Algorithmic Approaches to the Steiner Problem in Networks

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Diplom-Informatiker Siavash Vahdati Daneshmand
aus Ladenburg

Mannheim, 2003

Dekan: Professor Dr. Jürgen Potthoff, Universität Mannheim
Referent: Professor Dr. Matthias Krause, Universität Mannheim
Korreferent: Professor Dr. Ingo Wegener, Universität Dortmund

Tag der mündlichen Prüfung: 23. Februar 2004

Abstract

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 and a fundamental problem in network design with many practical applications. We approach this problem by different means: Relaxations, which relax the feasibility constraints, to get close to an optimal solution; heuristics to find good, but not necessarily optimal solutions; and reductions to simplify problem instances without abandoning the optimal solution. In each case, we study and improve existing methods, introduce new ones, and evaluate them experimentally. We integrate these components into an exact algorithm, which represents the state of the art for the optimal solution of this problem. Many of the presented methods could also be useful for similar problems.

Kurzzusammenfassung

Das Steiner-Problem in Netzwerken ist das Problem, in einem gewichteten Graphen eine gegebene Menge von Knoten kostenminimal zu verbinden. Es ist ein klassisches $\mathcal{NP}$-schweres Problem und ein fundamentales Problem bei der Netzwerkoptimierung mit vielen praktischen Anwendungen. Wir nehmen dieses Problem mit verschiedenen Mitteln in Angriff: Relaxationen, die die Zul¨assigkeitsbedingungen lockern, um eine optimale L¨osung ann¨ahern zu k¨onnen; Heuristiken, um gute, aber nicht garantiert optimale L¨osungen zu finden; und Reduktionen, um die Probleminstanzen zu vereinfachen, ohne eine optimale L¨osung zu zerst¨oren. In allen F¨allen untersuchen und verbessern wir bestehende Methoden, stellen neue vor und evaluieren sie experimentell. Wir integrieren diese Bausteine in einen exakten Algorithmus, der den Stand der Algorithmik f¨ur die optimale L¨osung dieses Problems darstellt. Viele der vorgestellten Methoden k¨onnen auch f¨ur verwandte Probleme von Nutzen sein.

Acknowledgments

I wish to thank Prof. Matthias Krause and Prof. Ingo Wegener for their support.

Words cannot express my thanks to my dear friend and colleague Tobias Polzin who accompanied me for many years on my academic path. Many of the results presented here were produced in joint work with him.

I gratefully acknowledge the financial support provided by the Deutsche Forschungsgemeinschaft. [1]

[1] DFG-Sachbeihilfe Kr 1521/6-1 im Rahmen des DFG-Schwerpunktprogramms 1126: Algorithmik großer und komplexer Netzwerke.

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 and a fundamental problem in network design with many practical applications.

We approach this problem by different means: Relaxations, which relax the feasibility constraints, to get close to an optimal solution; heuristics to find good, but not necessarily optimal solutions; and reductions to simplify problem instances without abandoning the optimal solution. In each case, we study and improve existing methods, introduce new ones, and evaluate them experimentally. We integrate these components into an exact algorithm, which represents the state of the art for the optimal solution of this problem. Many of the presented methods could also be useful for similar problems.

This work summarizes our research on this subject, which has been partly presented at conferences (APPROX, ESA, WEA) and published in journals (Discrete Applied Mathematics, Operations Research Letters). It has already received considerable recognition in the research community, visible in citations in recently published literature [RUW02, PW02, UdAR02, BKM01, Uch01, KMV01, CT01] and university lectures [JKP+02].

The implementation and most of the results presented were produced jointly with Tobias Polzin [PV00a, PV01a, PV01c, PV01e, PV02a, PV02b, PV03, APV03a]. I declare that my contribution constitutes at least half of this work.

1.1.1 Motivation

The study of the Steiner problem is motivated by its central role in network design [CD01, HRW92, MW95] and by its numerous practical applications (see Section 1.3.3). New theoretical results on this problem lead automatically to corresponding results for many other problems [HRS00, JMS03, KKPS00], and the analysis methods can be used for related problems (see for example Section 2.8.1).

From an algorithmic point of view, classical $\mathcal{NP}$-hard combinatorial optimization problems like the traveling salesman problem (TSP) or the Steiner problem have always served as “engines-of-discovery” for new methods that can also be applied to other problems. The outcome has been such general concepts as the cutting-plane method, which was introduced in the TSP context [ABCC03].

Why do we try to solve such problems exactly? Of course, we cannot deny that the challenge is tempting: These prominent $\mathcal{NP}$-hard problems have attracted researchers as test environments for their methods for decades. As a consequence, all relevant techniques in the fields of operations research and combinatorial optimization have been tried on these problems, leading to a respected competition for the best results. Furthermore, our ability to solve many non-trivial problem instances exactly helps us (and also other researchers) to design or evaluate heuristics and relaxations. On the other hand, the results of our exact algorithm set a high benchmark, such that many popular heuristics cannot be motivated by their results anymore (alternative justifications could be ease of design or implementation). Additionally, to get a successful exact algorithm we have to design extremely effective and efficient techniques for computing bounds and reducing the instances, so afterwards each (combination) of these components can be used, as adequate to the requirements. Finally, we even profit from the exact algorithms as a part of each of these components (see Section 5.3.1).

Should one try the same enterprise for every (hard) combinatorial optimization problem one encounters? Even if this were possible, the effort would probably not be justified. Nevertheless, it is instructive to see by the example of these prominent problems how far the limits can be pushed.

1.1.2 Experimental Studies in this Work

The primary aim of this work has not been delivering new complexity results, which, for example, enlarge the knowledge about (non-) approximability of the problem (on certain classes of instances). We do go into depth, presenting many proofs and using sophisticated algorithmic techniques, but always with the aim to improve the practical performance of the algorithms, preferably contributing to the (fast) solution of instances of a size or type that were beyond the reach before. We wish to stress that not the mere solution of such instances really matters, but the fact that to achieve this, one has to come up with new methods that perform well in practice.

In this work, algorithms are regularly evaluated by experimental studies. 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 behavior of the algorithm. In addition, we consider the comparability of results a critical issue, which strongly suggests using benchmark instances. For the Steiner problem in networks, the well-established benchmark library SteinLib [Ste97, KMV01] meanwhile contains over a thousand instances of many different types, contributed over the years by different researchers, sometimes from practical applications (see Appendix A for a description of the instance groups). Since giving experimental results for all these instances in each section would make the work unreasonably long, we have chosen a compromise option: In each chapter (for example for upper bounds or reductions), we give average results on each group of the problem instances from SteinLib (if the gap to the optimal solution value is measured, we restrict ourselves to those groups where all optimal values are known). For the final results of the complete exact algorithm, however, we additionally give results on single instances of SteinLib in the Appendix A. (We leave aside only those groups that nowadays can be considered as “too easy”.) The tables in this work reflect thousands of individual runs; we used ExpLab [HPKS02] and CVS [CVS03] to address the issue of reproducibility of experiments. Unless stated otherwise, all tests were performed on a SPARC III+ 900 MHz processor in a Sunfire 15000 (see Appendix A for a description of the computing environment). 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.

1.1.3 List of Main Contributions and Structure of the Thesis

In the following, we will give a list of our main contributions and where they can be found.

Relaxations and Lower Bounds (Chapter 2)

- 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 val-

ues. We present several new results, establishing very clear relations between relaxations which have often been treated as unrelated or incomparable, and construct 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 derive a practical version of such a relaxation and (later) show how it can optimized efficiently.

- For geometric Steiner problems, the state-of-the-art algorithms follow a two-phase approach: A first phase uses the geometric properties to produce the input for a second phase that is independent of the underlying geometry. The bottleneck of this approach has usually been the second phase, where the hitherto most successful approach is based upon an LP relaxation of the minimum spanning tree in hypergraph (MSTH) problem. We study this original and some new relaxations of the MSTH problem and show that they are all equivalent. We also clarify their relations to the relaxations of the Steiner problem. Later (in Section 5.4.1) we will see that our program outperforms the currently best MSTH-based algorithm.

- From an algorithmic point of view, the usefulness of a relaxation is decided not only by its optimum value, but also by the consideration how fast this value can be determined or sufficiently approximated. We analyze and improve some algorithmic approaches to the relaxations.

- Especially in the context of exact algorithms, a major problem arises when none of the (practically tractable) relaxations is strong enough to solve the instance without resorting to branching (or to enable further reductions). We present two theoretically interesting and practically applicable techniques for improving the relaxations, namely graph transformation and local cuts. These techniques have proven to be quite powerful, especially for the solution of large and complex instances. In particular, the method of graph transformation, which was applied by us for the first time to a network optimization problem, seems quite promising.

Reductions to Simplify Problem Instances (Chapter 3)

- 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 instance 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.

- In the solution process, particularly as the result of our other reduction techniques, we frequently encounter graphs of (locally) low connectivity; but the standard methods based on partitioning are not helpful for exploiting this situation. We present the new approach of using partitioning to design reduction methods. As we will show, the resulting methods have been quite effective in the context of Steiner problem, and the approach can also be useful for other problems.

- 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 (see for example the comparison in [CT01]).

Heuristics and Upper Bounds (Chapter 4)

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

Exact Algorithms (Chapter 5)

- We present an algorithm that exploits small width in (sub-) graphs, and show how it can be used profitably in combination with our other techniques in a more general context.
- We describe the interaction between different components and how we take advantage of it to design a very powerful reduction process. Then we outline how this reduction process is used in a branch-and-bound framework.

In each chapter, we present summarized experimental results, including comparisons of our results to the best other results we are aware of. Detailed results for the exact algorithm are given in the appendix. Considering these results, we can sum up:

- We could solve all instances in SteinLib that have been solved before; in most cases in running times that are smaller than those of all other authors by orders of magnitude (see Table 5.1 on page 123 and also the comparison in [CT01]).
- There were 74 instances in SteinLib that have not been solved by any other research group. We have been able to solve 33 of them. All still unsolved instances have been constructed to be difficult for known techniques (see Table A.2 on page 130).
- For geometric Steiner problems, our algorithm for general networks outperforms, for large instances by orders of magnitude, the specially tailored MSTH approach [WWZ00], which has received much attention (see Tables 5.2 and 5.3 on page 124).

Similar to other elaborate optimization packages, our program package for the Steiner 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 solutions 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 pseudocode. 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 fine-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 in case we use standard techniques, we give only pointers to them.

Some background information is given in the rest of this chapter, much more can be found in a book by Hwang, Richards and Winter [HRW92]; we have tried to keep the notation compatible with that book. The basic definitions are given in the next section, the rest is distributed over the chapters. The index should help the reader to find the searched terms.

1.2 Basic Definitions and Notations

We use the usual definitions concerning graphs, see for example [CLR90]. 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 $\{v_i, v_j\}$. For any directed graph $\vec{G} = (V, A)$, we use $[v_i, v_j]$ to denote the directed edge, or arc, from v_i to v_j, and define $a := |A|$. A network is here a weighted graph (V, E, c) with an edge cost 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 of (v_i, v_j) interchangeably to denote $c((v_i, v_j))$ (also denoted by $c(v_i, v_j)$, $c_{(v_i,v_j)}$ or c_{ij}). For any network H, $c(H)$ denotes the sum of the edge weights of H.

The **Steiner problem in networks** 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 such that in $T_G(R)$, there is a path between every pair of terminals, and the value $c(T_G(R))$ is minimized.

We define $r := |R|$. For ease of notation we sometimes 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 $\vec{G}$ and R, a root $z_r \in V$ is given and it is required that the solution contains a path from z_r 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 an arbitrary terminal z_r as the root. We define $R^{z_r} := R \setminus \{z_r\}$.

With $d_G(v_i, v_j)$ (or $d(v_i, v_j)$ or d_{ij}) we denote the distance (length of a shortest path) between v_i and v_j in G. For a given network $G = (V, E, c)$ and $W \subseteq V$, the corresponding **distance network** is defined as $D_G(W) := (W, W \times W, d)$. We define $D := D_G(V)$. It is easy to see that $c(T_G(R)) = c(T_D(R))$, so every instance of the Steiner problem can be transformed into an equivalent instance in a metric network in time, for example, $O(n^3)$. Furthermore, by computing a minimum spanning tree for $D_G(R)$ and replacing its edges with the corresponding paths in G, we get a feasible solution of the Steiner problem for the original instance; this is the core of a well-known heuristic with a worst-case performance ratio of $(2 - 2/r)$ for the Steiner problem which we call **DNH** (for Distance Network Heuristic; see for example [HRW92]). Mehlhorn [Meh88] showed how to compute such a tree efficiently by using a concept similar to that of Voronoi regions in algorithmic geometry; we will use this concept also in other contexts. For each terminal z_i, one can define a neighborhood $N(z_i)$, called the **Voronoi region** of z_i, as the set of vertices that are not closer to any other terminal. More precisely, a partition of V is defined:

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

If $v_k \in N(z_i)$, we call z_i the **base** of v_k (written $base(v_k)$). 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 $v_l \in N(z_j)$. Consider a graph G' with the vertex set R and an edge between each two neighbor terminals z_i, z_j with the cost $c'(z_i, z_j) := \min\{d(z_i, v_k) + c(v_k, v_l) + d(v_l, z_j) \mid v_k \in N(z_i), v_l \in N(z_j)\}$. A minimum spanning tree T' for G' will also be a minimum spanning tree $T'_D(R)$ for $D_G(R)$. The neighborhoods $N(z)$ for all $z \in R$, the graph G' and the tree T' can be constructed in total time $O(m + n \log n)$ [Meh88].

1.3 Background

The Steiner problem is one of the best-known and most-studied problems in (combinatorial) optimization. For the Steiner problem in networks, as we study it in this work, there are more than a hundred publications; this number grows to several hundreds if one also considers variants of this problem. There are several books devoted to the Steiner problem, for example [Voß90, HRW92, Cie98, DSR00, CD01, PS02].

1.3.1 History

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

Given three points in the plane, find a point the sum of whose distances from the given points is minimal.

Jakob Steiner (1796-1863) considered a generalization of this problem for r points (the general 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 proposed for the first time by Gauß (see [CD01]). The first (terminating) algorithm for the Euclidean Steiner problem was given by Melzak [Mel61]. More information on the Euclidean Steiner problem and its history can be found in [HRW92]. Its relation to the network variant will be discussed below.

The Steiner problem in networks was explicitly formulated for the first time by Hakimi [Hak71] and Levin [Lev71]. A good (although not fully up-to-date) overview is given in Hwang et al. [HRW92].

1.3.2 Geometric Steiner Problems

In geometric Steiner problems, a set of points (in the plane) is to be connected at minimum cost under some geometric norm. Two famous versions are:

Euclidean Steiner Problem: Given a finite set R of points in the plane, find a point set S together with a spanning tree T for $R \cup S$ such that T has minimum length under L_2 norm (Euclidean distance).

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

Furthermore, every Euclidean instance I can be approximated for any $\epsilon > 0$ by a network instance I_ϵ with less than $const\ \frac{r^2}{\epsilon^2}$ 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 lay a grid with proper granularity on the convex hull of the given points and allow only the grid points as possible Steiner nodes (for details see [HRW92]). But a direct application of this method is not practical: For a guarantee of 1% one ends up with a complete network with up to $const\ 10^4\ r^2$ vertices.

Rectilinear Steiner Problem: Given a finite set R of points in the plane, find a point set S together with a spanning tree T for $R \cup S$ such that T has minimum length under L_1 norm (rectilinear or Manhattan distance).

The rectilinear version is actually a special case of the network version, as shown by Hanan [Han66]: Draw horizontal and vertical lines through the given points. Define a network $G = (V, E, c)$, with V being the set of all line crossings and E corresponding to the resulting line segments. Hanan showed that an optimal solution in G is an optimal solution for the original rectilinear instance. But again, this approach is not very practical, since the special geometric information is mainly lost. Furthermore, for say 10,000 given points we could end up with a network with up to 100 millions of vertices, which is completely beyond the reach of current (exact) algorithms. But as we describe below, rectilinear instances with 10,000 points can now be regularly solved in relatively small times.

Both problems are very well studied and there is a huge amount of literature on them (see [HRW92] for an overview and [WWZ00] for some recent successful methods).

In this work, we do not work directly with geometric Steiner problems. However, as we will describe in Section 2.8, for Euclidean and rectilinear Steiner problems (and some variations of them) the currently most successful approach uses a geometric preprocessing (first) phase to convert the given problem into one that is independent of the underlying geometry. We will study this converted problem, both theoretically and experimentally. It turns out that our program is currently the fastest one for the second phase, solving instances with about 10,000 terminals usually in a matter of minutes to hours (see Section 5.4.1).

1.3.3 Applications

Various versions of the Steiner problem belong to the most-applied problems in combinatorial optimization. A voluminous book [CD01] is devoted to the applications of Steiner trees in industry. As examples, we briefly describe some applications of the Steiner problem in networks from various fields:

Routing in (Computer) Networks

In the multicast routing problem we are given a network $G = (V, E, c)$ with a source $s \in V$ and a set of destinations $S \subseteq V \setminus \{s\}$. The cost mapping c can be a complicated function with arguments like delay of a channel, fees, and so on. Two well-known special cases are $|S| = 1$ (single destination routing) and $|S| = |V| - 1$ (broadcasting). A routing tree T is a subnetwork of G, rooted at s and containing all vertices of S. The cost-optimal routing asks for a routing tree with minimum total cost.

An approach for cost-optimal routing calculates a Steiner (minimal) tree T for G with $R = S \cup \{s\}$. 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 [BKJ83].

For more information on this application, see [NRK01].

