104	11011	68	3307	1.4
alut2288	9070	16595	68	3843	3.2
alut2566	5021	9055	68	3073	2.5
alut2610	33901	62816	204	12239	95.6
alut2625	36711	68117	879	35459	305.9
alut2764	387	626	34	640	0.1
gap1307	342	552	17	549	0.1
gap1413	541	906	10	457	0.1
gap1500	220	374	17	254	0.1
gap1810	429	702	17	482	0.1
gap1904	735	1256	21	763	0.1
gap2007	2039	3548	17	1104	0.2
gap2119	1724	2975	29	1244	0.2
gap2740	1196	2084	14	745	0.1
gap2800	386	653	12	386	0.1
gap2975	179	293	10	245	0.1
gap3036	346	583	13	457	0.1
gap3100	921	1558	11	640	0.1
gap3128	10393	18043	104	4292	4.3
msm0580	338	541	11	467	0.1
msm0654	1290	2270	10	823	0.1
msm0709	1442	2403	16	884	0.1
msm0920	752	1264	26	806	0.1
msm1008	402	695	11	494	0.1
msm1234	933	1632	13	550	0.1
msm1477	1199	2078	31	1068	0.1
msm1707	278	478	11	564	0.1
msm1844	90	135	10	188	0.1
msm1931	875	1522	10	604	0.1
msm2000	898	1562	10	594	0.1
msm2152	2132	3702	37	1590	0.3
msm2326	418	723	14	399	0.1
msm2492	4045	7094	12	1459	0.4
msm2525	3031	5239	12	1290	0.3
msm2601	2961	5100	16	1440	0.5
msm2705	1359	2458	13	714	0.1
msm2802	1709	2963	18	926	0.1
msm2846	3263	5783	89	3135	0.8
msm3277	1704	2991	12	869	0.1
msm3676	957	1554	10	607	0.1
msm3727	4640	8255	21	1376	0.7
msm3829	4221	7255	12	1571	1.8
msm4038	237	390	11	353	0.1
msm4114	402	690	16	393	0.1
msm4190	391	666	16	381	0.1
msm4224	191	302	11	311	0.1
msm4312	5181	8893	10	2016	3.7
msm4414	317	476	11	408	0.1
msm4515	777	1358	13	630	0.1
taq0014	6466	11046	128	5326	2.9
taq0023	572	963	11	621	0.1
taq0365	4186	7074	22	1914	0.9
taq0377	6836	11715	136	6393	6.2
taq0431	1128	1905	13	897	0.2
taq0631	609	932	10	581	0.1
taq0739	837	1438	16	848	0.1
taq0741	712	1217	16	847	0.1
taq0751	1051	1791	16	939	0.2
taq0891	331	560	10	319	0.1
taq0903	6163	10490	130	5099	5.5
taq0910	310	514	17	370	0.1
taq0920	122	194	17	210	0.1
taq0978	777	1239	10	566	0.1

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
diw0234	5349	10086	25	1996	1.6
diw0250	353	608	11	350	0.1
diw0260	539	985	12	468	0.1
diw0313	468	822	14	397	0.1
diw0393	212	381	11	302	0.1
diw0445	1804	3311	33	1363	0.1
diw0459	3636	6789	25	1362	0.2
diw0460	339	579	13	345	0.1
diw0473	2213	4135	25	1098	0.1
diw0487	2414	4386	25	1424	0.2
diw0495	938	1655	10	616	0.1
diw0513	918	1684	10	604	0.1
diw0523	1080	2015	10	561	0.1
diw0540	286	465	10	374	0.1
diw0559	3738	7013	18	1570	0.5
diw0778	7231	13727	24	2173	1.1
diw0779	11821	22516	50	4440	8.0
diw0795	3221	5938	10	1550	1.2
diw0801	3023	5575	10	1587	0.9
diw0819	10553	20066	32	3399	3.3
diw0820	11749	22384	37	4167	6.8
dmxa0296	233	386	12	344	0.1
dmxa0368	2050	3676	18	1017	0.2
dmxa0454	1848	3286	16	914	0.2
dmxa0628	169	280	10	275	0.1
dmxa0734	663	1154	11	506	0.1
dmxa0848	499	861	16	594	0.1
dmxa0903	632	1087	10	580	0.1
dmxa1010	3983	7108	23	1488	0.2
dmxa1109	343	559	17	454	0.1
dmxa1200	770	1383	21	750	0.1
dmxa1304	298	503	10	311	0.1
dmxa1516	720	1269	11	508	0.1
dmxa1721	1005	1731	18	780	0.1
dmxa1801	2333	4137	17	1365	0.5

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
lin01	53	80	4	503	0.1
lin02	55	82	6	557	0.1
lin03	57	84	8	926	0.1
lin04	157	266	6	1239	0.1
lin05	160	269	9	1703	0.1
lin06	165	274	14	1348	0.1
lin07	307	526	6	1885	0.1
lin08	311	530	10	2248	0.1
lin09	313	532	12	2752	0.1
lin10	321	540	20	4132	0.1
lin11	816	1460	10	4280	0.2
lin12	818	1462	12	5250	0.3
lin13	822	1466	16	4609	0.2
lin14	828	1472	22	5824	0.2
lin15	840	1484	34	7145	0.2
lin16	1981	3633	12	6618	0.5
lin17	1989	3641	20	8405	0.7
lin18	1994	3646	25	9714	1.4
lin19	2010	3662	41	13268	1.4
lin20	3675	6709	11	6673	1.6
lin21	3683	6717	20	9143	1.2
lin22	3692	6726	28	10519	2.1
lin23	3716	6750	52	17560	2.9
lin24	7998	14734	16	15076	9.6
lin25	8007	14743	24	17803	12.6
lin26	8013	14749	30	21757	16.3
lin27	8017	14753	36	20678	13.7
lin28	8062	14798	81	32584	119.3
lin29	19083	35636	24	23765	31.7
lin30	19091	35644	31	27684	87.2
lin31	19100	35653	40	*[31436—31726]*	
lin32	19112	35665	53	*[39247—39926]*	
lin33	19177	35730	117	*[56010—56061]*	
lin34	38282	71521	34	*[44337—45123]*	
lin35	38294	71533	45	*[49061—50619]*	
lin36	38307	71546	58	*[53106—56043]*	
lin37	38418	71657	172	*[96421—99701]*	

Table A.5: Results on the VLSI and LIN-instances. Type: Grid graph with holes (not metric) from VLSI design. Instances not solved here could be solved using stronger reductions, see Table A.14.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
es10000fst	27019	39407	10000	716174280	1398.9

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
es1000fst01	2865	4267	1000	230535806	19.7
es1000fst02	2629	3793	1000	227886471	33.4
es1000fst03	2762	4047	1000	227807756	9.3
es1000fst04	2778	4083	1000	230200846	15.0
es1000fst05	2676	3894	1000	228330602	11.1
es1000fst06	2815	4162	1000	231028456	24.3
es1000fst07	2604	3756	1000	230945623	5.7
es1000fst08	2834	4207	1000	230639115	17.6
es1000fst09	2846	4187	1000	227745838	14.1
es1000fst10	2546	3620	1000	229267101	5.5
es1000fst11	2763	4038	1000	231605619	18.8
es1000fst12	2984	4484	1000	230904712	19.2
es1000fst13	2532	3615	1000	228031092	6.7
es1000fst14	2840	4200	1000	234318491	17.1
es1000fst15	2733	3997	1000	229965775	13.6