VLSI Layout

Several variants of the Steiner problem 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.

Some (classical) problems in VLSI layout used to be formulated as rectilinear Steiner problems. But for many applications, a network formulation is indicated, e.g., because there are obstacles on the chip (see for example [KM98]). Besides, often additional constraints are to be considered, which can be better modeled in a network framework.

For an overview of applications of Steiner problems in VLSI layout see [KPS90] or [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 s letters of a finite alphabet $\mathcal{A}$ (i.e., a vector from $\mathcal{A}^s$). Now we have a (complete) network $G = (V, E, c)$ with $V = \mathcal{A}^s$, 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 Steiner (minimal) 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 goals. An overview is given in [HRW92].

Of course, one cannot work directly with the (huge) network described above. One of the recognized methods is to work in a so-called quasi-median network [BFR99], which is guaranteed to contain all relevant Steiner minimal trees. A software package that combines methods for computing such networks and variants of our program for computing Steiner minimal trees is already in use and frequently cited [For03].

1.4 Complexity Results

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

For most important metrics, the Steiner problem remains $\mathcal{NP}$-hard, for example:

- distance in networks (direct consequence of $c(T_G(R)) = c(T_D(R))$, see Section 1.2),
- Euclidean distance (Garey, Graham and Johnson [GGJ77]),
- Manhattan distance (Garey and Johnson [GJ77]),
- Hamming distance (Foulds and Graham [FG82]).

1.4.1 Special Cases

For most important classes of networks, the Steiner problem remains $\mathcal{NP}$-hard, for example:

bipartite networks, even if all edges 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 from $\{1, 2\}$ (Bern and Plassman [BP89]).

An overview is given in [Joh85].

Polynomially Solvable Cases

Some important special cases of the Steiner problem can be solved very efficiently. The most important ones are:

$r = 2$: The solution is a shortest path between the two given terminals. It can be computed, for example, in time $O(m + n \log n)$. See [Tho97] for a more recent result.

$r = n$: The solution is a minimum spanning tree (MST) for G. It can be computed, for example, in time $O(m + n \log n)$. See [KKT95, PR00] for more recent results.

These facts can be slightly generalized:

- A Steiner minimal tree can be computed in time $O(n3^r + n^2 2^r + n^3)$ with a dynamic programming algorithm which uses the case $r = 2$ as the base case (Dreyfus and Wagner [DW71]). For (e.g.) constant r, we get a polynomial-time algorithm. However, this algorithm is practical only for very small r (as a guideline: at most 10); and even then, it is regularly outperformed by other exact algorithms empirically.
- A Steiner minimal tree can be computed in time $O(\min\{2^{n-r}, (n-r)^{r-2}\}r^2 + n^3)$ by enumeration of minimum spanning trees for supersets of R (Hakimi [Hak71] and Lawler [Law76]). For (e.g.) constant $n - r$ or r , we get a polynomial-time algorithm. If n is not large, this approach is practical for small $n - r$ (as a guideline: at most 15).

Only very special classes of networks (like trees, series-parallel networks, Halin networks, ...) are known to admit polynomial-time algorithms for the Steiner problem (see [HRW92, Chapter 5]). More interesting (and general) are networks with certain restrictions on (tree, branch, or path) width. In Section 5.2, we discuss this class of networks and show how to exploit restricted (path) width in the networks to derive a theoretically efficient and practically useful algorithm.

1.4.2 Approximability

The Steiner problem in networks is $\mathcal{APX}$-complete, even in complete networks with edge weights from $\{1, 2\}$ (Bern and Plassman [BP89]). So, there is some fixed $\epsilon > 0$ such that computing $(1 + \epsilon)$-approximations for this problem is $\mathcal{NP}$-hard (see Arora [Aro94]). The currently best lower bound for this ϵ is $1/95$ (Chlebík and Chlebíková [CC02]).

Most classical Steiner tree heuristics guarantee an approximation ratio of (almost) two. In recent years, better ratios were obtained step by step by a series of algorithms. The first one had a ratio of $11/6$ (Zelikovsky [Zel93]) and an efficient implementation of it has running time $O(r(m + n \log n + rn))$. But the empirical results of this algorithm are only mediocre and do not justify the relatively long running times [DV97]. The currently best approximation ratio is $1 + \ln(3)/2 \approx 1.55$ (Robins and Zelikovsky [RZ00]), but the polynomial describing the corresponding running time contains a k in the exponent that has to approach ∞ to reach this ratio. Thus, this result is hardly of practical relevance. For a recent survey on approximation results, see [GHNP01].

We mention here that for the Euclidean Steiner problem, Arora [Aro96] (and, independently, Mitchell [Mit96]) developed polynomial-time approximation schemes. The methods used are quite general and the results extend to a huge number of other geometric problems (in constant dimensions) under some L_p norm for $p \geq 1$ [Aro03]. (In fact the scheme was designed originally for the traveling salesman problem.) However, these algorithms seem to be not competitive for the traveling salesman or Steiner problem [Aro03], at least if the worst-case guarantees are to be maintained.

Chapter 2

Relaxations and Lower Bounds

2.1 Introduction

To attack a (hard) combinatorial optimization problem, a starting point could be a reformulation of the problem as an integer program. Dropping (relaxing) some of the constraints in such a program can make it more tractable. The solution of such a relaxation provides us with a lower bound (in case of minimization) for the original problem, which is a major component of many exact algorithms (see Chapter 5). But as we will show in Chapter 3 on reduction techniques and Chapter 4 on upper bounds, the information we get from handling such a relaxation can also be used beyond the computation of lower bounds.

Some of the most successful approaches to hard combinatorial optimization problems are based on linear programming, consider for example the long history of research for the traveling salesman problem (TSP) focusing on linear programming [ABCC03]. Here, the problem is first formulated as an integer linear program. Dropping the integrality constraints leads to the so-called **LP relaxation** (or **linear relaxation**) of the problem. In this way one can profit from the numerous sophisticated techniques for solving or approximating linear programs.

For $\mathcal{NP}$-hard optimization problems, unless $\mathcal{P} = \mathcal{NP}$, any linear relaxation of polynomial size (and any polynomial-time solvable relaxation) is bound to have a deviation from the solution of the original problem in some cases. The quality of the used relaxation can have a decisive impact on the performance of the algorithms based on it. For the Steiner problem in networks, many (mixed) integer programming formulations have been suggested as the starting point for relaxations, but not much was known about the relative quality of the corresponding relaxations. In Sections 2.2 to 2.5, we compare the linear relaxations of all classical 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 that have often been treated as unrelated or incomparable. In Section 2.6, we introduce a collection of new relaxations that are stronger than all previous linear relaxations. We will also discuss some variants for application in practical algorithms. Finally, in Section 2.7 we build a hierarchy of all discussed relaxations.

For geometric Steiner problems, the state-of-the-art algorithms follow a two-phase approach: First, by exploiting geometric properties a proper collection of the so-called full Steiner trees (FSTs) is generated. Then one chooses a subset of the generated FSTs whose concatenation yields a Steiner minimal tree. The bottleneck of this approach has usually been the second phase, where the hitherto most successful approach treats the generated FSTs as edges of a hypergraph and uses an LP relaxation of the minimum spanning tree in hypergraph (MSTH) problem. In Section 2.8, we study this original and some new relaxations of the MSTH problem and show that they are all equivalent. We also clarify their relations to the relaxations of the Steiner problem. An experimental comparison of these relaxations is presented in Section 2.13. Later (in Section 5.4.1) we will see that our program outperforms the currently best MSTH-based algorithm.

From an algorithmic point of view, the usefulness of a relaxation is decided not only by its optimum value, but also by the consideration how fast this value can be determined or sufficiently approximated. In Sections 2.9 to 2.11, we study some algorithmic approaches to the relaxations. In Section 2.9, we study several old and new algorithms for computing lower (and upper) bounds using dual-ascent and primal-dual strategies. We present several new results and some improvements, show that none of the known algorithms can both generate tight lower bounds empirically and guarantee their quality theoretically, and we present a new algorithm that combines both features. In Section 2.10, we outline some approaches based on Lagrangian relaxation. In Section 2.11, we describe how a row and column generation technique can be used to optimize an LP relaxation and show how to adapt this approach to one of our stronger relaxations.

Especially in the context of exact algorithms, a major problem arises when none of the (practically tractable) relaxations is strong enough to solve the instance without resorting to branching (or to enable further reductions). In Section 2.12, we present two theoretically interesting and practically applicable techniques for improving the relaxations, namely graph transformation and local cuts, in the context of the Steiner problem. These techniques have proven to be quite powerful, particularly for the solution of large and complex instances.

In Section 2.13, we report some experimental results for computing lower bounds based on the relaxations. We also demonstrate the impact of our techniques for improving relaxations in the solution of the largest benchmark instances ever solved.

2.1.1 Additional Definitions for Relaxations

We often use directed formulations, because the corresponding relaxations can be stronger. Remember (from Section 1.2) that every instance $(G = (V, E, c), R)$ of the undirected Steiner problem can be transformed into an instance of the directed Steiner problem in the corresponding (bi-) directed network $\vec{G} = (V, A, c)$ ($A := \{[v_i, v_j], [v_j, v_i] \mid (v_i, v_j) \in E\}$, c defined accordingly) by fixing an arbitrary terminal (say z_1) as the root.

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) linear programming formulations we use (binary) variables $x_{[v_i,v_j]}$ or x_{ij} for each arc (respectively $x_{(v_i,v_j)}$ or 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 $\sum_{b \in B} x_b$, 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 $\sum_{[v_i,v_j] \in A, v_i \notin W, v_j \in W} x_{ij}$.

We use the notation P_q for integer linear programs, where the abbreviation q describes the program further (see Table 2.1 for an explanation of the abbreviations). The corresponding linear relaxation is denoted by LP_q; the dual of such a relaxation is denoted by DLP_q and a Lagrangian relaxation by LaP_q. These notations denote a program corresponding to an arbitrary, but fixed instance of the Steiner problem. The value of an optimal solution of an integer programming formulation, denoted by $v(P_q)$, is of course the value of an optimal solution of the corresponding Steiner problem. Thus, in this context we are interested in the optimal value $v(LP_q)$ of the corresponding linear relaxation, which can differ from $v(P_q)$ (**integrality gap**).

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**.

2.2 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.

C	cut	Section 2.2.1
F	flow	Section 2.2.2
F^R	common-flow	Section 2.6.3
F^{k_1,k_2}	restricted version of F^R (multiple roots)	Section 2.6.4
F^2	restricted version of F^R (only one root)	Section 2.6.4
$2t$	two-terminal	Section 2.2.2
T	tree	Section 2.3
mT	multiple trees	Section 2.5
T_0	degree constrained spanning tree	Section 2.3.1
$\vec{T}$	directed version of T	
Uq	undirected version of q	
$q + FB$	q with added flow-balance constraints	Section 2.5.2
$q-$	weaker version of q	
$q++$	aggregated version of q	
q'	modified version of q	

Table 2.1: Abbreviations for Linear programs and their meaning.

2.2.1 Cut Formulations

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

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

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 by Aneja [Ane80]:

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

Lemma 1 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].

2.2.2 Flow Formulations

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

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

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 2 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 (see Lemma 3).

$$
\begin{aligned}
\boxed{P_{F++}} \qquad c \cdot x &\rightarrow \min, \\
Y(\delta^-(v_i)) &= Y(\delta^+(v_i)) + \begin{cases} 1 & (v_i \in R^{z_1}), \\ 0 & (v_i \in V \setminus R), \end{cases} && (4.1) \\
(r-1)x &\geq Y, && (4.2) \\
Y &\geq 0, && (4.3) \\
x &\in \{0,1\}^{|A|}. && (4.4)
\end{aligned}
$$

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

Lemma 3 LP_F is strictly stronger than LP_{F++}. The worst-case ratio $\frac{v(LP_F)}{v(LP_{F++})}$ is $r-1$ [Mac87, Dui93].

The formulation P_F is based on the flow formulation of the shortest path problem (the special case of the Steiner problem with $|R^{z_1}| = 1$). Liu [Liu90] stated the two-terminal formulation P_{2t}, which is based on the special case with $|R^{z_1}| = 2$, namely the two-terminal 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.

$$\begin{aligned}
\boxed{P_{2t}} \qquad c \cdot x &\rightarrow \min, \\
\check{y}^{kl}(\delta^-(v_i)) - \check{y}^{kl}(\delta^+(v_i)) &\geq \begin{cases} -1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_1), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1\}), \end{cases} && (5.1) \\
(\check{y}^{kl} + \grave{y}^{kl})(\delta^-(v_i)) - (\check{y}^{kl} + \grave{y}^{kl})(\delta^+(v_i)) &= \begin{cases} 1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_k), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1, z_k\}), \end{cases} && (5.2) \\
(\check{y}^{kl} + \acute{y}^{kl})(\delta^-(v_i)) - (\check{y}^{kl} + \acute{y}^{kl})(\delta^+(v_i)) &= \begin{cases} 1 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i = z_l), \\ 0 & (\{z_k, z_l\} \subseteq R^{z_1} \,;\, v_i \in V \setminus \{z_1, z_l\}), \end{cases} && (5.3) \\
\check{y}^{kl} + \acute{y}^{kl} + \grave{y}^{kl} &\leq x \quad (\{z_k, z_l\} \subseteq R^{z_1}), && (5.4) \\
\check{y}^{kl},\ \acute{y}^{kl},\ \grave{y}^{kl} &\geq 0 \qquad (\{z_k, z_l\} \subseteq R^{z_1}), && (5.5) \\
x &\in \{0,1\}^{|A|}. && (5.6)
\end{aligned}$$

Note that the flow described by $\check{y}$ can have an excess at some vertices (because of the inequality (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 4 LP_{2t} is strictly stronger than LP_F [Liu90].

2.3 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.

2.3.1 Degree-Constrained Tree Formulations

The Steiner 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) [BP87, Bea89]. 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. To see this, observe that T_0 can be transformed into a Steiner (minimal) tree of the same cost in G by removing edges (v_0, v_i) that are part of T_0, and vice versa. Beasley [Bea89] stated this reformulation as an integer program:

$$\begin{aligned}
\boxed{P_{T_0}} \qquad c \cdot X &\rightarrow \min, \\
\{(v_i, v_j) \mid X_{ij} = 1\} &: \text{ builds a spanning tree for } G_0, && (7.1) \\
X_{0k} + X_{ki} &\leq 1 \quad (v_k \in V \backslash R \,;\, (v_k, v_i) \in \delta(v_k)), && (7.2) \\
X &\in \{0,1\}^{|E_0|}. && (7.3)
\end{aligned}$$

The requirement (7.1) can be stated by linear constraints. In the following, we assume that (7.1) is replaced by the following constraints.

$$\begin{aligned}
X(E_0) &= n, && (7.4) \\
X(E_0(W)) &\leq |W| - 1 \quad (\emptyset \neq W \subset V_0). && (7.5)
\end{aligned}$$

The constraints (7.4) and (7.5), together with the non-negativity 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 (7.1) can lead to a stronger linear relaxation.

Again, a similar directed formulation using a network $\vec{G}_0$ can be stated, 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 $\vec{G}$.

$$
\begin{aligned}
\boxed{P_{\vec{T}_0}} \qquad c \cdot x &\rightarrow \min, \\
x(\delta^-(v_i)) &= 1 \quad (v_i \in V), && (8.1)\\
x(A_0(W)) &\leq |W| - 1 \quad (\emptyset \neq W \subseteq V_0), && (8.2)\\
x_{0i} + x_{ij} + x_{ji} &\leq 1 \quad (v_i \in V \setminus R\,;\, [v_i, v_j] \in \delta^+(v_i)\,), && (8.3)\\
x &\in \{0,1\}^{|A_0|}. && (8.4)
\end{aligned}
$$

The constraints (8.1) and (8.2), together with the non-negativity 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 (8.3) (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.

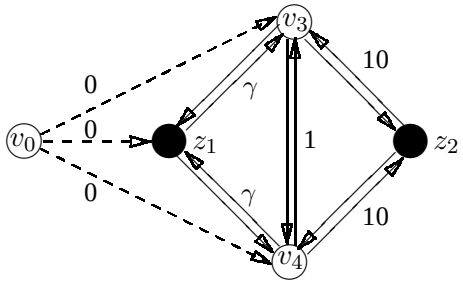


Figure 2.1: Example with $v(LP_{\vec{T}_0-}) \ll v(LP_{\vec{T}_0}) = v(LP_{T_0}) < v(P_{T_0})$.

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.

2.3.2 Rooted Tree Formulation

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

$$
\begin{array}{rcll}
\boxed{P_{\vec{T}}} \qquad c \cdot x & \rightarrow & \min, & \\
x(\delta^-(v_i)) & = & 1 \quad (v_i \in R^{z_1}), & (9.1)\\
x(\delta^-(v_i) \setminus \{[v_j, v_i]\}) & \geq & x_{ij} \quad (v_i \in V \setminus R\,;\, [v_i, v_j] \in \delta^+(v_i)), & (9.2)\\
x(A(W)) & \leq & |W| - 1 \quad (\emptyset \neq W \subseteq V), & (9.3)\\
x & \in & \{0,1\}^{|A|}. & (9.4)
\end{array}
$$

To get rid of the exponential number of constraints for avoiding cycles, many authors have considered replacing (9.3) 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 $v_i \in V$:

$$t_i - t_j + nx_{ij} \leq n - 1 \qquad ([v_i, v_j] \in A). \tag{9.5}$$

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 29).

Lemma 5 $v(LP_{\vec{T}-}) \leq v(LP_{\vec{T}})$.