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
a280fst	314	329	279	2502	0.1
att48fst	139	202	48	30236	0.3
att532fst	1468	2152	532	84009	4.5
berlin52fst	89	104	52	6760	0.1
bier127fst	258	357	127	104284	0.1
d1291fst	1365	1456	1291	481421	0.1
d1655fst	1906	2083	1655	584948	0.1
d198fst	232	256	198	129175	0.1
d2103fst	2206	2272	2103	769797	0.1
d493fst	1055	1473	493	320137	0.7
d657fst	1416	1978	657	471589	2.8
dsj1000fst	2562	3655	1000	17564659	1.9
eil101fst	330	538	101	605	1.5
eil51fst	181	289	51	409	3.4
eil76fst	237	378	76	513	1.1
fl1400fst	2694	4546	1400	17980523	263.2
fl1577fst	2413	3412	1577	19825626	1.4
fl3795fst	4859	6539	3795	25529856	279.7
fl417fst	732	1084	417	10883190	1.3
fnl4461fst	17127	27352	4461	182361	12967.0
gil262fst	537	723	262	2306	0.1
kroA100fst	197	250	100	20401	0.1
kroA150fst	389	562	150	25700	0.9
kroA200fst	500	714	200	28652	0.4
kroB100fst	230	313	100	21211	0.1
kroB150fst	420	619	150	25217	0.6
kroB200fst	480	670	200	28803	0.7
kroC100fst	244	337	100	20492	0.1
kroD100fst	216	288	100	20437	0.1
kroE100fst	226	306	100	21245	0.1

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
lin105fst	216	323	105	13429	0.1
lin318fst	678	1030	318	39335	0.6
linhp318fst	678	1030	318	39335	0.6
nrw1379fst	5096	8105	1379	56207	207.6
p654fst	777	867	654	314925	0.1
pcb1173fst	1912	2223	1173	53301	0.1
pcb3038fst	5829	7552	3038	131895	2.9
pcb442fst	503	531	442	47675	0.1
pla7397fst	8790	9815	7397	22481625	0.2
pr1002fst	1473	1715	1002	243176	0.1
pr107fst	111	110	107	34850	0.1
pr124fst	154	165	124	52759	0.1
pr136fst	196	250	136	86811	0.1
pr144fst	221	285	144	52925	0.1
pr152fst	308	431	152	64323	0.1
pr226fst	255	269	226	70700	0.1
pr2392fst	3398	3966	2392	358989	0.1
pr264fst	280	287	264	41400	0.1
pr299fst	420	500	299	44671	0.1
pr439fst	572	662	439	97400	0.1
pr76fst	168	247	76	95908	0.1
rat195fst	560	870	195	2386	1.3
rat575fst	1986	3176	575	6808	23.6
rat783fst	2397	3715	783	8883	18.1
rat99fst	269	399	99	1225	0.2
rd100fst	201	253	100	764269099	0.1
rd400fst	1001	1419	400	1490972006	1.9
rl11849fst	13963	15315	11849	8779590	0.8
rl1304fst	1562	1694	1304	236649	0.1
rl1323fst	1598	1750	1323	253620	0.1
rl1889fst	2382	2674	1889	295208	0.2
rl5915fst	6569	6980	5915	533226	0.1
rl5934fst	6827	7365	5934	529890	0.2
st70fst	133	169	70	626	0.1
ts225fst	225	224	225	1120	0.1
tsp225fst	242	252	225	356850	0.1
u1060fst	1835	2429	1060	21265372	1.5
u1432fst	1432	1431	1432	1465	0.1
u159fst	184	186	159	390	0.1
u1817fst	1831	1846	1817	5513053	0.1
u2152fst	2167	2184	2152	6253305	0.1
u2319fst	2319	2318	2319	2322	0.1
u574fst	990	1258	574	3509275	0.2
u724fst	1180	1537	724	4069628	0.3
vm1084fst	1679	2058	1084	2248390	0.6
vm1748fst	2856	3641	1748	3194670	7.2

Table A.6: Results on the ES10000, ES1000 and TSP-instances. Type: Rectilinear, derived with `geosteiner` from 10000 (respectively 1000) random points in the plane, respectively instances from TSPLIB.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
berlin52	52	1326	16	1044	0.1
brasil58	58	1653	25	13655	0.1
world666	666	221445	174	122467	0.8

Table A.7: Results on the X-instances. Type: Complete with euclidean weights.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
mc11	400	760	213	11689	0.1
mc13	150	11175	80	92	2.6
mc2	120	7140	60	71	1.7
mc3	97	4656	45	47	5.4
mc7	400	760	170	3417	0.1
mc8	400	760	188	1566	0.1

Table A.8: Results on the MC-instances. Type: Constructed difficult instances.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i160-001	160	240	7	2490	0.1
i160-002	160	240	7	2158	0.1
i160-003	160	240	7	2297	0.1
i160-004	160	240	7	2370	0.1
i160-005	160	240	7	2495	0.1
i160-011	160	812	7	1677	0.1
i160-012	160	812	7	1750	0.1
i160-013	160	812	7	1661	0.1
i160-014	160	812	7	1778	0.1
i160-015	160	812	7	1768	0.3
i160-021	160	12720	7	1352	0.2
i160-022	160	12720	7	1365	0.2
i160-023	160	12720	7	1351	0.2
i160-024	160	12720	7	1371	0.2
i160-025	160	12720	7	1366	0.2
i160-031	160	320	7	2170	0.1
i160-032	160	320	7	2330	0.1
i160-033	160	320	7	2101	0.1
i160-034	160	320	7	2083	0.1
i160-035	160	320	7	2103	0.1
i160-041	160	2544	7	1494	0.1
i160-042	160	2544	7	1486	0.1
i160-043	160	2544	7	1549	0.1
i160-044	160	2544	7	1478	0.1
i160-045	160	2544	7	1554	0.1
i160-101	160	240	12	3859	0.1
i160-102	160	240	12	3747	0.1
i160-103	160	240	12	3837	0.1
i160-104	160	240	12	4063	0.1
i160-105	160	240	12	3563	0.1
i160-111	160	812	12	2869	0.1
i160-112	160	812	12	2924	0.6
i160-113	160	812	12	2866	0.8
i160-114	160	812	12	2989	1.1
i160-115	160	812	12	2937	1.5
i160-121	160	12720	12	2363	0.3
i160-122	160	12720	12	2348	0.2
i160-123	160	12720	12	2355	0.3
i160-124	160	12720	12	2352	0.2
i160-125	160	12720	12	2351	0.2
i160-131	160	320	12	3356	0.1
i160-132	160	320	12	3450	0.1
i160-133	160	320	12	3585	0.1
i160-134	160	320	12	3470	0.1
i160-135	160	320	12	3716	0.1
i160-141	160	2544	12	2549	0.3
i160-142	160	2544	12	2562	1.5
i160-143	160	2544	12	2557	0.6
i160-144	160	2544	12	2607	1.2
i160-145	160	2544	12	2578	0.8

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i160-201	160	240	24	6923	0.1
i160-202	160	240	24	6930	0.1
i160-203	160	240	24	7243	0.1
i160-204	160	240	24	7068	0.1
i160-205	160	240	24	7122	0.1
i160-211	160	812	24	5583	3.1
i160-212	160	812	24	5643	9.3
i160-213	160	812	24	5647	9.1
i160-214	160	812	24	5720	7.3
i160-215	160	812	24	5518	3.5
i160-221	160	12720	24	4729	0.3
i160-222	160	12720	24	4697	0.3
i160-223	160	12720	24	4730	0.3
i160-224	160	12720	24	4721	0.3
i160-225	160	12720	24	4728	0.4
i160-231	160	320	24	6662	0.3
i160-232	160	320	24	6558	0.9
i160-233	160	320	24	6339	0.1
i160-234	160	320	24	6594	0.1
i160-235	160	320	24	6764	0.9
i160-241	160	2544	24	5086	5.6
i160-242	160	2544	24	5106	5.8
i160-243	160	2544	24	5050	3.7
i160-244	160	2544	24	5076	7.6
i160-245	160	2544	24	5084	5.3
i160-301	160	240	40	11816	0.1
i160-302	160	240	40	11497	0.1
i160-303	160	240	40	11445	0.1
i160-304	160	240	40	11448	0.1
i160-305	160	240	40	11423	0.5
i160-311	160	812	40	9135	14.9
i160-312	160	812	40	9052	29.8
i160-313	160	812	40	9159	12.0
i160-314	160	812	40	8941	9.2
i160-315	160	812	40	9086	15.3
i160-321	160	12720	40	7876	0.2
i160-322	160	12720	40	7859	0.3
i160-323	160	12720	40	7876	0.2
i160-324	160	12720	40	7884	0.3
i160-325	160	12720	40	7862	0.7
i160-331	160	320	40	10414	0.1
i160-332	160	320	40	10806	1.9
i160-333	160	320	40	10561	0.1
i160-334	160	320	40	10327	0.1
i160-335	160	320	40	10589	0.3
i160-341	160	2544	40	8331	7.2
i160-342	160	2544	40	8348	28.5
i160-343	160	2544	40	8275	7.6
i160-344	160	2544	40	8307	11.0
i160-345	160	2544	40	8327	16.1

Table A.9: Results on the I160-instances. Type: Incidence networks, constructed with the aim of being difficult for known techniques.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i320-001	320	480	8	2672	0.1
i320-002	320	480	8	2847	0.1
i320-003	320	480	8	2972	0.1
i320-004	320	480	8	2905	0.1
i320-005	320	480	8	2991	0.1
i320-011	320	1845	8	2053	0.5
i320-012	320	1845	8	1997	0.1
i320-013	320	1845	8	2072	1.3
i320-014	320	1845	8	2061	0.3
i320-015	320	1845	8	2059	0.7
i320-021	320	51040	8	1553	1.4
i320-022	320	51040	8	1565	1.4
i320-023	320	51040	8	1549	1.2
i320-024	320	51040	8	1553	1.1
i320-025	320	51040	8	1550	1.1
i320-031	320	640	8	2673	0.1
i320-032	320	640	8	2770	0.1
i320-033	320	640	8	2769	0.1
i320-034	320	640	8	2521	0.1
i320-035	320	640	8	2385	0.1
i320-041	320	10208	8	1707	0.7
i320-042	320	10208	8	1682	0.2
i320-043	320	10208	8	1723	0.3
i320-044	320	10208	8	1681	0.2
i320-045	320	10208	8	1686	0.2
i320-101	320	480	17	5548	0.1
i320-102	320	480	17	5556	0.1
i320-103	320	480	17	6239	0.1
i320-104	320	480	17	5703	0.1
i320-105	320	480	17	5928	0.2
i320-111	320	1845	17	4273	1.9
i320-112	320	1845	17	4213	3.6
i320-113	320	1845	17	4205	2.9
i320-114	320	1845	17	4104	2.4
i320-115	320	1845	17	4238	2.9
i320-121	320	51040	17	3321	1.7
i320-122	320	51040	17	3314	1.4
i320-123	320	51040	17	3332	1.8
i320-124	320	51040	17	3323	1.8
i320-125	320	51040	17	3340	1.8
i320-131	320	640	17	5255	0.6
i320-132	320	640	17	5052	0.1
i320-133	320	640	17	5125	0.1
i320-134	320	640	17	5272	0.1
i320-135	320	640	17	5342	0.1
i320-141	320	10208	17	3606	4.8
i320-142	320	10208	17	3567	3.6
i320-143	320	10208	17	3561	2.1
i320-144	320	10208	17	3512	0.2
i320-145	320	10208	17	3601	3.2

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i320-201	320	480	34	10044	0.1
i320-202	320	480	34	11223	0.1
i320-203	320	480	34	10148	0.4
i320-204	320	480	34	10275	0.3
i320-205	320	480	34	10573	0.1
i320-211	320	1845	34	8039	17.5
i320-212	320	1845	34	8044	20.8
i320-213	320	1845	34	7984	26.6
i320-214	320	1845	34	8046	28.8
i320-215	320	1845	34	8015	113.4
i320-221	320	51040	34	6679	1.8
i320-222	320	51040	34	6686	1.9
i320-223	320	51040	34	6695	1.9
i320-224	320	51040	34	6694	1.9
i320-225	320	51040	34	6691	1.5
i320-231	320	640	34	9862	1.8
i320-232	320	640	34	9933	5.1
i320-233	320	640	34	9787	0.1
i320-234	320	640	34	9517	0.6
i320-235	320	640	34	9945	2.0
i320-241	320	10208	34	7027	17.0
i320-242	320	10208	34	7072	39.5
i320-243	320	10208	34	7044	20.4
i320-244	320	10208	34	7078	30.2
i320-245	320	10208	34	7046	16.8
i320-301	320	480	80	23279	0.6
i320-302	320	480	80	23387	0.2
i320-303	320	480	80	22693	0.9
i320-304	320	480	80	23451	1.4
i320-305	320	480	80	22547	0.5
i320-311	320	1845	80	17945	5826.6
i320-312	320	1845	80	*[17609—18122]*	
i320-313	320	1845	80	17991	12932.8
i320-314	320	1845	80	*[17542—18108]*	
i320-315	320	1845	80	*[17454—17987]*	
i320-321	320	51040	80	15648	38.3
i320-322	320	51040	80	15646	72.6
i320-323	320	51040	80	15654	32.9
i320-324	320	51040	80	15667	146.5
i320-325	320	51040	80	15649	51.2
i320-331	320	640	80	21517	23.9
i320-332	320	640	80	21674	2.9
i320-333	320	640	80	21339	19.8
i320-334	320	640	80	21415	5.5
i320-335	320	640	80	21378	14.3
i320-341	320	10208	80	16296	2404.3
i320-342	320	10208	80	16228	88.2
i320-343	320	10208	80	16281	692.3
i320-344	320	10208	80	16295	1178.1
i320-345	320	10208	80	16289	1392.8