Proof: Let $\hat{x}$ denote an (optimal) solution for $LP_{\vec{T}}$. Obviously $\hat{x}$ satisfies the constraints (9.1) and (9.2). We now show that it is possible to construct $\hat{t}$ such that $(\hat{x}, \hat{t})$ satisfies (9.5), 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 (9.5)).
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 non-empty, 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 (9.3), $\hat{x}(C) \leq l - 1$. The consequence, $\frac{l-1}{l} > \frac{n-1}{n}$, is a contradiction. □

2.3.3 Equivalence of the 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 6 $v(LP_{\vec{T}_0}) = v(LP_{T_0})$.

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 Γ with $\Gamma_{ij} \in [0,1]$ arbitrarily (for all $(v_i, v_j) \in E$) and set x to $x_{ij} := \Gamma_{ij} X_{ij}$, $x_{ji} := (1 - \Gamma_{ij}) X_{ij}$ (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 (8.2) and (8.3) and yields the same value as $v(LP_{T_0})$.
The only question is, whether there is a Γ such that x satisfies the constraints (8.1), 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:

$$\begin{aligned}
|W \setminus \{v_0\}| + X(E_0 \setminus E_U) &\geq |W \setminus \{v_0\}| + X(E_0 \setminus E_W) && \text{(since } E_U \subseteq E_W\text{)} \\
&\geq |W| - 1 + X(E_0) - X(E_W) \\
&= |W| - 1 + n - X(E_W) && \text{(because of 7.4)} \\
&\geq n. && \text{(because of 7.5)}
\end{aligned}$$

It follows that the minimal cut has capacity n. □

Lemma 7 $v(LP_{\vec{T}}) = v(LP_{\vec{T}_0})$.

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 (8.1) and in $\vec{G}_0$ only arcs in A are incident with terminals in R^{z_1}, $\tilde{x}$ satisfies the constraints (9.1).
Furthermore, $\tilde{x}$ satisfies the constraints (9.2), because for every arc $[v_i, v_j] \in A$ with $v_i \in V \setminus R$ it holds that:

$$\begin{aligned}
\tilde{x}(\delta^-(v_i) \setminus \{[v_j, v_i]\}) &= \tilde{x}(\delta^-(v_i)) - \tilde{x}_{ji} && (\delta \text{ in } \vec{G}) \\
&= \hat{x}(\delta^-(v_i)) - \hat{x}_{0i} - \hat{x}_{ji} && (\delta \text{ in } \vec{G}_0) \\
&= 1 - \hat{x}_{0i} - \hat{x}_{ji} && \text{(because of (8.1))} \\
&\geq \hat{x}_{ij} && \text{(because of (8.3))} \\
&= \tilde{x}_{ij}.
\end{aligned}$$

Finally $\tilde{x}$ satisfies (9.3), because $\hat{x}$ satisfies (8.2).
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 (9.2) for some arc $[v_i, v_l]$ with $\tilde{x}(\delta^-(v_i) \setminus \{[v_l, v_i]\}) = \tilde{x}_{il}$, and it would follow that $1 < \tilde{x}(\delta^-(v_i)) = \tilde{x}_{li} + \tilde{x}_{il}$, but this is ruled out by (9.3) (for $W = \{v_i, v_l\}$). So $\hat{x}$ satisfies (8.1) in a trivial way.
The constraints (8.2) are satisfied by $\hat{x}$ for every $W \subseteq V$, because $\tilde{x}$ satisfies (9.3). For $W \subseteq V_0$ with $v_0 \in W$ it holds that:

$$\begin{aligned} \hat{x}(A_0(W)) &\leq \sum_{v_i \in W \setminus \{v_0\}} \hat{x}(\delta^-(v_i)) \qquad (\text{in } \vec{G}_0) \\ &= \sum_{v_i \in W \setminus \{v_0\}} 1 \qquad (\text{because of } (8.1)\ ) \\ &= |W| - 1. \end{aligned}$$

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

$$\begin{aligned} \hat{x}_{0i} + \hat{x}_{ij} + \hat{x}_{ji} &= 1 - \tilde{x}(\delta^-(v_i)) + \tilde{x}_{ij} + \tilde{x}_{ji} \qquad (\text{in } \vec{G}) \\ &= 1 - \tilde{x}(\delta^-(v_i) \setminus \{[v_j, v_i]\}) + \tilde{x}_{ij} \\ &\leq 1. \qquad (\text{because of } (9.2)\ ) \end{aligned}$$

Thus, $\hat{x}$ also satisfies the constraints (8.3). □

2.4 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 8 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 $[v_l, v_k] \in \delta^-(v_k) \setminus \delta^-(W)$ 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 †:

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

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 9 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 8 (for v_k and W), one gets a contradiction. □

Lemma 10 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 8 (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 11 $v(LP_{\vec{T}}) \leq v(LP_C)$.

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 $v_i \in R^{z_1}$, by Lemma 9, $\hat{x}$ satisfies (9.1).
Because of Lemma 10, $\hat{x}$ satisfies (9.2).
Let $W \subseteq V$ be a non-empty set. If $z_1 \in W$:

$$\begin{aligned}
\hat{x}(A(W)) &\leq \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) \\
&= \sum_{v_i \in W \backslash \{z_1\}} \hat{x}(\delta^-(v_i)) \qquad \text{(optimality of } \hat{x}) \\
&\leq \sum_{v_i \in W \backslash \{z_1\}} 1 \qquad \text{(Lemma 9)} \\
&= |W| - 1.
\end{aligned}$$

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

$$\begin{aligned}
\hat{x}(A(W)) &= \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - \hat{x}(\delta^-(W)) \\
&\leq \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - 1 \qquad (\Delta \geq 1) \\
&\leq \sum_{v_i \in W} 1 \quad - 1 \qquad \text{(Lemma 9)} \\
&= |W| - 1.
\end{aligned}$$

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

II) $\Delta < 1$:

$$\begin{aligned} \hat{x}(A(W)) &= \sum_{v_i \in W} \hat{x}(\delta^-(v_i)) - \hat{x}(\delta^-(W)) \\ &\leq \sum_{v_i \in W} \hat{x}(\delta^-(W)) - \hat{x}(\delta^-(W)) \qquad \text{(Lemma 8)} \\ &= (|W| - 1)\hat{x}(\delta^-(W)) \\ &< |W| - 1. \qquad (\Delta < 1) \end{aligned}$$

It follows that $\hat{x}$ satisfies (9.3) too. □

Corollary 11.1 The proof shows that adding constraints of $LP_{\vec{T}}$ to LP_C cannot improve $v(LP_C)$.

Corollary 11.2 Because the proofs of the equivalence of the tree relaxations require the optimality only in one step of Lemma 7 to show that $\tilde{x}(\delta^-(v_i)) \leq 1$, which is forced by Lemma 9 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.

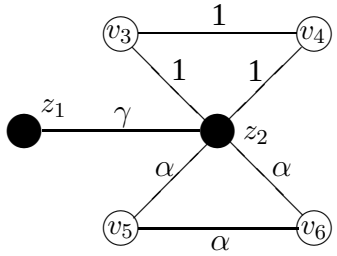


Figure 2.2: Example for $v(LP_{\vec{T}_-}) \ll v(LP_{\vec{T}}) \ll v(LP_F) = v(P_F)$.

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 $\frac{v(LP_{\vec{T}_-})}{v(LP_{\vec{T}})}$.

2.5 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.

2.5.1 Multiple Trees Formulation

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

$$
\begin{array}{rcll}
\boxed{P_{m\vec{T}}} \quad c \cdot X & \rightarrow & \min, & \\
X(\delta(v_i)) & \geq & 1 \qquad (v_i \in R), & (10.1)\\
X(\delta(v_i)) & \geq & 2s_i \qquad (v_i \in V \setminus R), & (10.2)\\
s_i & \geq & X_{ij} \qquad (v_i \in V \setminus R\,;\ (v_i, v_j) \in \delta(v_i)\ ), & (10.3)\\
x^k_{ij} + x^k_{ji} & = & X_{ij} \qquad (v_k \in R\,;\ (v_i, v_j) \in E), & (10.4)\\
x^k(\delta^-(v_i)) & = & \begin{cases} 1 & (v_k \in R\,;\ v_i \in R \setminus \{v_k\}), \\ 0 & (v_k \in R\,;\ v_i = v_k), \end{cases} & (10.5)\\
x^k(\delta^-(v_i)) & = & s_i \qquad (v_k \in R\,;\ v_i \in V \setminus R), & (10.6)\\
\{[v_i, v_j] \mid x^k_{ij} = 1\} & : & \text{contains no cycles} \quad (v_k \in R), & (10.7)\\
X & \in & \{0,1\}^{|E|}, & (10.8)\\
x^k & \in & \{0,1\}^{|A|} \qquad (v_k \in R), & (10.9)\\
s_i & \in & \{0,1\} \qquad (v_i \in V \setminus R). & (10.10)
\end{array}
$$

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.

2.5.2 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:

$$x(\delta^-(v_i)) \leq x(\delta^+(v_i)) \qquad (v_i \in V \setminus R). \tag{11.1}$$

We denote the linear program that consists of LP_F and (11.1) 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.

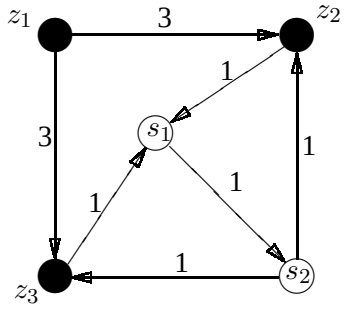


Figure 2.3: Example with $v(LP_F) < v(LP_{F+FB}) = v(P_{F+FB})$.

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:

$$\boxed{P_{F'+FB}} \qquad \begin{aligned} c \cdot X &\rightarrow \min, \\ x_{ij} + x_{ji} &= X_{ij} \qquad ((v_i, v_j) \in E), \qquad (12.1) \\ (x, y) &: \text{ is feasible for } P_{F+FB}. \qquad (12.2) \end{aligned}$$

Lemma 12 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 (12.1) 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 (11.1).
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}.

2.5.3 Relationship between the Two Models

We will now show that the linear relaxation $LP_{m\vec{T}}$ (where (10.7) is replaced by linear constraints of the form (9.3)) is equivalent to LP_{F+FB}.

Lemma 13 $v(LP_{m\vec{T}}) = v(LP_{F'+FB})$.

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 (10.4) 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:

$$\begin{aligned} y^t(\delta^-(v_i)) - y^t(\delta^+(v_i)) &= (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} > \hat{x}^t_{ji}\}) - \\ &\quad (\hat{x}^1 - \hat{x}^t)(\{[v_i, v_j] \in \delta^+(v_i) | \hat{x}^1_{ij} > \hat{x}^t_{ij}\}) \\ &= (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} > \hat{x}^t_{ji}\}) + \\ &\quad (\hat{x}^1 - \hat{x}^t)(\{[v_j, v_i] \in \delta^-(v_i) | \hat{x}^1_{ji} < \hat{x}^t_{ji}\}) \quad \text{(because of (10.4))} \\ &= (\hat{x}^1 - \hat{x}^t)(\delta^-(v_i)) \quad = \quad 0 \quad \text{(because of (10.5) or (10.6)).} \end{aligned}$$

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 (10.4) and (10.6) into (10.2) gives the flow-balance constraints (11.1). 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 12 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 11 it follows that $(X, \hat{x}, \hat{s})$ is feasible for $LP_{m\vec{T}}$. □

Corollary 13.1 The constraints (10.1), (10.3), and (10.7) are useless with respect to the value of the linear relaxation $LP_{m\vec{T}}$.

Corollary 13.2 The linear program $LP_{m\vec{T}-}$ (with the same objective function as $LP_{m\vec{T}}$) that contains (beside non-negativity constraints) only the equations (10.4), (10.5), and (10.6) is equivalent to LP_F.

2.6 A Collection of New Formulations

In [PV01a] we introduced the common flow relaxation LP_{F^2}. 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.

2.6.1 Integrality Gap of the Flow/Cut Relaxations

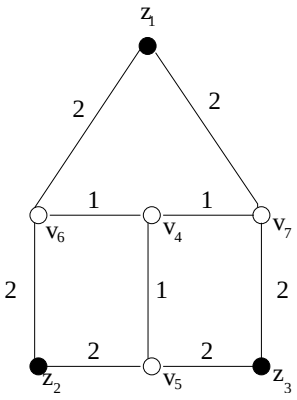


Figure 2.4: Example with $v(LP_C) < v(P_C)$.

A network with a deviation between the integral and the linear solution for the flow formulation 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 $\frac{16}{15}$. Goemans [Goe98] extended the example in Figure 2.4 to networks G_k whose integrality gap comes arbitrarily 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 deviation known for LP_F. [2]

[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 $i > 0$ without deteriorating the asymptotic result.

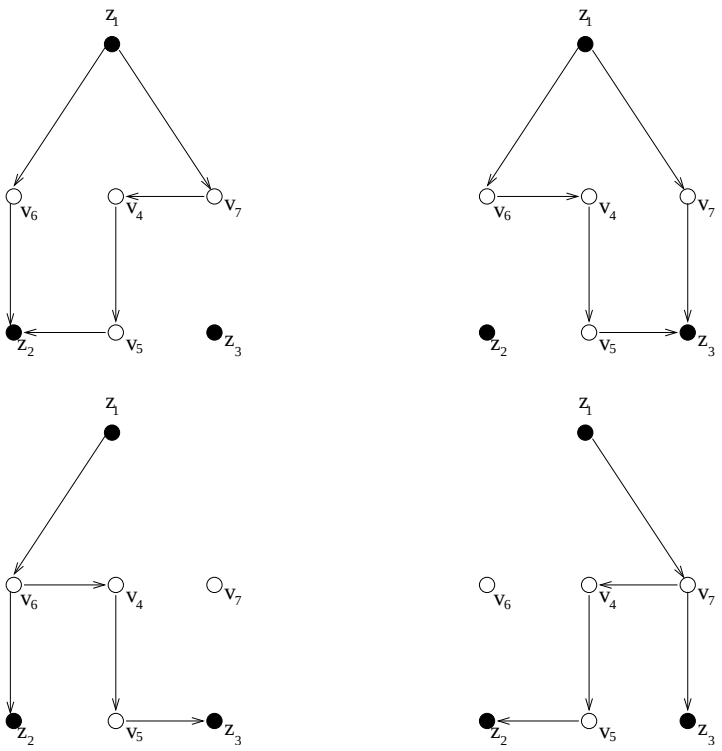


Figure 2.5: y-variables and decomposition into trees.

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

If the x-values 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 the vertex v_4 flows of 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 a vertex $v_i \in V$ is defined by the condition $\sum_{[v_j,v_i]\in\delta^-(v_i)} \max\{y^t_{ji} | z_t \in B\} > \sum_{[v_i,v_k]\in\delta^+(v_i)} \max\{y^t_{ik} | z_t \in B\}$. In Section 2.6.3 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 of a graph is obtained by a sequence of deletions and contractions, for the relevance of minors to linear relaxations of the Steiner problem see [CR94a]). Furthermore, the rejoining of flows of different commodities is a common property of all examples for integrality gap 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:

If we 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.

2.6.2 Common Flow

As we have seen in the last subsection, the basic 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 $z_t \in B$. In Figure 2.5, it is easy to see that the flow to z_2 and z_3 ($y^{\{z_2,z_3\}}$) incoming at v_4 is greater than the flow outgoing from v_4. This cannot be the case for x-values corresponding 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 2.6.1, the same network can or cannot show rejoining of flow, depending on the choice of the root. To detect all rejoinings we have to consider all possible roots. Thus, the variables that will be used in the common flow formulation will have the shape $y^{r,B}$ and they describe a flow from z_r to all terminals $z_t \in B \subseteq R^{z_r}$ (we will skip the chosen root in the notation if it is clear from the context).

2.6.3 A Template for Common Flow Formulations

We first state an exhaustive 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).

$$
\begin{aligned}
\boxed{P_{F^R}} \qquad c \cdot X &\rightarrow \min, \\
y^{r,\{z_t\}}(\delta^-(z_r)) &= y^{r,\{z_t\}}(\delta^+(z_r)) - 1 \quad (z_r \in R; z_t \in R^{z_r}), && (13.1)\\
y^{r,\{z_t\}} - y^{r,\{z_s\}} &\leq y^{s,\{z_t\}} \quad (\{z_r, z_s, z_t\} \subseteq R), && (13.2)\\
y^{r,\{z_s\}}_{ji} - y^{r,\{z_t\}}_{ji} &\leq y^{s,\{z_t\}}_{ij} \quad (\{z_r, z_s, z_t\} \subseteq R; [v_i, v_j] \in A), && (13.3)\\
y^{r,B}(\delta^-(v_i)) &\leq y^{r,B}(\delta^+(v_i)) \quad (z_r \in R; B \subseteq R^{z_r}; v_i \in V \setminus (B \cup \{z_r\}), && (13.4)\\
y^{r,B} &\leq y^{r,C} \quad (z_r \in R; B \subset C \subseteq R^{z_r}), && (13.5)\\
y^{r,R^{z_r}}_{ij} + y^{r,R^{z_r}}_{ji} &\leq X_{ij} \quad (z_r \in R; (v_i, v_j) \in E), && (13.6)\\
y^{r,\{z_r\}} &= 0 \quad (z_r \in R), && (13.7)\\
y &\geq 0, && (13.8)\\
X &\in \{0,1\}^{|E|}. && (13.9)
\end{aligned}
$$

It follows from the proof of Lemma 12 that the constraints (13.2) and (13.3) redirect the flows from a 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 (13.7) 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 (13.1) guarantee

[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.

an outflow of one unit out of z_r, and (together with the previous observation) an inflow of one unit in z_t. Together with the constraints (13.4) (with $B = \{z_t\}$) and (13.8) this guarantees a flow of one unit from z_r to each terminal $z_t \in R^{z_r}$.
The constraints (13.4) 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 (13.5) set each common flow from a root z_r to a set of terminals C to at least the maximum of all flows from z_r to a subset $B \subset C$ of the terminals.
Finally the constraints (13.6) build the edgewise maximum over all flows from all terminals in the edge variables X.