Table A.10: Results on the I320-instances. Type: Incidence networks, constructed with the aim of being difficult for known techniques. Instances not solved here could be solved using longer runs, see Table A.15.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i640-001	640	960	9	4033	0.1
i640-002	640	960	9	3588	0.1
i640-003	640	960	9	3438	0.1
i640-004	640	960	9	4000	0.1
i640-005	640	960	9	4006	0.1
i640-011	640	4135	9	2392	0.1
i640-012	640	4135	9	2465	1.2
i640-013	640	4135	9	2399	0.8
i640-014	640	4135	9	2171	0.1
i640-015	640	4135	9	2347	0.1
i640-021	640	204480	9	1749	10.1
i640-022	640	204480	9	1756	10.1
i640-023	640	204480	9	1754	10.2
i640-024	640	204480	9	1751	8.3
i640-025	640	204480	9	1745	10.2
i640-031	640	1280	9	3278	0.1
i640-032	640	1280	9	3187	0.1
i640-033	640	1280	9	3260	0.1
i640-034	640	1280	9	2953	0.1
i640-035	640	1280	9	3292	0.1
i640-041	640	40896	9	1897	5.0
i640-042	640	40896	9	1934	2.3
i640-043	640	40896	9	1931	1.3
i640-044	640	40896	9	1938	2.5
i640-045	640	40896	9	1866	0.9
i640-101	640	960	25	8764	0.2
i640-102	640	960	25	9109	0.1
i640-103	640	960	25	8819	0.1
i640-104	640	960	25	9040	0.2
i640-105	640	960	25	9623	1.0
i640-111	640	4135	25	6167	20.0
i640-112	640	4135	25	6304	21.6
i640-113	640	4135	25	6249	32.9
i640-114	640	4135	25	6308	17.2
i640-115	640	4135	25	6217	21.6
i640-121	640	204480	25	4906	12.1
i640-122	640	204480	25	4911	12.2
i640-123	640	204480	25	4913	12.2
i640-124	640	204480	25	4906	10.7
i640-125	640	204480	25	4920	12.3
i640-131	640	1280	25	8097	3.4
i640-132	640	1280	25	8154	1.6
i640-133	640	1280	25	8021	0.3
i640-134	640	1280	25	7754	0.1
i640-135	640	1280	25	7696	0.6
i640-141	640	40896	25	5199	32.3
i640-142	640	40896	25	5193	34.0
i640-143	640	40896	25	5194	20.5
i640-144	640	40896	25	5205	18.6
i640-145	640	40896	25	5218	39.7

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i640-201	640	960	50	16079	1.0
i640-202	640	960	50	16324	0.1
i640-203	640	960	50	16124	1.1
i640-204	640	960	50	16239	0.1
i640-205	640	960	50	16616	0.8
i640-211	640	4135	50	*[11498—12062]*	
i640-212	640	4135	50	11795	1070.3
i640-213	640	4135	50	11879	1873.9
i640-214	640	4135	50	11898	7554.4
i640-215	640	4135	50	12081	6170.4
i640-221	640	204480	50	9821	109.6
i640-222	640	204480	50	9798	99.5
i640-223	640	204480	50	9811	88.9
i640-224	640	204480	50	9805	13.7
i640-225	640	204480	50	9807	13.7
i640-231	640	1280	50	15014	16.5
i640-232	640	1280	50	14630	10.2
i640-233	640	1280	50	14797	8.8
i640-234	640	1280	50	15203	4.1
i640-235	640	1280	50	14803	59.6
i640-241	640	40896	50	10230	190.3
i640-242	640	40896	50	10195	89.6
i640-243	640	40896	50	10215	122.5
i640-244	640	40896	50	10246	526.8
i640-245	640	40896	50	10223	159.7
i640-301	640	960	160	45005	4.1
i640-302	640	960	160	45736	8.2
i640-303	640	960	160	44922	4.7
i640-304	640	960	160	46233	2.1
i640-305	640	960	160	45902	9.8
i640-311	640	4135	160	*[34622—36005]*	
i640-312	640	4135	160	*[34691—35997]*	
i640-313	640	4135	160	*[34596—35758]*	
i640-314	640	4135	160	*[34532—35727]*	
i640-315	640	4135	160	*[34683—35934]*	
i640-321	640	204480	160	31094	4071.8
i640-322	640	204480	160	31068	2485.9
i640-323	640	204480	160	31080	2606.7
i640-324	640	204480	160	31092	2920.8
i640-325	640	204480	160	31081	2967.7
i640-331	640	1280	160	42796	213.2
i640-332	640	1280	160	42548	3636.1
i640-333	640	1280	160	42345	1221.8
i640-334	640	1280	160	42768	16992.6
i640-335	640	1280	160	43035	3761.3
i640-341	640	40896	160	*[31842—32089]*	
i640-342	640	40896	160	*[31867—31978]*	
i640-343	640	40896	160	*[31801—32015]*	
i640-344	640	40896	160	*[31799—31998]*	
i640-345	640	40896	160	*[31783—31995]*	

Table A.11: Results on the I640-instances. Type: Incidence networks, constructed with the aim of being difficult for known techniques. Instances i640-211, i640-34[1-5] could be solved using longer runs, see Table A.15.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
cc10-2p	1024	5120	135	*[34133—35687]*	
cc10-2u	1024	5120	135	*[331—345]*	
cc11-2p	2048	11263	244	*[61773—64366]*	
cc11-2u	2048	11263	244	*[600—620]*	
cc12-2p	4096	24574	473	*[117941—122925]*	
cc12-2u	4096	24574	473	*[1144—1197]*	
cc3-10p	1000	13500	50	*[12173—12964]*	
cc3-10u	1000	13500	50	*[115—127]*	
cc3-11p	1331	19965	61	*[14883—15816]*	
cc3-11u	1331	19965	61	*[140—154]*	
cc3-12p	1728	28512	74	*[17947—19011]*	
cc3-12u	1728	28512	74	*[171—187]*	
cc3-4p	64	288	8	2338	10.6
cc3-4u	64	288	8	23	7.8
cc3-5p	125	750	13	3661	447.6
cc3-5u	125	750	13	36	636.3
cc5-3p	243	1215	27	*[6773—7299]*	
cc5-3u	243	1215	27	*[66—71]*	
cc6-2p	64	192	12	3271	2.2
cc6-2u	64	192	12	32	5.0
cc6-3p	729	4368	76	*[19847—20456]*	
cc6-3u	729	4368	76	*[194—199]*	
cc7-3p	2187	15308	222	*[54694—57459]*	
cc7-3u	2187	15308	222	*[531—554]*	
cc9-2p	512	2304	64	*[16520—17451]*	
cc9-2u	512	2304	64	*[161—172]*	

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
bip42p	1200	3982	200	*[24364—24688]*	
bip42u	1200	3982	200	*[232—237]*	
bip52p	2200	7997	200	*[24180—24823]*	
bip52u	2200	7997	200	*[230—235]*	
bip62p	1200	10002	200	*[22436—22959]*	
bip62u	1200	10002	200	*[214—221]*	
bipa2p	3300	18073	300	*[34671—35905]*	
bipa2u	3300	18073	300	*[330—341]*	
bipe2p	550	5013	50	5616	3328.0
bipe2u	550	5013	50	54	3674.3
hc10p	1024	5120	512	*[59202—60679]*	
hc10u	1024	5120	512	*[568—581]*	
hc11p	2048	11264	1024	*[117360—120471]*	
hc11u	2048	11264	1024	*[1126—1160]*	
hc12p	4096	24576	2048	*[232849—241286]*	
hc12u	4096	24576	2048	*[2233—2304]*	
hc6p	64	192	32	4003	27.7
hc6u	64	192	32	39	13.5
hc7p	128	448	64	7905	14362.7
hc7u	128	448	64	77	17253.5
hc8p	256	1024	128	*[15103—15327]*	
hc8u	256	1024	128	*[146—148]*	
hc9p	512	2304	256	*[29866—30310]*	
hc9u	512	2304	256	*[287—292]*	

Table A.12: Results on the PUC-instances. Type: Constructed difficult instances: hypercubes, from code covering, and bipartite graphs [RdAR$^+$01].

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
antiwheel5	10	15	5	7	0.1
design432	8	20	4	9	0.1
oddcycle3	6	9	3	4	0.1
oddwheel3	7	9	4	5	0.1
se03	13	21	4	12	0.1
w13c29	783	2262	406	*[500—508]*	
w23c23	1081	3174	552	*[684—694]*	
w3c571	3997	10278	2284	2854	15910.5

Table A.13: Results on the SP-instances. Type: Constructed difficult instances, combination of odd wheels and odd circles, difficult for Linear Programming approaches

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
lin31	19100	35653	40	31696	1002
lin32	19112	35665	53	39832	3559
lin33	19177	35730	117	56061	1416
lin34	38282	71521	34	45018	9144
lin35	38294	71533	45	50559	8194
lin36	38307	71546	58	55608	763693
lin37	38418	71657	172	99560	297795
wrp3-55	1645	3186	55	5500888	15569
wrp3-83	3168	6220	83	8300906	115224

Table A.14: Instances solved using stronger extended reductions.

Instance	Size			Opt	Time
	$\|V\|$	$\|E\|$	$\|R\|$		
i320-312	320	1845	80	18122	33542
i320-314	320	1845	80	18088	45856
i320-315	320	1845	80	17987	24918
i640-211	640	4135	50	11984	67237
i640-341	640	40896	160	32042	402451
i640-342	640	40896	160	31978	18682
i640-343	640	40896	160	32015	99206
i640-344	640	40896	160	31991	77280
i640-345	640	40896	160	31994	68199
cc5-3p	243	1215	27	7299	41856
cc5-3u	243	1215	27	71	220668
hc8p	256	1024	128	15322	400071
hc8u	256	1024	128	148	511646

Table A.15: Instances solved in longer runs.

Summary

The Steiner problem in networks is the problem of connecting a set of required vertices in a weighted graph at minimum cost. This is a classical $\mathcal{NP}$-hard problem with many important applications in network design in general and VLSI layout in particular. The primary goal of our research has been the development of empirically successful algorithms. This means we designed and implemented algorithms that

1. generate Steiner trees of low cost in reasonable running times (upper bounds),

2. prove the quality of a Steiner tree by providing a lower bound on the optimal value (lower bounds),

3. or find an optimal Steiner tree (exact algorithms).

4. As an important prerequisite for the first three tasks, we used preprocessing techniques to reduce the size of the original problem without changing the optimal solution (reduction tests).

The value of our algorithms is measured by comparing our results to those of other research groups on the huge and well-established benchmark library for Steiner tree problems, SteinLib. In the case of exact algorithms, this measure is the best one can get because of the $\mathcal{NP}$-hardness result. For upper and lower bounds, the extensive experimental evaluation gives much sharper and much more relevant information than a worst-case analysis.

In the following, we will give a list of our main contributions:

Lower Bounds:

- There are many (mixed) integer programming formulations of the Steiner problem. The corresponding linear programming relaxations are of great interest particularly, but not exclusively, for computing lower bounds, but not much was known about the relative quality of these relaxations. We compare the linear relaxations of all classical, frequently cited integer programming formulations of this problem from a theoretical point of view with respect to their optimal values. We present several new results, establishing very clear relations between relaxations, which have often been treated as unrelated or incomparable, forming a **hierarchy of relaxations**.
- We introduce a **collection of new relaxations** that are stronger than any known relaxation that can be solved in polynomial time, and place the new relaxations into our hierarchy. Further, we show how such a relaxation can be used in practical algorithms. Except for the flow-balance constraints introduced by Koch and Martin, this is the first successful attempt to use a relaxation that is stronger than Wong's directed cut relaxation from 1984.

Preprocessing/Reduction techniques:

- For some of the classical reduction tests, which would have been too time-consuming for large instances in their original form, we design **efficient realizations**, improving the worst-case running time to $O(m + n \log n)$ in many cases. Furthermore, we design new tests, filling some of the gaps left by the classical tests.

- Previous reduction tests were either alternative based or bound based. That means to simplify the problem they either argued with the existence of alternative solutions, or they used some constrained lower bound and upper bound. We develop a framework for **extended reduction tests**, which extends the scope of inspection of reduction tests to larger patterns and combines for the first time alternative-based and bound-based approaches effectively.
- We introduce the new concept of **partitioning-based reduction techniques**, which has a significant impact on the reduction results in some cases.
- We integrate all tests into a **reduction packet**, which performs stronger reductions than any other package we are aware of. Additionally, the reduction results of other packages can be achieved typically in a fraction of the running time.

Upper Bounds:

- We present variants of known path heuristics, including an empirically fast variant with a fast worst-case running time of $O(m + n \log n)$. The previous running time for this kind of path heuristic was $O(rm \log n)$.
- We introduce a new meta-heuristic, **reduction-based heuristics**. On the basis of this concept, we develop heuristics that achieve typically sharper upper bounds than the strongest known heuristics for this problem despite running times that are smaller by orders of magnitude.

Exact Algorithm:

We integrate the previously mentioned building blocks into an exact algorithm that achieves very good running times. Additionally, we present a procedure that uses the fixed-parameter tractability of the Steiner problem for subgraphs of small width.

- For most benchmark instances the program computes the exact solution in running times that are shorter than the running times of other authors by orders of magnitude.
- There are 73 instances in SteinLib that have not been solved by any other research group. We have been able to solve 32 of them.
- For geometric Steiner problems, our algorithm for general networks is (together with a preprocessing algorithm from Warme, Winter, and Zachariasen that exploits some of the geometric properties) the fastest algorithm and beats the specially tailored MSTH approach, which has received much attention.

From the current experiences one can expect that instances of the Steiner problem with thousands of vertices and edges can be solved in minutes or hours if the instances were not intentionally constructed with the aim of being difficult for the known techniques.

Zusammenfassung

Das Steiner Problem in Netzwerken ist das Problem, eine Menge von Basisknoten in einem gewichteten Graphen kostenminimal zu verbinden. Es ist ein klassisches $\mathcal{NP}$-schweres Problem mit vielen wichtigen Anwendungen in der Netzwerkoptimierung im Allgemeinen und im VLSI-Entwurf im Besonderen. Der Schwerpunkt unserer Arbeit hat in der Entwicklung empirisch erfolgreicher Algorithmen gelegen. Das heißt, wir haben Algorithmen entworfen, implementiert und getestet,

1. die schnell kosteng¨unstige Steinerb¨aume berechnen (obere Schranken),
2. die die Qualit¨at eines Steinerbaums durch die Berechnung einer unteren Schranke f¨ur den Wert einer optimalen L¨osung nachweisen (untere Schranken),
3. oder die einen optimalen Steinerbaum finden (exakte Algorithmen).
4. Als wichtige Voraussetzung f¨ur die ersten drei Aufgaben benutzen wir Vorverarbeitungs (*preprocessing*) Techniken, die die Schwierigkeit einer Probleminstanz reduzieren sollen, ohne die optimale L¨osung zu ver¨andern (Reduktionstests).