We do not know of any network where the relaxation LP_{F^R} 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 2.11.1). Note that due to the exponential number of constraints *and* variables the possibility that there is no integrality gap is not ruled out by known complexity arguments (and common assumptions).

2.6.4 Polynomial Variants

We will now show how to derive polynomial-size relaxations from LP_{F^R}: We limit the number of terminal sets B polynomially, 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 resulting 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 that 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 $B \in \mathcal{B} = \{C \subseteq R^{z_r} \mid |C| \in \{1, 2, |R| - 1\}\}$.

$$
\begin{aligned}
\boxed{P_{F^2}} \qquad c \cdot x &\rightarrow \min, \\
y^{r,\{z_t\}}(\delta^-(z_r)) &= y^{r,\{z_t\}}(\delta^+(z_r)) - 1 \quad (z_t \in R^{z_r}), && (14.1)\\
y^{r,B}(\delta^-(v_i)) &\leq y^{r,B}(\delta^+(v_i)) \quad (B \in \mathcal{B}; v_i \in V \setminus (B \cup \{z_r\}), && (14.2)\\
y^{r,B} &\leq y^{r,C} \quad (B \subset C \in \mathcal{B}), && (14.3)\\
y^{r,R^{z_r}} &= x, && (14.4)\\
y &\geq 0, && (14.5)\\
x &\in \{0,1\}^{|A|}. && (14.6)
\end{aligned}
$$

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 the 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 the choice of k_2 can weaken 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. On the other hand, for $k_1 \geq 3$ or $k_2 \geq 2$ there is no integrality gap.

The example in Figure 2.6 can be generalized to networks $G[k_1, k_2]$, which produce an integrality gap for any common flow relaxation that is limited by some constants k_1 and k_2 (Figure 2.6 shows the network $G[2, 1]$): Around a central 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 these terminals is connected to all $v_j, j \neq 0, j \neq i$ with edges of cost $k_1 + 1$. Then connect the central vertex to k_2 different terminals

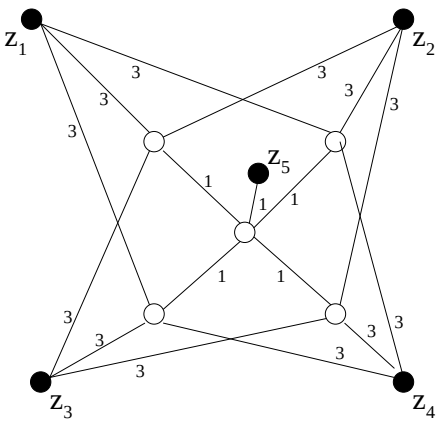


Figure 2.6: $v(LP_F) = 14\frac{1}{3} < v(LP_{F^{2,1}}) = 14\frac{2}{3} < v(LP_{F^R}) = 15 = v(P_{F^R})$.

$(b_1, b_2, \ldots, b_{k_2})$ 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-values 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 $k_2 + (k_1 + 1)(k_1 + 2) + 2 - (k_1 + 1)^{-1}$.

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 $k_1 + k_2 \leq j_1 + j_2$:

if $k_1 \leq j_1, k_2 \leq j_2$**:** 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 integral solution, while $LP_{F^{k_1,k_2}}$ has an integrality gap.

otherwise: The relaxations are incomparable, 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 14 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 terminals a_1, a_2, or a_3 as the root gives the integer optimal value, while with b_1 as root there is an integrality gap. □

2.6.5 Practical Adaptations

We have experimented with different restricted versions of the common flow relaxation, with no definitive conclusion which one is the best. Nevertheless, we can present a strongly restricted, but still very useful approach.

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 $k_2 = 1$, typically there 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 anymore, we can replace the $y^{R^{z_r}}$ with x-variables.

In a cut-and-price framework (see Section 2.11) 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 needed, all variables and all flow conservation constraints have to be inserted to get something useful. Therefore the cut-based formulations are more suitable in this context. In analogy to P_C, we can use constraints of the form $y^{\{z_i\}}(\delta^-(W)) \geq 1$ 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 (13.5). 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:

$$
\begin{aligned}
\boxed{P_{C'}} \quad c \cdot x &\rightarrow \min, \\
x(\delta^-(W)) &\geq 1 \quad (z_r \notin W, R \cap W \neq \emptyset), && (15.1)\\
y^{R\setminus\{z_r,z_i\}}(\delta^-(W) \cap \delta^-(z_i)) + x(\delta^-(W) \setminus \delta^-(z_i)) &\geq 1 \quad (z_r \notin W, R^{z_i} \cap W \neq \emptyset), && (15.2)\\
x(\delta^-(v_i)) &\leq x(\delta^+(v_i)) \quad (v_i \in V \setminus R), && (15.3)\\
y^{R\setminus\{z_r,z_i\}}(\delta^-(z_i)) &\leq x(\delta^+(z_i)) \quad (z_i \in R^{z_r}), && (15.4)\\
y &\geq 0, && (15.5)\\
x &\in \{0,1\}^{|A|}. && (15.6)
\end{aligned}
$$

This new formulation has less 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 2.11.1.

2.6.6 Relation to Other Relaxations

Lemma 15 The relaxation $LP_{F^{1,1}}$ is equivalent to LP_{F+FB}.

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 12, 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 (13.2) and (13.3). □

Lemma 16 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 they are also strictly stronger, consider the network $G[1,1]$. If we choose

$a_1, a_2,$ or a_3 as root, $LP_{C'}$ has an integrality gap, but not LP_{F^2}. 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. □

Lemma 17 The relaxation LP_{F^2} is strictly stronger then LP_{2t+FB}.

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 of (14.2) and (14.4), x satisfies also the flow-balance constraints (11.1), so LP_{F^2} is stronger than LP_{2t+FB}. The following example shows that it is even strictly stronger. □

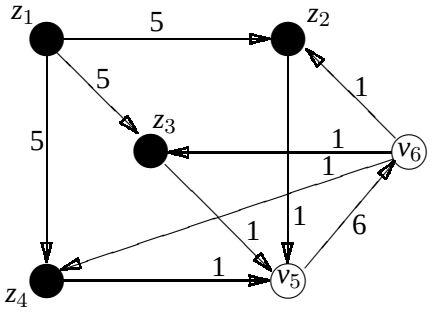


Figure 2.7: Example with $v(LP_{2t}) < v(LP_{F+FB}) = v(LP_{F^2}) = v(P_{F^2})$.

Example 4 In Figure 2.7, 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.

Both relaxations LP_{F^2} and LP_{2t} make 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 Figure 2.7 (setting all x-variables to 0.5).

2.7 A Hierarchy of Relaxations

2.7.1 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 21.

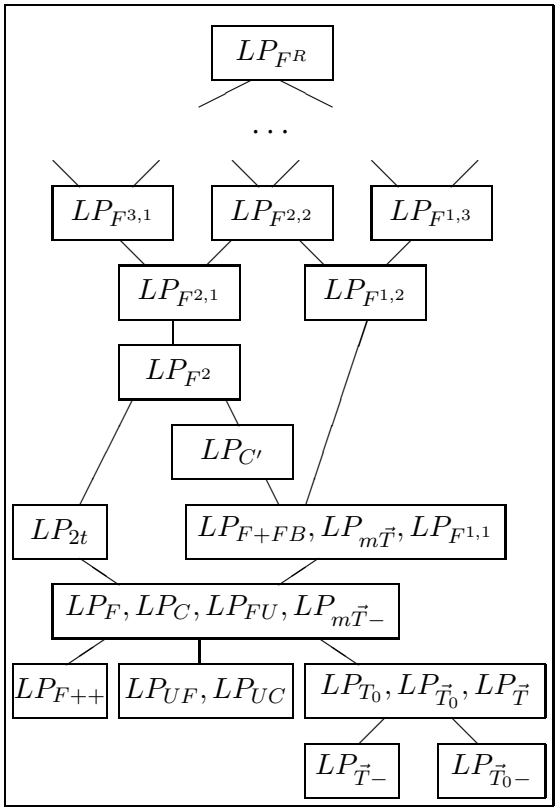


Figure 2.8: Hierarchy of relaxations.

2.7.2 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 5 or 6); 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 7 or Theorem 11). 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.

2.8 Steiner Trees and Minimum Spanning Trees in Hypergraphs

For geometric Steiner problems (Section 1.3), an approach based on full Steiner trees has been successful (Warme, Winter and Zachariasen [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, can be decomposed into its full Steiner trees by splitting its inner terminals (a full Steiner tree (FST) is a tree with no inner terminals, i.e., all terminals have degree 1). 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 (all FSTs of) a Steiner minimal tree. In the second phase, the FST concatenation phase, one chooses a subset of the generated FSTs whose concatenation yields an Steiner minimal tree. Although there are point sets that give rise to an exponential number of FSTs in the first phase, usually only a linear number of FSTs are generated, and empirically the bottleneck of this approach has usually been the second phase, where originally methods like backtracking or dynamic programming have been used. A breakthrough occurred as Warme [War98] observed that FST concatenation can be reduced to finding a minimum spanning tree in a hypergraph whose vertices are the terminals and whose hyperedges 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.

In this section, we first compare the mentioned relaxation to some other, new relaxations of the MSTH problem. We show that all these relaxations are equivalent (yield the same value), and thereby refute a conjecture in the literature that a (straightforward) directed version of the original relaxation might be stronger. Then we compare these relaxations with other relaxations that are based directly on formulations of the Steiner problem in graphs. Note that the union of (the edge sets of) the FSTs generated in the first phase is a graph and the FST concatenation problem reduces to solving the classical Steiner problem in this graph. Some experimental results, both on the quality of the relaxations and on FST concatenation methods based on them, can be found in Sections 2.13 and 5.4.1, demonstrating that our program package, which is designed for general networks, is an efficient alternative to the MSTH-based method. Although the approach used by us was known, previous attempts had lead to the assumption that it is unlikely to become competitive to the MSTH approach [War98].

Further Definitions for the MSTH approach

A Steiner tree T for a subset $Q \subseteq R$ is called a **full Steiner tree (FST)** if all terminals in Q are leaves of T. Let F be the set of FSTs constructed in the FST generation phase. By identifying each FST $T \in F$ with its set of terminals, we get a hypergraph $H = (R, F)$. For each FST T, let c_T be the sum of its edge weights. Any FST T can be rooted from each of its k leaves, leading to a set of directed FSTs $\{\vec{T}_1, \ldots, \vec{T}_k\}$. We denote the set of directed FSTs generated from F in this way by $\vec{F}$. In the following, we use the term FST both for the tree T and the corresponding hyperedge in H, the meaning should be clear from the context.

Cuts in hypergraphs can be defined similarly to those in usual graphs (Section 2.1.1); here we use Δ instead of δ (for example, $\Delta(S) := \{T \in F \mid T \cap S \neq \emptyset, T \cap \bar{S} \neq \emptyset\}$).

2.8.1 Minimum Spanning Trees in Hypergraphs: Formulations and Relaxations

We begin with a formulation of Warme [War98] for the MSTH problem:

$$\boxed{P_{FST}} \qquad \sum_{T \in F} c_T X_T \;\rightarrow\; \min,$$

$$\sum_{T \in F} (|T|-1) X_T = |R|-1, \tag{16.1}$$

$$\sum_{T,\, T \cap S \neq \emptyset} (|T \cap S|-1) X_T \leq |S|-1 \quad (\emptyset \neq S \subset R), \tag{16.2}$$

$$X_T \in \{0,1\} \quad (T \in F). \tag{16.3}$$

Lemma 18 Any feasible solution of P_{FST} describes a spanning tree for the hypergraph (R, F) and vice versa.

Proof: A proof (with slightly different syntax) is given in [War98]. □

Using the directed counterpart of F and following the same line as for minimum spanning trees for usual graphs in Magnanti and Wolsey [MW95], we get the following integer program:

$$\boxed{P_{FST\vec{}}} \qquad \sum_{\vec{T} \in \vec{F}} c_{\vec{T}} x_{\vec{T}} \;\rightarrow\; \min,$$

$$\sum_{\vec{T} \in \vec{F}} (|\vec{T}|-1) x_{\vec{T}} = |R|-1, \tag{17.1}$$

$$\sum_{\vec{T},\, \vec{T} \in \Delta^-(z_t)} x_{\vec{T}} = 1 \quad (z_t \in R^{z_1}), \tag{17.2}$$

$$\sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} (|\vec{T} \cap S|-1) x_{\vec{T}} \leq |S|-1 \quad (\emptyset \neq S \subset R), \tag{17.3}$$

$$x_{\vec{T}} \in \{0,1\} \quad (\vec{T} \in \vec{F}). \tag{17.4}$$

It is easy to see that $P_{FST\vec{}}$ is a valid formulation of the MSTH problem.

Lemma 19 $LP_{FST\vec{}}$ is equivalent to LP_{FST}.

Proof: The equivalence can be shown by a (proper) choice of the variables representing each FST T and corresponding directed FSTs $\vec{T}_1, \ldots, \vec{T}_k$ such that $X_T = x_{\vec{T}_1} + \cdots + x_{\vec{T}_k}$. The basic ideas are similar to those in the proof of Lemma 6 in Section 2.3.3. Details can be found in [PV01d]. □

Now consider the following cut-formulation of the MSTH problem:

$$\boxed{P_{FSC}} \qquad \sum_{\vec{T} \in \vec{F}} c_{\vec{T}} x_{\vec{T}} \;\rightarrow\; \min,$$

$$\sum_{\vec{T} \in \vec{F}} (|\vec{T}|-1) x_{\vec{T}} = |R|-1, \tag{18.1}$$

$$\sum_{\vec{T},\, \vec{T} \in \Delta^-(S)} x_{\vec{T}} \geq 1 \quad (z_1 \notin S,\ S \cap R^{z_1} \neq \emptyset), \tag{18.2}$$

$$x_{\vec{T}} \in \{0,1\} \quad (\vec{T} \in \vec{F}). \tag{18.3}$$

It can be verified (and follows from the proof of the next lemma) that P_{FSC} is a valid formulation of the MSTH problem.

Lemma 20 $LP_{FS\vec{T}}$ is equivalent to LP_{FSC}.

Proof: First observe that for any x feasible for LP_{FSC} summing (18.2) for all $z_t \in R^{z_1}$ we have:

$$|R| - 1 \leq \sum_{z_t \in R^{z_1}} \sum_{\vec{T} \in \Delta^-(z_t)} x_{\vec{T}} \leq \sum_{\vec{T}} (|\mathit{leaves}(\vec{T})|) x_{\vec{T}} = \sum_{\vec{T}} (|\vec{T}| - 1) x_{\vec{T}} \, .$$

Together with (18.1) this means that x satisfies (17.2). It follows that $\sum_{\vec{T} \in \Delta^-(z_1)} x_{\vec{T}} = 0$.

Now consider any x that is feasible for LP_{FST} or LP_{FSC}; we will show that in either case x is feasible for both. For any partition $S \dot{\cup} \bar{S} = R$, we have:

$$\begin{aligned}
\sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} (|\vec{T} \cap S| - 1) x_{\vec{T}} &= \sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} (|\mathit{leaves}(\vec{T}) \cap S| + |root(\vec{T}) \cap S| - 1) x_{\vec{T}} \\
&= \sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} |\mathit{leaves}(\vec{T}) \cap S| x_{\vec{T}} - \sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset,\, root(\vec{T}) \notin S} x_{\vec{T}} \\
&= \sum_{z_t \in S} \sum_{\vec{T} \in \Delta^-(z_t)} x_{\vec{T}} - \sum_{\vec{T},\, \vec{T} \in \Delta^-(S)} x_{\vec{T}} \\
&= |S \cap R^{z_1}| - \sum_{\vec{T},\, \vec{T} \in \Delta^-(S)} x_{\vec{T}}
\end{aligned}$$

Now there are two cases:

(I) $z_1 \in \bar{S}$**:**

$$\sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} (|\vec{T} \cap S| - 1) x_{\vec{T}} = |S| - \sum_{\vec{T},\, \vec{T} \in \Delta^-(S)} x_{\vec{T}}$$

This means that x satisfies (17.3) if and only if it satisfies (18.2).

(II) $z_1 \in S$**:**

$$\sum_{\vec{T},\, \vec{T} \cap S \neq \emptyset} (|\vec{T} \cap S| - 1) x_{\vec{T}} = |S| - 1 - \sum_{\vec{T},\, \vec{T} \in \Delta^-(S)} x_{\vec{T}}$$

So x satisfies (17.3), because it is non-negative.

□

Note that we have actually proved a stronger result: The sets of feasible solutions (and corresponding polyhedra) are identical for both relaxations. With respect to optimal solutions, our assumption that the edge costs are positive leads directly to the observation that $\sum_{\vec{T} \in \Delta^-(z_1)} x_{\vec{T}} = 0$. A more detailed analysis (similar to our proofs of Lemmas 8 and 9 in Section 2.4) leads to the observation that for any optimal solution for LP_{FSC} without (18.1) and for every $z_t \in R^{z_1}$, it holds: $\sum_{\vec{T} \in \Delta^-(z_t)} x_{\vec{T}} = 1$. So dropping the constraints (18.1) does not change the optimal solution value of LP_{FSC}.