F¨ur die Bewertung unserer Algorithmen vergleichen wir unsere Ergebnisse auf der umfangreichen und in der Forschung weithin akzeptierten Benchmark-Bibliothek SteinLib mit denen anderer Forschergruppen. Bez¨uglich exakter Algorithmen ist diese Bewertung wegen der $\mathcal{NP}$-schwere Resultate die bestm¨ogliche. Bez¨uglich oberer und unterer Schranken liefert die ausf¨uhrliche, experimentelle Bewertung sch¨arfere und relevantere Informationen als eine worst-case Analyse.

Im folgenden stellen wir eine Liste der wesentlichen Ergebnisse der Arbeit vor:

Untere Schranken:

- Es gibt viele (gemischt-) ganzzahlige Formulierungen f¨ur das Steiner Problem. Die dazugeh¨origen Linearen Relaxationen sind von großem Interesse insbesondere (aber nicht ausschließlich) f¨ur die Berechnung von unteren Schranken. Allerdings war nur wenig ¨uber die relative Qualit¨at dieser Relaxationen bekannt. Wir vergleichen die Linearen Relaxationen von allen klassischen, h¨aufig zitierten ganzzahligen Formulierungen dieses Problems von einem theoretischen Standpunkt aus bez¨uglich ihrer optimalen Werte. Dabei zeigen wir einige neue Resultate und stellen sehr klare Beziehungen zwischen Relaxationen auf, die bisher oft als unvergleichbar behandelt wurden. Die Beziehungen werden in einer **Hierarchie von Relaxationen** dargestellt.
- Wir stellen eine **Klasse von neuen Relaxationen** vor, die st¨arker ist als alle anderen bekannten Relaxationen, die in Polynomialzeit gel¨ost werden k¨onnen. Wir plazieren die neuen Relaxationen in unserer Hierarchie. Weiter zeigen wir, wie eine Relaxation der Klasse in einem praktischen Algorithmus eingesetzt werden kann. Abgesehen von den Flussgleichgewichts-Nebenbedingungen, die von Koch und Martin eingef¨uhrt wurden, ist dies der erste erfolgreiche Versuch, eine Relaxation algorithmisch zu benutzen, die st¨arker ist als Wongs gerichtete Schnitt-Relaxation von 1984.

Reduktionstechniken:

- F¨ur einige der klassischen Reduktionstests, die in ihrer Originalform f¨ur große Instanzen zu zeitaufw¨andig sind, entwerfen wir **effiziente Realisierungen**. In vielen F¨allen k¨onnen

wir die worst-case Laufzeit auf $O(m + n \log n)$ verbessern. Weiterhin haben wir einige neue Tests entworfen, die einige der L¨ucken schließen, die die bisherigen Tests gelassen hatten.

- Alle bisherigen Reduktionstests waren entweder alternativenbasiert oder schrankenbasiert. Das heißt, um das Problem zu vereinfachen argumentierten sie entweder mit der Existenz (oder Nichtexistenz) von alternativen L¨osungen, oder sie benutzten obere Schranken und (eingeschr¨ankte) untere Schranken. Wir haben ein Ger¨ust f¨ur **erweiterte Reduktionstests** entwickelt, bei dem der Beobachtungsbereich von Reduktionstests auf gr¨oßere Muster erweitert wird und zum ersten Mal alternativenbasierte und schrankenbasierte Ans¨atze effektiv miteinander kombiniert werden.
- Wir entwickeln das neue Konzept der **partitionierungsbasierten Reduktionstechniken**, das in einigen F¨allen deutliche Auswirkung auf die Reduktionsresultate hat.
- Wir verbinden alle Tests zu einem **Reduktionspaket**, das st¨arkere Reduktionen durchf¨uhren kann als alle anderen uns bekannten Pakete. Dar¨uberhinaus lassen sich die Reduktionsergebnisse anderer Pakete typischerweise in einem Bruchteil der Laufzeit erreichen.

Obere Schranken:

- Wir entwickeln Varianten einer bekannten Pfadheuristik, unter anderem eine empirisch schnelle Variante mit einer schnellen worst-case Laufzeit von $O(m + n \log n)$. Die bisherige Laufzeitgarantie f¨ur diese Art von Pfadheuristik war $O(rm \log n)$.
- Wir stellen eine neue Meta-Heuristik vor, **reduktionsbasierte Heuristiken**. Auf der Basis dieses Konzepts entwickeln wir Heuristiken, die typischerweise bessere obere Schranken erzielen als die st¨arksten bekannten Heuristiken und dabei um Gr¨oßenordnungen kleinere Laufzeiten haben.

Exakter Algorithmus:

Wir verbinden die zuvor erw¨ahnten Bausteine zu einem exakten Algorithmus, der sehr gute Laufzeiten erzielt. Dar¨uberhinaus zeigen wir, wie L¨osbarkeit des Steinerproblems bez¨uglich bestimmter Problemparameter (*fixed-parameter tractability*) f¨ur Teilgraphen mit kleiner Breite benutzt werden kann.

- F¨ur die meisten Benchmarkinstanzen berechnet unser Programm die exakten L¨osungen in Laufzeiten, die um Gr¨oßenordnungen k¨urzer sind als die anderer Forschergruppen.
- Von den 73 Instanzen, die bisher von keiner anderen Forschergruppe gel¨ost wurde, konnten wir 32 l¨osen.
- F¨ur geometrische Steinerbaum Probleme ist unser Algorithmus f¨ur Steinerprobleme in allgemeinen Netzwerken (zusammen mit einer Vorverarbeitung von Warme, Winter, und Zachariasen, die die geometrischen Eigenschaften ausnutzt) der schnellste Algorithmus und schl¨agt den speziell f¨ur dieses Problem entwickelten MSTH-Ansatz.

Nach den derzeitigen Erfahrungen ist zu erwarten, dass Instanzen des Steinerproblems mit tausenden von Knoten und Kanten in einem Zeitraum von Minuten oder Stunden gel¨ost werden k¨onnen, wenn die Instanzen nicht speziell mit der Absicht konstruiert wurden, schwer f¨ur die bekannten Methoden zu sein.

Bibliography

[ABCC01] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. TSP cuts which do not conform to the template paradigm. In Michael J¨unger and Denis Naddef, editors, *Computational Combinatorial Optimization*, volume 2241 of *Lecture Notes in Computer Science*. Springer, 2001.

[ACP87] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[Ane80] Y. P. Aneja. An integer linear programming approach to the Steiner problem in graphs. *Networks*, 10:167–178, 1980.

[APV03] E. Althaus, T. Polzin, and S. Vahdati Daneshmand. Improving linear programming approaches for the Steiner tree problem. Research Report MPI-I-2003-1-004, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, February 2003.

[Aro94] S. Arora. *Probabilistic Checking of Proofs and Hardness of Approximation Problems*. PhD thesis, Princeton University, 1994.

[Aro96] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. Technical report, Princeton University, 1996.

[BD93] S. Bhattacharya and B. Dasgupta. Steiner problem in multistage computer networks. In D . Z. Du and P. M. Pardalos, editors, *Network Optimization Problems*, pages 387–401. World Scientific Publishing, 1993.

[Bea84] J. E. Beasley. An algorithm for the Steiner problem in graphs. *Networks*, 14:147–159, 1984.

[Bea89] J. E. Beasley. An SST-based algorithm for the Steiner problem in graphs. *Networks*, 19:1–16, 1989.

[Bea90] J. E. Beasley. OR-Library. `http://graph.ms.ic.ac.uk/info.html`, 1990.

[BKJ83] K. Bharath-Kumar and J. M. Jaffe. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications*, 31:343–351, 1983.

[BL98] J. E. Beasley and A. Lucena. A branch and cut algorithm for the Steiner problem in graphs. *Networks*, 31:39–59, 1998.

[Bod93] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[Bod96] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[Bod02] H. L. Bodlaender. Personal communication, 2002.

[BP75] E. Balas and M. Padberg. On the set-covering problem: II. An algorithm for set partitioning. *Operations Research*, 23:74–90, 1975.

[BP87] A. Balakrishnan and N. R. Patel. Problem reduction methods and a tree generation algorithm for the Steiner network problem. *Networks*, 17:65–85, 1987.

[BP89] M. W. Bern and P. Plassman. The Steiner problem with edge lengths 1 and 2. *Information Processing Letters*, 32:171–176, 1989.

[CD01] X. Cheng and D.-Z. Du, editors. *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*. Kluwer Academic Publishers, Dordrecht, 2001.

[CG97] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[CGR92] S. Chopra, E. R. Gorres, and M. R. Rao. Solving the Steiner tree problem on a graph using branch and cut. *ORSA Journal on Computing*, 4:320–335, 1992.

[Coo02] W. Cook. Personal communication, 2002.

[CR94a] S. Chopra and M. R. Rao. The Steiner tree problem I: Formulations, compositions and extension of facets. *Mathematical Programming*, pages 209–229, 1994.

[CR94b] S. Chopra and M. R. Rao. The Steiner tree problem II: Properties and classes of facets. *Mathematical Programming*, pages 231–246, 1994.

[CS02] W. Cook and P. Seymour. Tour merging via branch-decomposition (draft). `http://www.isye.gatech.edu/~wcook/papers/tmerge.ps`, December 2002.

[CT01] S. Chopra and C.-Y. Tsai. Polyhedral approaches for the Steiner tree problem on graphs. In X. Cheng and D.-Z. Du, editors, *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 175–202. Kluwer Academic Publishers, Dordrecht, 2001.

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

[Dui00] C. W. Duin. Preprocessing the Steiner problem in graphs. In D-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 173–233. Kluwer Academic Publishers, 2000.

[DV87] C. W. Duin and T. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17:353–364, 1987.

[DV89] C. W. Duin and T. Volgenant. Reduction tests for the Steiner problem in graphs. *Networks*, 19:549–567, 1989.

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

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:127–136, 1971.

[Esb95] H. Esbensen. Computing near-optimal solutions to the Steiner problem in a graph using a genetic algorithm. *Networks*, 26:173–185, 1995.

[FG82] L. R. Foulds and R. L. Graham. The Steiner tree problem in phylogeny is NP-complete. *Advances in Applied Mathematics*, 3:43–49, 1982.

[Fre97] Clemens Frey. Heuristiken und genetische algorithmen f¨ur modifizierte Steinerbaumprobleme. Master's thesis, Universit¨at Bayreuth, 1997.

[GB93] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:145–166, 1993.

[GGJ77] M. R. Garey, R. L. Graham, and D. S. Johnson. The complexity of computing Steiner minimal trees. *SIAM Journal on Applied Mathematics*, 32:835–859, 1977.

[GHK+02] C. Gentile, U.-U. Haus, M. K¨oppe, G. Rinaldi, and R. Weismantel. A primal approach to the stable set problem. In R. M¨ohring and R. Raman, editors, *Algorithms - ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 525–537, Rom, Italy, 2002. Springer.

[GHNP01] C. Gr¨opl, S. Hougardy, T. Nierhoff, and H. J. Pr¨omel. Approximation algorithms for the Steiner tree problem in graphs. In X. Cheng and D.-Z. Du, editors, *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 235–280. Kluwer Academic Publishers, Dordrecht, 2001.

[GJ77] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32:826–834, 1977.

[GK98] N. Garg and J. K¨onemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Proc. of the 39th Annual IEEE Computer Society Conference on Foundations of Computer Science*, 1998.

[GK02] N. Garg and R. Khandekar. Fast approximation algorithms for Steiner forest and related problems. (draft), April 2002.

[GM93] M. X. Goemans and Y. Myung. A catalog of Steiner tree formulations. *Networks*, 23:19–28, 1993.

[Goe98] M. X. Goemans. Personal communication, 1998.

[Hak71] S. L. Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1:113–133, 1971.

[Han66] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14:255–265, 1966.

[HKW01] U.-U. Haus, M. K¨oppe, and R. Weismantel. The integral basis method for integer programming. *Mathematical Methods of Operations Research*, 53(3):353–361, 2001.

[HPKS02] S. Hert, T. Polzin, L. Kettner, and G. Sch¨afer. Exp lab – a tool set for computational experiments. Research Report MPI-I-2002-1-004, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, November 2002.

[HRG00] M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.

[HRW92] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1992.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[JKP+02] D. G. Jørgensen, J. Krarup, D. Pisinger, M. Sigurd, P. Winter, and M. Zachariasen. Seminar: Recent research results. `http://www.diku.dk/teaching/2002f/459`, 2002.

[Joh85] D. S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 6:434–451, 1985.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[KKT95] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[KM96] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. Technical report, Konrad-Zuse-Zentrum f¨ur Informationstechnik, Berlin, 1996.

[KM98] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[KMV01] T. Koch, A. Martin, and S. Voß. Steinlib: An updated library on Steiner tree problems in graphs. In X. Cheng and D.-Z. Du, editors, *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 285–326. Kluwer Academic Publishers, Dordrecht, 2001.

[KP95] B. N. Khoury and P. M. Pardalos. An exact branch and bound algorithm for the Steiner problem in graphs. In D. Du and M. Li, editors, *Proceedings of COCOON'95*, volume 959 of *Lecture Notes in Computer Science*, pages 582–590. Springer-Verlag, 1995.

[KPH93] B. N. Khoury, P. M. Pardalos, and D. W. Hearn. Equivalent formulations for the Steiner problem in graphs. In D. Z. Du and P. M. Pardalos, editors, *Network Optimization Problems*, pages 111–123. World Scientific Publishing Co., 1993.

[KPS90] B. Korte, H. J. Pr¨omel, and A. Steger. Steiner trees in VLSI-layout. In B. Korte, L. Lovasz, H. J. Pr¨omel, and A. Schrijver, editors, *Paths, Flows, and VLSI-Layout*, pages 185–214. Springer-Verlag, Berlin, 1990.