2.8.2 Relation to the Relaxations of the Steiner Problem in Graphs

Lemma 21 LP_{FSC} is (strictly) stronger than LP_C and LP_{C+FB}.

Proof: Let $\hat{x}$ be an optimal solution of LP_{FSC}. For each arc $[v_i, v_j] \in A$ that is part of directed FSTs $\vec{T}_1, \ldots, \vec{T}_l$, let $x^*_{ij} := \hat{x}_{\vec{T}_1} + \cdots + \hat{x}_{\vec{T}_l}$. It is easy to verify that x^* is feasible for LP_C and yields the same value as $v(LP_{FSC})$. Now consider a non-terminal v_j. Any directed FST containing an arc $[v_i, v_j] \in \delta^-(v_j)$ includes also at least one arc $[v_j, v_k] \in \delta^+(v_j)$, so x^* satisfies also the flow-balance constraints (11.1).

The following example shows that $v(LP_{FSC})$ can indeed be larger than $v(LP_{C+FB})$. □

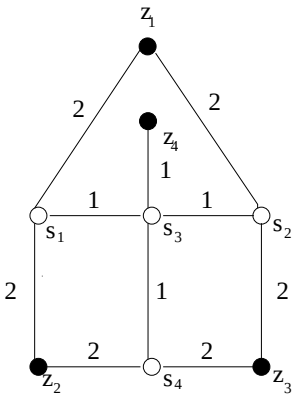


Figure 2.9: Example with $v(LP_{C+FB}) < v(LP_{FSC}) = v(P_{FSC})$

Example 5 In Figure 2.9, we have $v(P_{FSC}) = v(LP_{FSC}) = 9$, $v(LP_C) = v(LP_{C+FB}) = 8.5$.

Lemma 22 LP_{FSC} is incomparable to LP_{2t} or $LP_{C'}$ or LP_{F^2}.

Proof: The lemma follows from the following two examples. □

Example 6 After contracting the edge (s_3, z_4) in Figure 2.9, we have $v(LP_{FST}) = 7.5$ (by setting to 0.5 the X-values for the FSTs (z_1, s_1, z_4, z_2), (z_1, s_2, z_4, z_3) and (z_4, s_4, z_2, z_3)), but $v(LP_{2t}) = v(LP_{C'}) = v(LP_{F^2}) = v(P_{F^2}) = 8$.

Note that this is also an example where the choice of FSTs in the first phase influences the value of LP_{FST}: If only the FSTs (z_1, s_1, z_2), (z_1, s_1, z_4), (z_1, s_2, z_3), (z_1, s_2, z_4) and (z_4, s_4, z_2, z_3) are generated in the first phase (a Steiner minimal tree can be constructed by the concatenation of the second and the last FST), then $v(LP_{FST}) = v(P_{FST}) = 8$. Note also that $v(LP_C) = v(LP_{C+FB}) = 7.5$ in both cases.

Example 7 In Figure 2.9, if we choose z_4 as the root we get $v(LP_{2t}) = v(LP_{C'}) = v(LP_{F^2}) = 8.5$, but $v(P_{FSC}) = v(LP_{FSC}) = 9$.

We note that the argumentation above can be extended to $LP_{F^{k_1,k_2}}$ for restricted k_1, k_2.

2.9 Approximation: (Primal-) Dual Algorithms

To use an LP relaxation algorithmically, many approaches are based on the LP duality theory. Any feasible solution to the dual of such a relaxation provides a lower bound for the original problem. The classical dual-ascent algorithms construct a dual feasible solution step by step, in each step increasing some dual variables while preserving dual feasibility. This is also the main idea of many recent approximation algorithms based on the primal-dual method, where an approximate solution to the original problem and a feasible solution to the dual of an LP relaxation are constructed simultaneously. The performance guarantee is proved by comparing the values of both solutions [GW96].

In this section we study some old and new dual-ascent based algorithms for computing lower and upper bounds for the Steiner problem. Two approximation ratios will be of concern: the ratio between the upper bound and the optimum, and the ratio between the (integer) optimum and the lower bound. The main emphasis here will be on lower bounds, with upper bounds mainly used in a primal-dual context to prove a performance guarantee for the lower bounds. Despite the fact that calculating tight lower bounds efficiently is highly desirable (for example in the context of exact algorithms (Chapter 5) or reduction tests (Chapter 3)), this issue has found much less attention in the literature. For powerful techniques for computing upper bounds, see Chapter 4.

We begin with the classical primal-dual algorithm for the (generalized) Steiner problem based on the undirected cut relaxation. We give some new insights into this algorithm, and show that its bounds can even be computed in time $O(m + n \log n)$, improving the previous time bound of $O(n^2 \log n)$. Then we study a classical dual-ascent approach based on the directed cut relaxation, and show that it cannot guarantee a constant approximation ratio for the generated lower (or upper) bounds. Finally, we introduce a new primal-dual algorithm based on the directed cut relaxation which guarantees a ratio of at most 2 between the upper and lower bounds, while producing tight lower bounds empirically. Detailed computational experiments and some additional explanations are given in [PV00b].

For LP_C, the dual linear program DLP_C uses dual variables u_W for each Steiner cut $(\overline{W}, W)$ ($z_1 \in \overline{W}$ and $R^{z_1} \cap W \neq \emptyset$):

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

$$\sum_{W,\, [v_i,v_j] \in \delta^-(W)} u_W \;\leq\; c_{ij} \quad ([v_i, v_j] \in A), \tag{19.1}$$

$$u \;\geq\; 0. \tag{19.2}$$

The constraints $\sum u_W \leq c_{ij}$ are called the **(cut) packing constraints**. The dual of the undirected program LP_{UC} is similar, replacing directed cuts and edges with undirected ones.

2.9.1 Undirected Cuts: A Primal-Dual Algorithm

Some of the best-known primal-dual approximation algorithms are designed for a class of constrained forest problems which includes the Steiner problem (see [GW95]). These heuristics are essentially dual-ascent algorithms based on undirected cut formulations. For the Steiner problem, such an algorithm guarantees an upper bound of $2 - 2/r$ on the ratio between the values of the provided primal and dual solutions. This is the best possible guarantee when using the undirected cut relaxation LP_{UC}, since it is easy to construct instances (even with $r = n$) where the ratio $v(P_{UC})/v(LP_{UC})$ is exactly $2 - 2/r$ (see for example [GB93]). In the following, we briefly describe such an algorithm when restricted to the Steiner problem, show how to make it much faster for this special case, and give some new insights into it. We denote this algorithm with PD_{UC} (PD stands for Primal-Dual and UC stands for Undirected Cut).

The algorithm maintains a forest F, which initially consists of isolated vertices of V. A connected component S of F is called an active component if $(\overline{S}, S)$ defines a Steiner cut. In each iteration, dual variables corresponding to active components are increased uniformly until a new packing constraint becomes tight, i.e. the reduced cost $c(e) - \sum u_S$ of some edge e becomes zero, which is then added to F (ties are broken arbitrarily). The algorithm terminates when no active component is left; at this time, F defines a feasible Steiner tree and $\sum_{(\overline{S},S) \text{ Steiner cut}} u_S$ represents a lower bound on the weight of any Steiner tree for the observed instance. In a subsequent pruning phase, every edge of F that is not on a path (in F) between two terminals is removed. In [GW95], it is shown how to make this algorithm (for the generalized problem) run in $O(n^2 \log n)$ time; see also [GGW93, Kle94] for some improvements.

When restricted to the Steiner problem and as far as the constructed Steiner tree is considered, the algorithm PD$_{UC}$ is essentially the DNH (Section 1.2), implemented by an interleaved computation of shortest paths trees out of terminals and a minimum spanning tree for the terminals with respect to their distances. In fact, every Steiner tree T provided by Mehlhorn's $O(m + n \log n)$ time implementation of DNH can be considered as a possible result of PD$_{UC}$. We observed that even the lower bound calculation can be performed in the same time: Let T' be a minimum spanning tree for R provided by Mehlhorn's implementation of DNH and let $e'_1, \ldots, e'_{r-1}$ be its edges in non-decreasing cost order. The algorithm PD$_{UC}$ increases all dual variables corresponding to the initially r active components by $\frac{c'(e'_1)}{2}$, then the components corresponding to the vertices of e'_1 are merged. The dual variables of the remaining $r-1$ components are increased by $\frac{c'(e'_2)-c'(e'_1)}{2}$ (which is possibly zero) before the next two components are merged, and so on. Therefore, the lower bound provided by PD$_{UC}$ is (defining $c'(e'_0) := 0$) simply $\sum_{i=1}^{r-1}(r-i+1)\frac{c'(e'_i)-c'(e'_{i-1})}{2} = \frac{1}{2}(c'(e'_{r-1}) + \sum_{i=1}^{r-1} c'(e'_i)) = \frac{1}{2}(c'(e'_{r-1}) + c'(T'))$, which can be computed in $O(r)$ time once T' is available.

Using this line of argumentation, the results concerning the bounds provided by PD$_{UC}$ can be proven without the notion of primal-dual algorithms altogether:

Lemma 23 Let T_{opt} be an optimal Steiner tree for an instance (G, R) of the Steiner problem and T' a minimum spanning tree in the distance network $D_G(R)$ with edges $e'_1, \ldots, e'_{r-1}$ as described before. Define $L := \frac{1}{2}(c'(e'_{r-1}) + c'(T'))$. It holds: $L \leq c(T_{opt}) \leq c'(T') \leq (2 - \frac{2}{r})L$.

Proof: Consider a preorder walk around the tree T_{opt} beginning with an arbitrary terminal as the root. Such a walk will traverse every edge of T_{opt} exactly twice. Introduce a new edge e''_j with the same cost $c''(e''_j)$ as the corresponding path in the walk every time a new terminal is encountered and when the walk terminates at the root. Let $e''_1, \ldots, e''_r$ be the so constructed edges in non-decreasing cost order. The edges $e''_1, \ldots, e''_{r-1}$ build a tree T'' with the cost $c''(T'') = c''(e''_1) + \cdots + c''(e''_{r-1})$ which spans all terminals. Obviously, T'' is no cheaper than T'; and its longest edge e''_{r-1} is not cheaper than e'_{r-1} (otherwise, replacing e'_{r-1} by a suitable edge of T'' would create a tree spanning all terminals and cheaper than T'). So we have: $2c(T_{opt}) = c''(T'') + c''(e''_r) \geq c''(T'') + c'(e'_{r-1}) - c''(e''_{r-1}) + c''(e''_r) \geq c''(T'') + c'(e'_{r-1}) - c''(e''_r) + c''(e''_r) = c''(T'') + c'(e'_{r-1}) \geq c'(T') + c'(e'_{r-1}) = 2L$.
On the other hand, we have: $2L = c'(T') + c'(e'_{r-1}) \geq c'(T') + \frac{1}{r-1}c'(T') = (1 + \frac{1}{r-1})c'(T')$, so $c(T_{opt}) \leq c'(T') \leq (2 - \frac{2}{r})L$. □

From this new viewpoint at PD$_{UC}$ we get some insight about the gap between the provided upper and lower bounds. Assuming that the cost of T' is not dominated by the cost of its longest edge and that the Steiner tree corresponding to T' is not much cheaper than T' itself (which is usually the case), the ratio between the upper and lower bound is nearly two; and this suggests that either the lower bound, or the upper bound, or both are not really tight.

Empirically, results on different types of instances (from SteinLib [Ste97]) show an average gap of about 45% (of optimum) between the upper and the lower bounds calculated by PD_{UC}. This is in accordance with the relation we established above between these two values. This gap is mainly due to the lower bounds, where the gap to optimum is typically over 30%. So although this heuristic can be implemented to be very fast empirically (small fractions of a second even for fairly large instances (several thousands of vertices)), it is not suitable for computing tight bounds.

2.9.2 Directed Cuts: A Dual-Ascent Algorithm

In the search for an approach for computing tighter lower and upper bounds, the directed cut relaxation is a promising alternative. Although no better upper bound than the $2 - 2/r$ one from the previous section is known on the integrality gap of this relaxation, the gap is conjectured to be much closer to 1, and the worst instance known has an integrality gap of approximately $8/7$ (Section 2.6.1). There are many theoretical and empirical investigations that indicate that the directed relaxation is a much stronger relaxation than the undirected one (see for example [CGR92, CR94a]). 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 often been sufficient to solve the instance exactly (without branching) through bound-based reduction techniques (Section 3.3). So, a really interesting problem is how to calculate (or sufficiently approximate) a solution for LP_C.

In [PV01c], we could achieve impressive empirical results (including extremely tight lower and upper bounds) using this relaxation. In that work, extensions of a dual ascent algorithm of Wong [Won84] played a major role. Although many works on the Steiner problem use variants of this heuristic (see for example [Dui93, HRW92, Voß92]), none of them includes a discussion of the theoretical quality of the generated bounds. In this section, we show that none of these variants can guarantee a constant approximation ratio for the generated lower or upper bounds.

The dual-ascent algorithm in [Won84] is described using the multicommodity flow relaxation. Here we give a short alternative description of it as a dual-ascent algorithm for the (equivalent) relaxation LP_C, which we denote with DA_C. The algorithm maintains a set H of arcs with zero reduced costs, which is initially empty. For each terminal $z_t \in R^{z_1}$, define the component of z_t as the set of all vertices for which there exists a directed path to z_t in H. A component is said to be active if it does not contain the root. In each iteration, an active component is chosen and the dual variable of the corresponding Steiner cut is increased until the packing constraint for an arc in this cut becomes tight. Then the reduced costs of the arcs in the cut are updated and the arcs with reduced cost zero are added to H. The algorithm terminates when no active component is left; at this time, H (regarded as a subgraph of G) is a feasible solution for the observed instance of the (directed) Steiner problem. To get a (directed) Steiner tree, in [Won84] the following method is suggested: Let Q be the set of vertices reachable from z_1 in H. Compute a minimum directed spanning tree for the subgraph of G induced by Q and prune this tree until all its leaves are terminals. In [HRW92], this method is adapted to the undirected version, mainly by computing a minimum (undirected) spanning tree instead of a directed one.

To investigate the quality of the bounds generated by DA_C, two difficulties must be considered. The first one is the choice of the root: although the value $v(LP_C)$ for an instance of (undirected) Steiner problem is independent of the choice of the root [GM93], the lower bound generated by DA_C is not, so the argumentation must be independent of this choice. The second difficulty is the choice of an active component in each iteration. In the original work of Wong [Won84], the chosen component is merely required to be a so-called root component. A component S corresponding to a terminal z_t is called a root component if for any other terminal z_s in this component, there is a

path from z_t to z_s in H. This is equivalent to S being a minimal (with respect to inclusion) active component. An empirically more successful variant uses a size criterion: at each iteration, an active component of minimum size is chosen (see [Dui93, PV01c]). Note that such a component is always a root component. So, in this context it is sufficient to study the variant based on the size criterion. For reference in later sections, we give a high-level description of DA_C using the size criterion, which we simply denote with **DUAL-ASCENT**:

- 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]). Although the algorithm usually runs much faster than this bound would suggest, we have constructed instances on which every dual-ascent algorithm following the same scheme must perform $\Theta(n^4)$ operations.

Now we show that the bounds generated by DA_C can deviate arbitrarily from optimum.

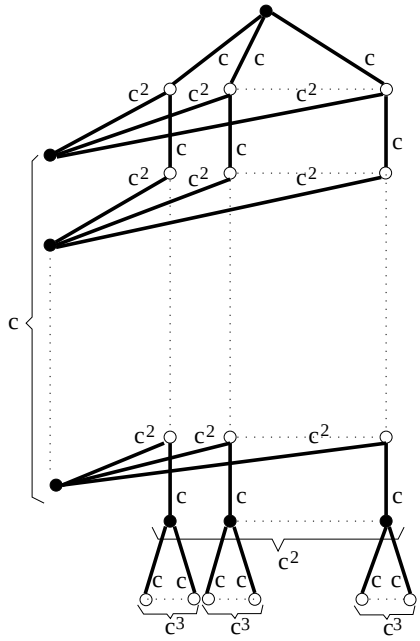


Figure 2.10: Arbitrarily bad case for DA_C.

Example 8 In Figure 2.10, there are $c^2 + c + 1$ terminals (filled circles); the top terminal is considered as the root. The edges incident with the left c terminals have costs c^2, all the other edges have costs c. According to the size criterion, each of the terminals (i.e. their components) at the left is chosen twice before any of the terminals at the bottom can be chosen a second time. But then, there is no

active component anymore and the algorithm terminates. So, the lower bound generated by DA$_C$ is in $\Theta(c^3)$. On the other hand, it is easy to see that for this instance: $v(LP_C) = v(P_C) \in \Theta(c^4)$.

Now imagine c copies of this graph sharing the top terminal. For the resulting instance, we have $v(LP_C) = v(P_C) \in \Theta(c^5)$; but the lower bound generated by DA$_C$ will be in $\Theta(c^4)$ independent of the choice of the root, because the observation above will remain valid in at least $c - 1$ copies.

Now we turn to upper bounds: By changing the costs of the edges incident to the left terminals from c^2 to $c + \epsilon$ (for a small ϵ) in Figure 2.10 we get an instance for which the ratio between the upper bound calculated by the algorithm described in this section and $v(P_C)$ can be arbitrarily large. This is also the case for all other approaches in the literature for computing upper bounds based on the graph H provided by DA$_C$, because $v(P_C) \in \Theta(c^3)$ for such an instance, but there is no solution with cost $o(c^4)$ in the subgraph H generated by DA$_C$.