[KS90] E. Korach and N. Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded tree-width. Technical Report 632, Technicon - Israel Institute of Technology, Computer Science Department, Haifa, Israel, 1990.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Chichester, England, 1990.

[Lev71] A. J. Levin. Algorithm for shortest connection of a group of graph vertices. *Soviet Math. Doklady*, 12:1477–1481, 1971.

[Liu90] W. Liu. A lower bound for the Steiner tree problem in directed graphs. *Networks*, 20:426–434, 1990.

[Mac87] N. Maculan. The Steiner problem in graphs. *Annals of Discrete Mathematics*, 31:185–212, 1987.

[Meh88] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[Meh02] K. Mehlhorn. The reliable algorithmic software challenge (RASC). `http://www.mpi-sb.mpg.de/mehlhorn/ftp/RASC.pdf`, 2002.

[Mel61] Z. A. Melzak. On the problem of Steiner. *Canad. Math. Bull.*, 4:143–148, 1961.

[Mit96] J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: Part II – A simple polynomial-time approximation scheme for geometric k-MST, TSP and related problems. Technical report, Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, 1996.

[MTZ60] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329, 1960.

[MW95] T. L. Magnanti and L. A. Wolsey. Optimal Trees. In M. O. Ball et al., editors, *Handbooks in Operations Research and Management Science*, volume 7, chapter 9. Elsevier Science, 1995.

[NI92] N. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[NK94] R. Novak and G. Kandus. Adaptive Steiner tree balancing in distributed algorithm for multicast connection setup. *Microprocessing and Microprogramming*, 40:795–798, 1994.

[PR00] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. In *Automata, Languages and Programming*, pages 49–60, 2000.

[PT01] A. P¨onitz and P. Tittmann. Computing network reliability in graphs of restricted pathwidth. Technical report, Hochschule Mittweida, 2001.

[PV97] T. Polzin and S. Vahdati Daneshmand. Algorithmen f¨ur das Steiner-Problem. Master's thesis, Universit¨at Dortmund, 1997.

[PV00] T. Polzin and S. Vahdati Daneshmand. Primal-Dual Approaches to the Steiner Problem. In K. Jansen and S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 214–225, 2000.

[PV01a] T. Polzin and S. Vahdati Daneshmand. A comparison of Steiner tree relaxations. *Discrete Applied Mathematics*, 112:241–261, 2001.

[PV01b] T. Polzin and S. Vahdati Daneshmand. Extending reduction techniques for the Steiner tree problem: A combination of alternative- and bound-based approaches. Research Report MPI-I-2001-1-007, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, 2001.

[PV01c] T. Polzin and S. Vahdati Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112:263–300, 2001.

[PV01d] T. Polzin and S. Vahdati Daneshmand. On Steiner trees and minimum spanning trees in hypergraphs. Research Report MPI-I-2001-1-005, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, 2001.

[PV01e] T. Polzin and S. Vahdati Daneshmand. Partitioning techniques for the Steiner problem. Research Report MPI-I-2001-1-006, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, 2001.

[PV02a] T. Polzin and S. Vahdati Daneshmand. Extending reduction techniques for the Steiner tree problem. In R. M¨ohring and R. Raman, editors, *Algorithms - ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 795–807, Rom, Italy, 2002. Springer.

[PV02b] T. Polzin and S. Vahdati Daneshmand. Using (sub)graphs of small width for solving the Steiner problem. Research Report MPI-I-2002-1-001, Max-Planck-Institut f¨ur Informatik, Stuhlsatzenhausweg 85, 66123 Saarbr¨ucken, Germany, 2002.

[PV03] T. Polzin and S. Vahdati Daneshmand. On Steiner trees and minimum spanning trees in hypergraphs. *Operations Research Letters*, 31(1):12–20, 2003.

[PW02] M. Poggi de Aragão and R. F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 1–15, 2002.

[RdAR+01] I. Rosseti, M. P. de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. New benchmark instances for the Steiner problem in graphs. In *Extended Abstracts of the 4th Metaheuristics International Conference (MIC'2001)*, pages 557–561, Porto, 2001.

[Rei91] G. Reinelt. TSPLIB —a traveling salesman problem library. *ORSA Journal on Computing*, 3:376 – 384, 1991.

[R¨oh98] H. R¨ohrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut f¨ur Informatik, Saarbr¨ucken, 1998.

[RS91] N. Robertson and P. D. Seymour. Graph minors – X: Obstructions to tree-decompositions. *J. Comb. Theory Series B*, 52:153–190, 1991.

[RUW02] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid grasp with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing*, 14(3):228–246, 2002.

[RZ00] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 770–779, 2000.

[SteinLib] SteinLib. `http://elib.zib.de/steinlib`, 1997. T. Koch, A. Martin, and S. Voß.

[Tar79] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26:690–715, 1979.

[Tho97] Mikkel Thorup. Undirected single source shortest path in linear time. In *IEEE Symposium on Foundations of Computer Science*, pages 12–21, 1997.

[TM80] H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica*, 24:573–577, 1980.

[Uch01] E. Uchoa. *Algoritmos Para Problemas de Steiner com Aplicações em Projeto de Circuitos VLSI (in Portuguese)*. PhD thesis, Departamento De Informática, PUC-Rio, Rio de Janeiro, April 2001.

[UdAR99] E. Uchoa, M. P. de Aragão, and C. C. Ribeiro. Preprocessing Steiner problems from VLSI layout. Technical Report MCC. 32/99, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil, 1999.

[UdAR02] E. Uchoa, M. P. de Aragão, and C. C. Ribeiro. Preprocessing Steiner problems from VLSI layout. *Networks*, 40:38–50, 2002.

[Ver96] M. G. A. Verhoeven. *Parallel Local Search*. PhD thesis, Eindhoven University of Technology, 1996.

[VJ83] T. Volgenant and R. Jonker. The symmetric traveling salesman problem and edge exchanges in minimal 1-trees. *European Journal of Operational Research*, 12:394–403, 1983.

[Voß92] S. Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40:45–72, 1992.

[War98] D. M. Warme. *Spanning Trees in Hypergraphs with Applications to Steiner Trees*. PhD thesis, University of Virginia, 1998.

[Win95] P. Winter. Reductions for the rectilinear Steiner tree problem. *Networks*, 26:187–198, 1995.

[Won84] R. T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287, 1984.

[WS92] P. Winter and J. MacGregor Smith. Path-distance heuristics for the Steiner problem in undirected networks. *Algorithmica*, 7:309–327, 1992.

[WWZ00] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, 2000.

[WWZ01] D. M. Warme, P. Winter, and M. Zachariasen. GeoSteiner 3.1. `http://www.diku.dk/geosteiner/`, 2001.

[Zac01] M. Zachariasen. The rectilinear Steiner tree problem: A tutorial. In X. Cheng and D.-Z. Du, editors, *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 467–507. Kluwer Academic Publishers, Dordrecht, 2001.

[Zel93] A. Z. Zelikovsky. A faster approximation algorithm for the Steiner tree problem in graphs. *Information Processing Letters*, 46:79–83, 1993.

[ZR00] M. Zachariasen and A. Rohe. Rectilinear group Steiner trees and applications in VLSI design. Technical Report 00906, Institute for Discrete Mathematics, 2000.