Despite its bad performance in the worst case, the algorithm typically provides fairly tight lower bounds, with average gaps ranging from a small fraction of a percent to about 2%, depending on the type of instances. For example, out of the 20 D-instances of the OR-Library, a run of DUAL-ASCENT yields a lower bound equal to the optimum (i.e., $v(P_C)$) for 12 instances; the average gap between lower bound and optimum is 0.4%. The upper bounds are not good, with average gaps from 8% to 30%, again depending on the type of instances. The running times are still quite tolerable (less than one second even for fairly large instances).

The described choices in DA$_C$, namely the choice of the root and the choice of z_t in each iteration, can also have a great practical impact. For this reason we start DA$_C$ 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. Further improvements can be achieved by using the reductions and upper bound calculations, which are done in combination with DA$_C$ (see Sections 3.3.2 and 4.4).

We also tested different criteria for the choice of z_t in each iteration (our standard criterion is the size criterion). We had some success with the following idea that tries to guide DA$_C$ with the help of a heuristically constructed Steiner tree: Assume that the upper bound is already optimal. DA$_C$ 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.

Here a question arises naturally: Is there another (efficient) strategy that always leads to better lower bounds or even $v(LP_C)$ itself? Note that the latter would not be surprising from a complexity theory point of view, because $v(LP_C)$ can be calculated in polynomial time. This problem is particularly relevant if an instance is to be solved to optimality: although the lower bounds generated by DA$_C$ usually come close to $v(LP_C)$, the remaining gap is sometimes larger than desired in the context of exact algorithms. If such instances are very large, a modification of DA$_C$ seems to be the most efficient approach. A principal difficulty here is that LP_C might have only fractional optimal solutions and so $v(LP_C)$ might be fractional even for integer edge costs; something which is never the case for the solutions provided by DA$_C$. But this could be remedied by allowing dual variables to be increased by a suitable fraction of (reduced) edge costs (see also the next subsection). So the question here is: Is there a variant of DA$_C$, probably involving a more sophisticated strategy for choosing active components and allowing arbitrary increment of dual variables, which always reaches the value $v(LP_C)$? We answer this question in the negative by presenting an instance for which no order of choosing the components can lead to the "correct" cuts.

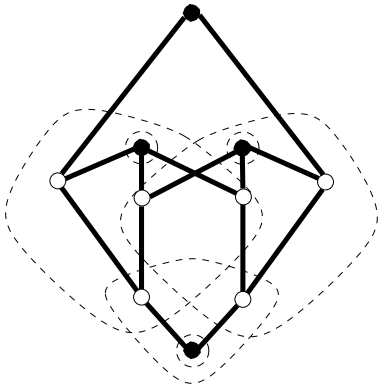


Figure 2.11: No choice in DA$_C$ leads to correct cuts.

Example 9 In Figure 2.11, the top terminal is considered as the root; and all edges have costs 1. It is easy to see that for this instance, $v(P_C) = v(LP_C) = 6$ (the relevant cuts are sketched using dashed lines). Now consider the behavior of DA$_C$: If the dual variables corresponding to the terminals in the middle are not raised over 1, the dual variables of the components corresponding to the terminal at the bottom will sum to 3, and the generated lower bound will be 5. Now assume that the dual variables corresponding to a terminal in the middle sum to $1 + \alpha$, $0 < \alpha \leq 1$. In the network with the resulting reduced costs, it is easy to find a Steiner tree with the cost $5 - 2\alpha$, so the generated lower bound would be at most $(1 + \alpha) + (5 - 2\alpha) = 6 - \alpha$. Again, the argumentation can be made independent of the choice of the root by letting several copies of this graph share the top terminal.

2.9.3 Directed Cuts: A New Primal-Dual Algorithm

The previously described heuristics had complementary advantages: The first, PD$_{UC}$, guarantees an upper bound of 2 on the ratio between the generated upper and lower bounds, but empirically, it does not perform much better than in the worst case. The second one, DA$_C$, cannot provide such a guarantee, but empirically it performs much better than the first one, especially for computing lower bounds. In this section we describe a new algorithm that combines both features.

An idea which is promising at the first sight is the direct application of the primal-dual method of PD$_{UC}$ (simultaneous increasing of all dual variables corresponding to active components and merging components that share a vertex) to the directed cut relaxation. The obtained lower bound (using an additional trick to make it independent of the choice of the root) is the weight of a minimum spanning tree in a graph $\hat{G}$ similar to the graph G' defined in Section 1.2, but with the cost of each edge (z_i, z_j) defined as $\hat{c}(z_i, z_j) := \min\{\min\{d_G(z_i, v_k), d_G(z_j, v_l)\} + c(v_k, v_l) \mid v_k \in N(z_i), v_l \in N(z_j)\}$. We will use this lower bound $\hat{l}$ in Section 3.3.1 in the context of reduction techniques. It holds that $\hat{l} \geq \frac{1}{2}c'(T')$, where T' is a minimum spanning tree for G': Let $z_i - \hat{v}_k - \hat{v}_l$ be the path in G corresponding to the edge (z_i, z_j) in $\hat{G}$. Since $\hat{v}_l \in N(z_j)$, we have $d_G(z_j, \hat{v}_l) \leq d_G(z_i, \hat{v}_k) + c(\hat{v}_k, \hat{v}_l)$, so $\hat{c}(z_i, z_j) \geq \frac{1}{2}(d_G(z_i, \hat{v}_k) + c(\hat{v}_k, \hat{v}_l) + d_G(z_j, \hat{v}_l)) \geq \frac{1}{2}c'(z_i, z_j)$, and the claimed relation between the weights of the corresponding minimum spanning trees follows straightforwardly. So, we have again a ratio of at most 2 between the upper and the lower bound, and we can calculate both bounds in $O(m + n \log n)$ time, as it was the case for PD$_{UC}$; although the two methods for computing lower bounds are incomparable (neither method always generate tighter bounds than the other). But the main drawback of PD$_{UC}$ remains: Empirically, the generated lower bounds are again not nearly as tight as those provided by DA$_C$.

The main idea for a successful new approach is not to merge the components, but to let them grow as long as they are (minimally) active. As a consequence, dual variables corresponding to several cuts that share the same arc may be increased simultaneously. Because of that, the reduced costs of arcs that are in the cuts of many active components are decreased much faster than the other ones and we have constructed instances where a straightforward primal-dual algorithm based on this approach fails to give a performance ratio of two.

Therefore, we group all components that share a vertex together and postulate that in each iteration, the total increase Δ of dual variables corresponding to each *group* containing at least one active component must be the same. If we denote the number of active components in a group Γ with *activesInGroup*(Γ), the dual variable corresponding to each of these components will be increased by $\Delta / activesInGroup(\Gamma)$. Similar to the case of DA$_C$, a component is called active if it does not contain the root or include an active component of another terminal (ties are broken arbitrarily). A terminal is called active if its component is active; and a group is called active if it contains an active terminal (by this definition it is guaranteed that each active root component corresponds to one active terminal). If we denote with *activeGroups* the number of active groups, the lower bound *lower* will be increased in each iteration by $\Delta \cdot$*activeGroups*.

To manage the reduced costs efficiently, a concept like that of distance estimates in the algorithm of Dijkstra is used (see for example [CLR90]). For each arc f, the value $d(f)$ estimates the value of $dGroup$ (amount of uniform increase of group duals, i.e. the sum of Δ-values) that would make f tight (set its reduced cost $\tilde{c}(f)$ to zero). Because of the definition of groups, for an arc f with reduced cost $\tilde{c}(f) > 0$, all active components S with $f \in \delta^-(S)$ will be in the same group Γ. If there are $activesOnArc(f)$ such components, then $d(f)$ should be $\tilde{c}(f) \cdot activesInGroup(\Gamma)/activesOnArc(f) + dGroup$. For updating the d-values we use two further variables for each arc f: *reducedCost*(f) and *lastReducedCostUpdate*(f); they are initially set to $c(f)$ and 0, respectively. If *activesOnArc*(f) and/or *activesInGroup*(Γ) change, the new value for $d(f)$ can be calculated by:

$reducedCost(f) := reducedCost(f) - (dGroup - lastReducedCostUpdate(f)) \cdot$
$\quad activesOnArc_{old}(f)/activesInGroup_{old}(\Gamma)$;
$d(f) := reducedCost(f) \cdot activesInGroup_{new}(\Gamma)/activesOnArc_{new}(f) + dGroup$;
$lastReducedCostUpdate(f) := dGroup$.

Below we give a description of the algorithm PD$_C$ in pseudocode with macros (a call to a macro is to be simply replaced by its body). A priority queue PQ manages the arcs using the d-values as keys. The groups are stored in a disjoint-set data structure *Groups*. Two lists $\vec{H}$ and H store the tight arcs and the corresponding edges. A *Stack* is used to perform depth-first searches from vertices newly added to a component. The array *visited*$[z, v]$ indicates whether the vertex v is in the component of the terminal z; *firstSeenFrom*$[v]$ gives the first terminal whose component has reached the vertex v; *active*$[z]$ indicates whether the terminal z is active; $d[f]$ gives the d-value of the arc f; *activesInGroup*$[\Gamma]$ stores the number of the active components in the group Γ; and *activesOnArc*$[f]$ gives the number of components that have the arc f in their cuts.

$PD_C(G, R, z_1)$
1 *initialize PQ, Groups,* H, $\vec{H}$;
2 **forall** $z \in R^{z_1}$: (initializing the components)
3 *Groups.MAKE-SET*(z); *activesInGroup*$[z] := 1$; *active*$[z]$:=*TRUE*;
4 **forall** $f \in \delta^-(z)$:
5 *activesOnArc*$[f] := 1$; $d[f] := c(f)$; *PQ.INSERT*$(f, d[f])$;
6 **forall** $v \in V$: *visited*$[z, v] :=$ *FALSE*;

visited$[z,z]$:= *TRUE*; *firstSeenFrom*$[z]$:= z;
forall $v \in V \setminus R$: *firstSeenFrom*$[v]$:= 0;
activeGroups := $r-1$; *dGroup* := 0; *lower* := 0;
while *activeGroups* > 0 :
  $f := [v_i, v_j]$:= *PQ.EXTRACT-MIN*(); (get the next arc becoming tight)
  Δ := $d[f]$−*dGroup*; *dGroup* := $d[f]$; *lower* := *lower* + Δ·*activeGroups*;
  mark $[v_i, v_j]$ *as tight*;
  if (v_i, v_j) *is not in* H : (i.e. $[v_j, v_i]$ is not tight)
    H.*APPEND*$((v_i, v_j))$; $\vec{H}$.*APPEND*$([v_i, v_j])$;
  z_i := *firstSeenFrom*$[v_i]$; z_j := *firstSeenFrom*$[v_j]$;
  if $z_i = 0$: *firstSeenFrom*$[v_i]$:= z_j;
  else if *Groups.FIND*$(z_i) \neq$ *Groups.FIND*(z_j) : *MERGE-GROUPS*(z_i, z_j);
  forall *active* $z \in R^{z_1}$:
    if *visited*$[z, v_j]$ **and not** *visited*$[z, v_i]$: *EXTEND-COMPONENT*(z, v_i);
H' := H; $\vec{H}'$:= $\vec{H}$; *PD-PRUNE*$(H', \vec{H}')$;
return H', *lower*; ($upper$: the cost of H')

EXTEND-COMPONENT(z, v_i) (modified depth-first search)
Stack.INIT(); *Stack.PUSH*(v_i);
while not *Stack.EMPTY*() :
  v :=*Stack.POP*();
  if $(v = z_1)$ **or** $(v \in R \setminus \{z\}$ **and** *active*$[v])$:
    REMOVE-COMPONENT(z);
    break;
  if not *visited*$[z, v]$:
    visited$[z, v]$:= *TRUE*;
    forall $[v, w] \in \delta^+(v)$:
      if *visited*$[z, w]$:
        activesOnArc$[[v, w]]$:=*activesOnArc*$[[v, w]] - 1$;
        update the key of $[v, w]$ *in PQ*;
      else :
        if $[w, v]$ *is already tight* : *Stack.PUSH*(w);
        else :
          activesOnArc$[[w, v]]$:=*activesOnArc*$[[w, v]] + 1$;
          update the key of $[w, v]$ *in PQ*;

MERGE-GROUPS(z_i, z_j)
g_i := *Groups.FIND*(z_i); g_j := *Groups.FIND*(z_j);
if *activesInGroup*$[g_i] > 0$ **and** *activesInGroup*$[g_j] > 0$:
  update in PQ the keys of all arcs entering these groups;
  activeGroups := *activeGroups*−1;
Groups.UNION(g_i, g_j); g_{new} :=*Groups.Find*(g_i);
activesInGroup$[g_{new}]$:= *activesInGroup*$[g_i]$ + *activesInGroup*$[g_j]$;

REMOVE-COMPONENT(z)
active$[z]$:=*FALSE*; g :=*Groups.FIND*(z);
update in PQ the keys of all arcs entering g *or the component of* z;
activesInGroup$[g]$:= *activesInGroup*$[g] - 1$;

4 **if** *activesInGroup*$[g] = 0$: *activeGroups* := *activeGroups* -1;

PD-PRUNE$(H', \vec{H}')$
1 **forall** $[v_i, v_j]$ *in* $\vec{H}'$, *in reverse order* :
2 **if** H' *without* (v_i, v_j) *connects all terminals* :
3 H'.*DELETE*$((v_i, v_j))$; $\vec{H}'$.*DELETE*$([v_i, v_j])$;

In PD$_C$, all initializations in Lines 1-9 need $O(rn + a \log n)$ time. The loop in the Lines 10-20 is repeated at most a times, because in each iteration an arc becomes tight and there will be no active terminal (or group) when all arcs are tight. Over all iterations, Line 11 needs $O(a \log n)$ time and Lines 12-20 excluding the macros $O(ar)$ time. Each execution of MERGE-GROUPS needs $O(a \log n)$ time and there can be at most $r - 1$ such executions; the same is true for REMOVE-COMPONENT. For each terminal, the adjacency list of each vertex is considered only once during all executions of EXTEND-COMPONENT, so each arc is considered (and its key is updated in PQ) at most twice for each terminal, leading to a total time of $O(ra \log n)$ for all executions of EXTEND-COMPONENT. So the Lines 1-20 can be executed in $O(ra \log n)$ time. The following lemma enables us to perform the reverse order deletion in PD-PRUNE efficiently.

Lemma 24 Consider a graph $\tilde{H}$ with the edge set H in which the weight of each edge $\tilde{e}$ is the position $p(\tilde{a})$ of the corresponding arc $\tilde{a}$ in the list $\vec{H}$. Let T' be the (edge set of a) tree generated by computing a minimum spanning tree for $\tilde{H}$ and pruning it until it has only terminals as leaves. Then we have: $T' = H'$.

Proof: First notice that there is a unique minimum spanning tree in $\tilde{H}$, since no two edge weights are equal. Now consider the computation of the minimum spanning tree $\tilde{T}$ for $\tilde{H}$ with the algorithm of Kruskal. Let e^* be an edge in H'. By the construction of H', we know that the endpoints of e^* are not connected by edges in $E_1 := \{\tilde{e} \in H' \mid p(\tilde{e}) > p(e^*)\} \cup \{\tilde{e} \in H \mid p(\tilde{e}) < p(e^*)\}$, because adding e^* to E_1 changes the connectivity relation for at least one pair of terminals. Consequently, the endpoints are not connected by edges in $\{\tilde{e} \in \tilde{T} \mid p(\tilde{e}) < p(e^*)\}$, which is a subset of E_1. Thus, e^* is included in $\tilde{T}$ by the algorithm of Kruskal.
Now we have $H' \subseteq \tilde{T}$. No edge $\tilde{e}$ in $\tilde{T} \setminus H'$ can be on a path between two terminals in $\tilde{T}$, because then there would be another path between these two terminals in H' (and hence in $\tilde{T}$) that does not use $\tilde{e}$ and $\tilde{T}$ would contain a cycle. So, no edge in $\tilde{T} \setminus H'$ will be present in T', meaning that $T' \subseteq H'$. Now observe that T' is a tree that contains all terminals. No edges can be added to such a tree without creating cycles or non-terminals of degree 1, both features that are not present in H' by its construction. So we have $H' = T'$. □

Using Lemma 24, PD-PRUNE can be implemented by (mainly) computing a minimum spanning tree in $\tilde{H}$. Since the edges of $\tilde{H}$ are already available in a sorted list, this minimum spanning tree can be computed even in $O(m\ \alpha(m, n))$ time. This leads to a total time of $O(ra \log n)$ for PD$_C$.

Below we show that the ratio between the upper bound *upper* and the lower bound *lower* generated by PD$_C$ is at most 2.

Let $\vec{T}$ be (the arcs of) the directed tree obtained by rooting H' at z_1. For each component S, we denote with *activesInGroupOf*(S) the total number of active components in the group of S.

Lemma 25 At the beginning of each iteration in the algorithm PD$_C$, it holds:

$$\sum_{S \text{ active}} \frac{|\vec{H}' \cap\ \delta^-(S)|}{activesInGroupOf(S)} \leq (2 - \frac{1}{r-1}) \cdot activeGroups.$$

Proof: Several invariants are valid at the beginning of each iteration in PD$_C$:

- **(1)** All vertices in a group are connected by the edges currently in H.
- **(2)** For each active group Γ, at most one arc of $\delta^-(\Gamma)$ will belong to $\vec{T}$, since all but one of the edges in $\delta(\Gamma) \cap H$ will be removed by PD-PRUNE because of (1). So $\vec{T}$ will still be a tree if for each active group Γ, all arcs that begin and end in Γ are contracted.
- **(3)** For each group Γ and each active component $S \subseteq \Gamma$, no arc $[v_i, v_j] \in \delta^-(S)$ with $v_i, v_j \in \Gamma$ will be in $\vec{H}'$, since it is not yet in $\vec{H}$ (otherwise it would not be in $\delta^-(S)$) and if it is added to $\vec{H}$ later, it will be removed by PD-PRUNE because of (1).
- **(4)** For each active group Γ and each arc $[v_i, v_j] \in \vec{T} \cap \delta^+(\Gamma)$, there is at least one active terminal in the subtree $\vec{T}_j$ of $\vec{T}$ with the root v_j. Otherwise (v_i, v_j) would be removed by PD-PRUNE, because all terminals in $\vec{T}_j$ are already connected to the root by edges in H.
- **(5)** Because of (2), (4) and since at least one arc in $\vec{T}$ leaves z_1, it holds: $\sum_{\Gamma \text{ active group}} |\vec{T} \cap \delta^-(\Gamma)| \geq 1 + \sum_{\Gamma \text{ active group}} |\vec{T} \cap \delta^+(\Gamma)|$.
- **(6)** Because of (3), for each active group Γ holds: $\sum_{S \subseteq \Gamma,\ S \text{ active}} |\vec{H}' \cap \delta^-(S)| \leq activesInGroup(\Gamma) \cdot |\vec{H}' \cap \delta^-(\Gamma)|$.

We split $\vec{H}'$ into $\vec{H}' \cap \vec{T}$ and $\vec{H}' \backslash \vec{T}$. Because $\vec{H}'$ and $\vec{T}$ differ only in the direction of some arcs, $\vec{H}' \backslash \vec{T}$ is just $\vec{T} \backslash \vec{H}'$ with reversed arcs. Now we have:

$$
\begin{aligned}
\sum_{S \text{ active}} \frac{|\vec{H}' \cap\ \delta^-(S)|}{activesInGroupOf(S)} &= \sum_{\Gamma \text{ active group}} \ \sum_{S \text{ active},\ S \subseteq \Gamma} \frac{|\vec{H}' \cap \delta^-(S)|}{activesInGroup(\Gamma)} \\
&\leq \sum_{\Gamma \text{ active group}} |\vec{H}' \cap \delta^-(\Gamma)| && \text{(because of (6))} \\
&= \sum_{\Gamma \text{ active group}} |\vec{H}' \cap \vec{T} \cap \delta^-(\Gamma)| + |(\vec{T} \backslash \vec{H}') \cap \delta^+(\Gamma)| \\
&\leq \left(\sum_{\Gamma \text{ active group}} |\vec{H}' \cap \vec{T} \cap \delta^-(\Gamma)| + |\vec{T} \cap \delta^-(\Gamma)| \right) - 1 && \text{(because of (5))} \\
&\leq 2 \cdot activeGroups - 1. && \text{(because of (2))}
\end{aligned}
$$

Because *activeGroups* $\leq r - 1$ this proves the lemma. □

Theorem 26 Let $upper$ and $lower$ be the bounds generated by PD$_C$. It holds that: $\frac{upper}{lower} \leq (2 - \frac{1}{r-1})$.

Proof: Let Δ_i be the value of Δ in the iteration i. For each directed Steiner cut $(\overline{S}, S)$, let u_S be the value of the corresponding dual variable as (implicitly) calculated by PD$_C$ (in iteration i each dual variable u_S corresponding to an active component S is increased by $\Delta_i /$ *activesInGroupOf*(S)). Since all arcs of $\vec{H}'$ have zero reduced costs, we have: $upper = \sum_{f \in \vec{H}'} c(f) = \sum_{f \in \vec{H}'} \sum_{S,\ f \in \delta^-(S)} u_S = \sum_S |\vec{H}' \cap \delta^-(S)| \cdot u_S$. This value is zero at the beginning and is increased by $\sum_{S \text{ active}} |\vec{H}' \cap \delta^-(S)| \cdot \Delta_i /$ *activesInGroupOf*(S) in the iteration i. By Lemma 25, this increase is at most $(2 - \frac{1}{r-1}) \cdot activeGroups \cdot \Delta_i$. Since $lower$ is zero at the beginning and is increased exactly by $activeGroups \cdot \Delta_i$ in the iteration i, we have $upper \leq (2 - \frac{1}{r-1}) \cdot lower$ after the last iteration. □

The following two examples show that the proven approximation ratios for upper and lower bounds are both tight.

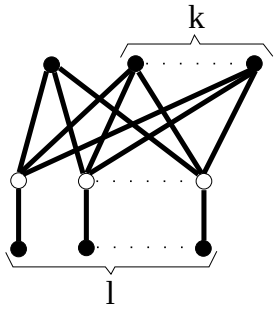


Figure 2.12: $PD_C:\ v(P_C) = v(LP_C) \approx 2 \cdot lower$.

Example 10 In Figure 2.12, the top-left terminal is considered as the root; and all edges have costs 1. Note that this graph is even bipartite. It is easy to see that for this instance, $v(P_C) = v(LP_C) = 2l+k$. But PD_C will deliver the lower bound $l + k + \frac{l+k}{k+1}$. Choosing $l = k^2 >> 1$ we will get a gap of approximately 2.
Again, the argumentation can be made independent of the choice of the root by letting l copies of this graph share the top-left terminal.

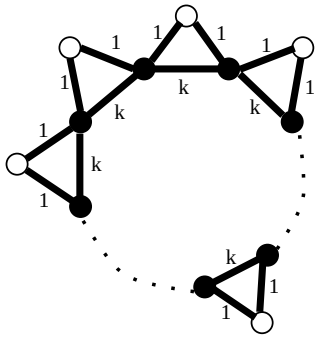


Figure 2.13: $PD_C:\ upper \approx 2v(P_C) = 2v(LP_C)$.

Example 11 In Figure 2.13, setting $k = 1 + \epsilon$ for a small ϵ will ensure that for all terminals not adjacent to the (arbitrary) root, the incident arcs with costs 1 will be inserted into $\vec{H}$ before those with cost k, meaning that PD-PRUNE will remove the latter. So we will have $upper = 2(r-2) + (1+\epsilon)$, whereas $v(P_C) = v(LP_C) = (r-1)(1+\epsilon)$. By choosing a large r we will get a gap of approximately 2.

The discussion of the algorithm PD_C assumes exact real arithmetic. Of course, actual computers cannot handle infinite precision arithmetic; and simply replacing real numbers with floating-point numbers is not appropriate due to the unpredictable perturbations caused by the roundoff errors. But even if we adopt the (usual) assumption that all numbers in the input are integers, using exact arithmetic could deteriorate the worst-case running time due to the growing denominators. But if we allow a deterioration of ϵ (for a small constant ϵ) in the approximation ratio, we can overcome this difficulty as follows.
We rescale the *reducedCost*-values using $1/step_r$ units and the d-values using $1/step_d$ units, where $step_r$ and $step_d$ are integers described below. In each recalculation of a d-value (see page 50), we

round up the result of the division in the first assignment and round down in the second assignment; this can be done by appropriately using the integer DIV operation. The only inaccuracy introduced this way is that the reduced costs of some arcs in $\vec{H}$ can be slightly larger than zero. For each arc f in $\vec{H}$, this error can be bounded by splitting it up into the error made by rounding down in the final d-value calculation (at most $r \cdot step_r/step_d^2$) and the sum of the errors made by rounding up in all updates of *reducedCost*(f). The effect of the latter errors can be kept small by choosing $step_r \gg step_d$, because then the error made in each updating of *reducedCost*(f) is relatively small (at most $r/step_r$) when compared to the change in *reducedCost*(f) (at least $1/(r \cdot step_d)$), meaning that the total relative error made this way is small (at most $(r/step_r)/(1/(r \cdot step_d)) = r^2 \cdot step_d/step_r$ over all updates of *reducedCost*(f)). Finally, since there are at most n arcs in $\vec{H}'$, a maximum error of ϵ in the approximation ratio can be guaranteed with polynomially large factors $step_r$ and $step_d$ (e.g. $(4n/\epsilon)^3 r^5$ and $(4n/\epsilon)^2 r^3$, respectively), meaning that all numbers involved in the computation are (up to a constant factor) of the same size as those in the input.

Empirically, this algorithm behaves similarly to (our improved version of) DA$_C$. The lower bounds are again fairly tight, with average gaps from a fraction of a percent to about 2%, depending on the type of instances. The upper bounds, although more stable than those of DA$_C$, are not good; the average gaps are about 8%. The running times (using the same test bed as before) are, depending on the type of instances, sometimes better and sometimes worse than those of DA$_C$; altogether they are still tolerable (several seconds for large and dense graphs).

2.9.4 Further Remarks on Primal-Dual Algorithms

A major point to be improved is the approximation ratio of 2. Assuming that the integrality gap of the directed cut relaxation is well below 2, an obvious desire is to develop algorithms based on it with a better worst-case ratio between the upper and lower bounds (thus proving the assumption). There are two major approaches for devising approximation algorithms based on linear programming relaxations: LP rounding and primal-dual schema. A discussion in [RV99] indicates that no better guarantee can be obtained using a standard LP rounding approach based on this relaxation. The discussion in this section indicates the same for a standard primal-dual approach. Thus, to get a better ratio, extensions of the primal-dual schema will be needed. Two such extensions are used by Rajagopalan and Vazirani [RV99], where a ratio of 3/2 is proven for the special class of quasi-bipartite graphs (no edges between non-terminals).

Another area for further development are the fully polynomial-time approximation schemes by Garg and K¨onemann [GK98] and Garg and Khandekar [GK02] which are applicable to LP_F. Our preliminary tests indicate that these algorithms (at least in their current form) are not practical, because the convergence is too slow.

2.10 Lagrangian Relaxation and Subgradient Optimization

In this section, we only briefly (and mainly for the sake of completeness) outline some approaches based on Lagrangian relaxation of the integer programming formulations; they do not play a major role in this work. We assume that the reader is familiar with the basic ideas of Lagrangian relaxation and subgradient optimization (see for example [AMO93]). In this context, the Lagrangian subproblems often have the integrality property, meaning that solving the Lagrangian multiplier problem is equivalent to solving the LP relaxation of the original (integer) problem, so a subgradient method can be used to approximate the solution value arbitrarily well.

2.10.1 Relaxing the Spanning Tree Formulation

A Lagrangian relaxation LaP_{T_0} of the tree formulation P_{T_0} is described by Beasley [Bea89], relaxing the degree constraints (7.2). 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 (the integrality property is given). The problem here is the value $v(LP_{T_0})$ itself. Theorem 11 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 11.1 in Section 2.4, 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.

2.10.2 Relaxing the Cut Formulation

A natural way for a tight 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 $O(r(m+n\log n))$. The integrality property is given, so this relaxation 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 in this work and is replaced by the approach described in the next section.

2.11 Optimization by Row and Column Generation: Cut and Price

To get an optimal 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 LP_F, 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.

Instead, 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 re-optimizing 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 by Koch and Martin [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 re-optimizations) 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 re-optimizing 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 that 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 re-optimizations and the time needed per re-optimization 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 (11.1) 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 11), a group of them (namely constraints (9.2) of $LP_{\vec{T}}$) can speed up the process if its violated members are added to the current program.

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 Chapter 3), 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 3.3). However, we could use a more sophisticated pricing scheme profitably for our stronger relaxations, as described in the following.

2.11.1 Adaptations for Stronger Relaxations

Here we outline our approaches for actually using stronger relaxations than LP_C algorithmically.

The complete common flow relaxation LP_{FR} is not a good starting point for designing 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 instances 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 utilization cannot be ruled out, turning to the restricted version $LP_{C'}$ (see Section 2.6.5) 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 (15.3), 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 (15.1) as described before.

2. For those $y^{R\setminus\{z_r,z_i\}}$ that have already been inserted, we look for violated Steiner cut constraints (15.2), again with the same procedure as previously described.

3. Now, we are looking for violated constraints (15.4) of the form $y^{R\setminus\{z_r,z_i\}}(\delta^-(z_i)) \leq x(\delta^+(z_i))$ for some z_i. Here, we 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 (15.4), we also include the necessary variables $y^{R\setminus\{z_r,z_i\}}$.

With this procedure the linear program will reach the value $v(LP_{C'})$, which is in some cases significantly higher than $v(LP_{C+FB})$ (see Section 2.13). Some other practical approaches for improving the relaxations are described in Section 2.12.

2.12 Improving Relaxations

The methods described in the previous sections already generate quite tight lower bounds (see Section 2.13), but in some cases, the gaps are still larger than desired. Especially for large and complex problem instances, very small differences in the integrality gap can cause an enormous additional computational effort in the context of an exact algorithm. Therefore, methods for improving the quality of the relaxations are very important. We present here two approaches for improving the lower bounds, using the directed cut relaxation as the starting point.

- In Section 2.12.1, we introduce the "vertex splitting" technique: We identify locations in the network that contribute to the integrality gap and split up the decisive vertices in these locations. Thereby, we transform the problem instance into one that is equivalent with respect to the integral solution, but the solution of the relaxation may improve.

  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.

- In Section 2.12.2, we show how to adopt the "local cuts" approach, introduced by Applegate, Bixby, Chvátal, and Cook [ABCC01] in the context of the TSP: Additional constraints are generated using projection, lifting and optimal solutions of subinstances of the problem. To apply this approach to the Steiner problem, we develop new shrinking operations and separation techniques.

We have applied both methods successfully; in particular, they have played a decisive role for the solution of the largest benchmark instances ever solved (see Section 2.13). Furthermore, we believe that these new techniques are also interesting for other combinatorial optimization problems.

2.12.1 Graph Transformation: Vertex Splitting

In this section, we describe a new technique for effectively improving the lower bound corresponding to the directed cut relaxation by manipulating the underlying network.

We use the property that in an optimal directed Steiner tree, each vertex has in-degree at most 1. Implicitly, we realize a case distinction: If an arc $[v_i, v_j]$ is in an optimal Steiner tree, we know that other arcs in $\delta^-(v_j)$ cannot be in the tree. The only necessary operation to realize this case distinction for the Steiner problem is the splitting of a vertex. A vertex v_j is replaced by several vertices v_j^i, one for each arc $[v_i, v_j]$ entering v_j. Each new vertex v_j^i has only one incoming arc $[v_i, v_j^i]$, and essentially the same outgoing arcs as v_j. In Figure 2.14, the splitting of vertex v_j is depicted. The explanation of the figure also provides some intuition how splitting can be useful. Later we will describe how we identify candidates for splitting.

The splitting operation is described formally by the pseudocode below. We maintain an array *orig* that points for each vertex in the transformed network to the vertex in the original network that it derives from. Initially, $\mathit{orig}[v_j] = v_j$ for all $v_j \in V$. With $P(v_i)$ we denote the longest common suffix of all paths from z_1 to v_i after every path is translated back to the original network. The intuition behind this definition is that if v_i is in an optimal Steiner arborescence, $P(v_i)$ must also be in the arborescence after it is translated into the original network. Note that the path $P(v_i)$ consists of vertices in the original network and may contain cycles; in this case, v_i cannot be part of an optimal arborescence. In Figure 2.14, $P(v_a)$ consists of v_a and $P(v_j^a)$ is the path of length 1 from v_a to v_j. To compute $P(v_i)$, one can reverse all arcs and use breadth-first-search. The main purpose of using $P(v_i)$ is to avoid inserting unnecessary arcs. This can improve the value of and the computation times for the lower bound. It is also necessary for the proof of termination.

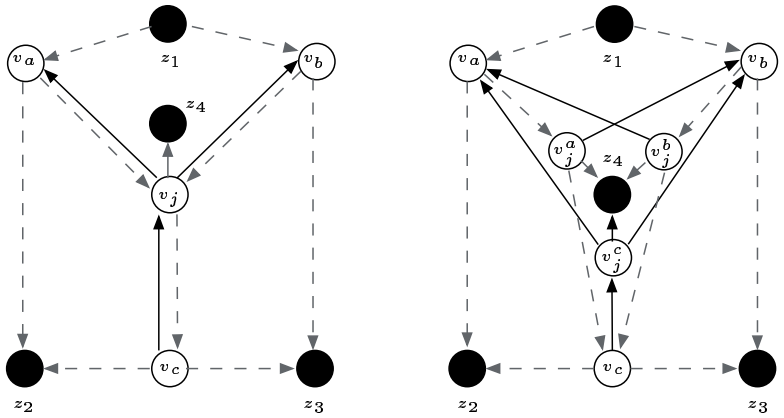


Figure 2.14: Splitting of vertex v_j. The filled circles are terminals, z_1 is the root, all arcs have cost 1. An optimal Steiner arborescence has value 6 in each network. In the left network $v(LP_{C+FB})$ is 5.5 (set the x-values of the dashed arcs to 0.5 and of $[v_j, z_4]$ to 1), but 6 in the right 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.

For the ease of presentation, we assume that the root terminal z_1 has no incoming arcs, and that all other terminals have no outgoing arcs. If this is not the case, we simply add copies of the terminals and connect them with appropriate zero-cost arcs to the old terminals.

SPLIT-VERTEX$(G, R, v_j$,*orig*$)$ (assuming $v_j \notin R$)
1 **forall** $[v_i, v_j] \in \delta^-(v_j)$:
2 **if** $P(v_i)$ *contains a cycle* **or** *orig*$[v_j]$ *in* $P(v_i)$:
3 **continue** *with next arc in* $\delta^-(v_j)$;
4 *insert a new vertex* v_j^i *into* G, *orig*$[v_j^i] :=$ *orig*$[v_j]$;
5 *insert an arc* $[v_i, v_j^i]$ *with cost* $c_{[v_i,v_j]}$ *into* G;
6 **forall** $[v_j, v_k] \in \delta^+(v_j)$:
7 **if** *orig*$[v_k]$ *not in* $P(v_i)$:
8 *insert an arc* $[v_j^i, v_k]$ *with cost* $c_{[v_j,v_k]}$ *into* G;
9 *delete* v_j;
10 *delete all vertices that are not reachable from* z_1;

Correctness

Now we prove that the transformation is valid, i.e., it does not change the value of an optimal Steiner arborescence.

Lemma 27 Any optimal Steiner arborescence with root z_1 in the original network can be transformed into a feasible Steiner arborescence with root z_1 in the transformed network with the same cost and vice versa.

Proof: We consider one splitting operation on vertex $v_j \in V \setminus R$, transforming a network G into G'. Repeating the argumentation extends the result to multiple splits. We use a condition $(\dagger)$ for a tree T

denoting that for every v_k, v_l in T, it holds: *orig*$[v_k] =$ *orig*$[v_l] \Leftrightarrow v_k = v_l$. Note that condition (†) holds for an optimal Steiner arborescence in the original network.

Let T be an optimal Steiner arborescence with root z_1 for G satisfying (†). If $v_j \notin T$, T is part of G' and we are done. If $v_j \in T$, there is exactly one arc $[v_i, v_j] \in T$. When $[v_i, v_j]$ is considered in the splitting, $P(v_i)$ is a subpath of the path from z_1 to v_i in T after it is translated to the original network. Together with (†) follows that neither *orig*$[v_j]$, nor *orig*$[v_k]$ for any $[v_j, v_k] \in T$ is in $P(v_i)$. Therefore, all arcs $[v_j, v_k] \in T$ can be replaced by arcs $[v_j^i, v_k]$ and the arc $[v_i, v_j]$ can be replaced by $[v_i, v_j^i]$. The transformed T is part of G', connects all terminals, has the same cost as T and satisfies condition (†).

Now, let T' be an optimal Steiner arborescence for G'. Obviously, T' can be transformed into a feasible solution T with no higher cost for G. □

Termination

The following lemmas show that iterating the splitting operation will terminate.

Lemma 28 For all non-terminals v_j, $P(v_j)$ is the common suffix of all paths $P(v_i)$ appended by *orig*$[v_j]$ for all $v_i, [v_i, v_j] \in \delta^-(v_j)$.

Proof: As the Line 10 of SPLIT-VERTEX guarantees that there is always a path from z_1 to v_j, the claim follows directly from the definition of $P(v_j)$. □

Lemma 29 For any two non-terminals v_s and $v_t, v_s \neq v_t$, $P(v_s)$ is not a suffix of $P(v_t)$.

Proof: Assume the lemma is not true. We choose two vertices v_s and $v_t, v_s \neq v_t, P(v_s)$ is a suffix of $P(v_t)$ such that the length of $P(v_s)$ is minimal. Obviously, *orig*$[v_s] =$ *orig*$[v_t]$. Thus, v_s and v_t were inserted in some splits. After these splits, v_s and v_t have in-degree 1. Only splitting a vertex v_s' with $[v_s', v_s] \in \delta^-(v_s)$ can increase the in-degree of v_s, but *orig*$[v_s']$ is the same for all $[v_s', v_s] \in \delta^-(v_s)$. Together with Lemma 28 for $P(v_s)$ follows that $P(v_s)$ contains at least two vertices. As it is a suffix of $P(v_t)$, this also holds for $P(v_t)$. For any two vertices v_s', v_t' with $[v_s', v_s] \in \delta^-(v_s)$ and $[v_t', v_t] \in \delta^-(v_t)$ it holds that $v_s' \neq v_t'$, $P(v_s')$ is a suffix of $P(v_t')$ and it is shorter than $P(v_s)$, a contradiction. □

Lemma 30 After splitting a vertex v_j with in-degree greater than 1, for any newly inserted vertex v_j^i it holds that $P(v_j^i)$ is longer than $P(v_j)$ was before the split.

Proof: Assume that there is a newly inserted vertex v_j^a such that $P(v_j^a)$ is not longer than $P(v_j)$. From Lemma 28 for $P(v_j^a)$ and $P(v_j)$ follows that $P(v_j^a) = P(v_a)$ appended by *orig*$[v_j]$ and that $P(v_j)$ is a suffix of $P(v_j^a)$. Together with the assumption follows $P(v_j) = P(v_j^a)$. As v_j had in-degree greater than 1 before the split, we know that there was a vertex $v_b, v_b \neq v_a, [v_b, v_j] \in \delta^-(v_j)$. From Lemma 29 follows that $P(v_a)$ was not a suffix of $P(v_b)$. Thus, the common suffix of $P(v_a)$ and $P(v_b)$ did not contain $P(v_a)$. Using Lemma 28 for $P(v_j)$, it follows that $P(v_j)$ did not contain $P(v_a)$, a contradiction to $P(v_j) = P(v_j^a)$. □

Lemma 31 Repeated splitting of vertices with in-degree greater than 1 will stop with a network in which all non-terminals have in-degree 1. As a consequence, there is exactly one path from z_1 to v_i for all non-terminals v_i.

Proof: As long as there is a non-terminal with in-degree greater than 1, we can split it, which will delete the vertex and possibly replace it by some vertices with in-degree 1. We only have to show that this procedure terminates, as a split may increase the in-degree of other vertices.

If splitting a vertex v_j deletes it without inserting any new vertex, we label v_j as *invalid*.

Now, we examine the changes in the network as an arbitrary vertex v_j with in-degree greater than 1 is split. Let v_m be any non-terminal after the split that was not newly inserted. From the definition of $P(v_m)$ follows that $P(v_m)$ can only change if some vertex or arc is not inserted because of the conditions in Lines 2 and 7 of SPLIT-VERTEX and some paths from z_1 to v_m do not exist any longer. Since there is sill a path from z_1 to v_m left, $P(v_m)$ can only become longer, it may even visit some vertex twice (i.e., $P(v_m)$ contains a cycle). In the latter case, v_m becomes invalid.

From Lemma 27 follows that a transformed optimal tree will always be contained in the current network, thus after at most $|V|$ splits, there will be a split of a valid vertex. If a split is performed on a valid vertex v_j, at least one new vertex v_j^i will be inserted. From Lemma 30 follows that $P(v_j^i)$ is longer than $P(v_j)$ was before the split. But as $P(v_j^i)$ does not contain a cycle (Line 2 of SPLIT-VERTEX), its length is bounded by the number of vertices in the original network. Thus, the procedure terminates. □

Implementation Issues

Of course, for a practical application one does not want to split all vertices, which could blow up the network exponentially. In a cutting-plane algorithm one first adds violated Steiner cut or flow-balance constraints (Section 2.11). If no such constraint can be found, we search for good candidates for the splitting procedure, i.e., vertices where more than one incoming arc and at least one outgoing arc have an x-value greater than zero. After splitting these vertices, the modified network will be used for the computation of new constraints, using the same algorithms as before. To represent this transformation in the linear program, we add new variables for the newly added arcs, and additional constraints that the x-values for all newly added arcs corresponding to an original arc $[v_i, v_j]$ must sum up to $x_{[v_i,v_j]}$. Using this procedure the constraints calculated for the original network can still be used.

2.12.2 Project, Separate, and Lift: Local Cuts

Let $S = (G, R) = (V, E, c, R)$ be an instance of the Steiner problem. Let $\mathcal{ST}(S)$ be the set of all incidence vectors of Steiner trees of S; for ease of notation, we sometimes identify a Steiner tree with its incidence vector. Define $\mathcal{SG}(S) := \mathcal{ST}(S) + \mathbb{R}_+^{|E|}$; we call the elements of $\mathcal{SG}(S)$ the **Steiner graphs** of S. We consider Steiner graphs, since Steiner graphs are invariant under the shrink operation (defined later). Note that the values $x_{(v_i,v_j)}$ are not restricted to be integral or bounded. It is obvious that if the objective function is non-negative, there exists a minimum Steiner graph that is a Steiner tree. Thus all vertices of the polyhedron $\mathrm{conv}(\mathcal{SG}(S))$ are Steiner trees. Furthermore, $\mathrm{conv}(\mathcal{SG}(S))$ is full dimensional if G is connected.

From a high-level view, local cuts can be described as follows. Assume we want to separate x^* from $\mathrm{conv}(\mathcal{SG}(S))$. Using a linear mapping ϕ, we project the given point x^* into a small-dimensional vector $\phi(x^*)$ and solve the separation problem over $\mathrm{conv}(\phi(\mathcal{SG}(S)))$. If we can find a violated inequality $a \cdot \tilde{x} \geq b$ that separates $\phi(x^*)$ from $\mathrm{conv}(\phi(\mathcal{SG}(S)))$, we know that the linear inequality $a \cdot \phi(x) \geq b$ separates x^* from $\mathrm{conv}(\mathcal{SG}(S))$. The method is illustrated in Figure 2.15.

To make this method work, we have to choose ϕ such that

1. there is a good chance that $\phi(x^*) \notin \mathrm{conv}(\phi(\mathcal{SG}(S)))$ if $x^* \notin \mathrm{conv}(\mathcal{SG}(S))$,

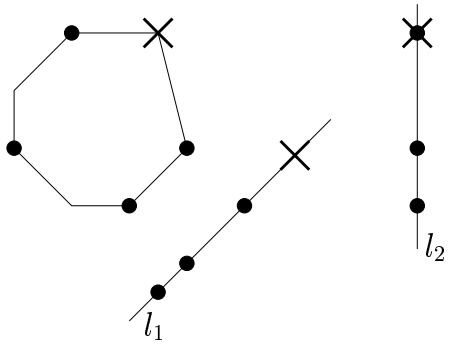


Figure 2.15: The feasible integer solutions are marked as dots, the fractional solution to separate by the cross. If we project the solutions to the line l_1, we can obtain a valid violated inequality and lift it back to the original space. If we project to the line l_2, the fractional solution falls into the convex hull of the integer solutions and no such inequality can be found.

2. we can solve the separation problem over $\mathrm{conv}(\phi(\mathcal{SG}(S)))$ efficiently and

3. the inequalities $a \cdot \phi(x) \geq b$ are strong.

We choose ϕ in such a way that for every solution $x \in \mathcal{SG}(S)$ of our Steiner problem instance S, the projected $\phi(x)$ is a Steiner graph of a small Steiner problem instance S^ϕ, i.e., $\mathrm{conv}(\phi(\mathcal{SG}(S))) = \mathrm{conv}(\mathcal{SG}(S^\phi))$ for an instance S^ϕ of the Steiner problem. Since our Steiner tree program package tends to be very efficient for solving small Steiner problem instances, we can handle the separation problem, as we will see later.

We use iterative shrinking to obtain the linear mappings. First, we review the well-known concept of shrinking. After that, we introduce our separation algorithm for small Steiner graph instances. So far, we always assumed that we are looking at the undirected version of the Steiner problem, since our separation algorithm is much faster for this variant. As explained before, the directed cut relaxation is stronger than the undirected variant. At the end, we will discuss how we can use the directed formulation without solving directed Steiner graph instances in the separation algorithm.

Shrinking

We define our linear mappings as an iterative application of the following simple, well-known mapping, called shrinking. For the Steiner problem, shrinking was introduced by Chopra and Rao [CR94a].

Shrinking means to replace two vertices v_a and v_b by a new vertex $\langle v_a, v_b \rangle$ and replace edges (v_i, v_a) and (v_i, v_b) by an edge $(v_i, \langle v_a, v_b \rangle)$ with value $x^*_{(v_i,v_a)} + x^*_{(v_i,v_b)}$ (We assume $x^*_{(v_i,v_j)} = 0$ if $(v_i, v_j) \notin E$). The new vertex $\langle v_a, v_b \rangle$ is in the set of terminals R if v_a or v_b (or both) are in R. This informally defines the mapping ϕ and the instance S^ϕ. Note that for any incidence vector of a Steiner graph for the original problem, the new vector is the incidence vector of a Steiner graph in the reduced problem. Furthermore, for every Steiner graph $\tilde{x}$ in $\mathcal{SG}(S^\phi)$ there is a Steiner graph $x \in \mathcal{SG}(S)$ such that $\phi(x) = \tilde{x}$. Thus $\mathrm{conv}(\phi(\mathcal{SG}(S))) = \mathrm{conv}(\mathcal{SG}(S^\phi))$.

Note that if we iteratively shrink a set of vertices $W \subset V$ into one vertex $\langle W \rangle$, the obtained linear mapping is independent of the order in which we apply the shrinks. We denote the unique linear mapping that shrinks a subset $W \subset V$ into one vertex by ϕ^W.

We have developed conditions on x^* under which we can prove that $\phi(x^*)$ is not in the convex hull of $\mathcal{SG}(S^\phi)$ if x^* is not in the convex hull of $\mathcal{SG}(S)$.

Lemma 32 Let $x^* \geq 0$.

1. *(edge of value 1):* Let $x^*_{(v_a,v_b)} \geq 1$ and $W = \{v_a, v_b\}$. $x^* \in \text{conv}(\mathcal{SG}(S)) \Leftrightarrow \phi^W(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^W}))$.

2. *(non-terminal of degree 2):* Let v_a be in $V \setminus R$ and the vertices $(v_1, \dots v_k)$ in V be ordered according to their $x^*_{(\cdot,v_a)}$ value (in decreasing order). Furthermore, let $W = \{v_a, v_1\}$. If $x^*_{(v_3,v_a)} = 0$, then $x^* \in \text{conv}(\mathcal{SG}(S)) \Leftrightarrow \phi^W(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^W}))$.

3. *(cut of value 1):* Let W be such that $x^*(\delta(W)) = 1$ and $\emptyset \neq R \cap W \neq R$. Let $\overline{W} = V \setminus W$. $x^* \in \text{conv}(\mathcal{SG}(S)) \Leftrightarrow \phi^W(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^W})) \wedge \phi^{\overline{W}}(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^{\overline{W}}}))$.

4. *(biconnected components):* Let $U, W \subset V$ and $v_a \in V$ be such that $U \cup W = V$, $U \cap W = \{v_a\}$ and $x^*_{(v_k,v_l)} = 0$ for all $v_k \in U \setminus \{v_a\}$ and $v_l \in W \setminus \{v_a\}$. Furthermore, let $\emptyset \neq R \cap W \neq R$. $x^* \in \text{conv}(\mathcal{SG}(S)) \Leftrightarrow \phi^U(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^U})) \wedge \phi^W(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^W}))$.

5. *(triconnected components):* Let $U, W \subset V$ and $v_a, v_b \in V$ be such that $U \cup W = V \setminus \{v_a\}$, $U \cap W = \{v_b\}$ and $x^*_{(v_k,v_l)} = 0$ for all $v_k \in U \setminus \{v_b\}$ and $v_l \in W \setminus \{v_b\}$. Let furthermore $x^*(\delta(v_a)) = 1$ and $v_a, v_b \in R$. $x^* \in \text{conv}(\mathcal{SG}(S)) \Leftrightarrow \phi^U(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^U})) \wedge \phi^W(x^*) \in \text{conv}(\mathcal{SG}(S^{\phi^W}))$.

Proof: We already argued that if $x^* \in \text{conv}(\mathcal{SG}(S))$ then $\phi(x^*) \in \text{conv}(\mathcal{SG}(S^\phi))$ for every linear mapping obtained by iterative shrinking independent of x^*. Thus we only have to show the reverse direction of the claims, i.e., if $\phi(x^*) \in \text{conv}(\mathcal{SG}(S^\phi))$ (for the last three claims, if both projections are in the convex hull) then $x^* \in \text{conv}(\mathcal{SG}(S))$.

It suffices to prove the claims for the case that x^* is rational.

1. We can find a large integer N and, for $1 \leq i \leq N$, (incidence vectors of) Steiner trees $\tilde{t}^i$ in $\mathcal{ST}(S^{\phi^W})$ such that $N\phi^W(x^*) \geq \sum_{1 \leq i \leq N} \tilde{t}^i$.

   The idea is as follows: We will create Steiner trees $t^{i,j}$ out of $\tilde{t}^i$ by including the edge (v_a, v_b) in every tree and for every edge $(v_k, \langle W \rangle)$ in $\tilde{t}^i$ we use either the edge (v_k, v_a) or (v_k, v_b). The number of Steiner trees in which we use a specific edge (v_k, v_a) or (v_k, v_b) is determined by the ratio between $x^*_{(v_k,v_a)}$ and $x^*_{(v_k,v_b)}$.

   Let M be a large integer such that $Mx^*_{(v_k,v_l)}/\phi^W(x^*)_{(v_k,\langle W \rangle)}$ is integral for every $v_k \in V \setminus W$ and $v_l \in W$. We know that $\phi^W(x^*)_{(\langle W \rangle, v_k)} = x^*_{(v_a,v_k)} + x^*_{(v_b,v_k)}$ for all $v_k \in V \setminus W$. For every $\tilde{t}^i$ and $1 \leq j \leq M$ we define $t^{i,j}$ with

   - $t^{i,j}_{(v_a,v_b)} = 1$,
   - $t^{i,j}_{(v_k,v_l)} = \tilde{t}^i_{(v_k,v_l)}$ for $v_k, v_l \in V \setminus \{v_a, v_b\}$,
   - for $v_k \in V \setminus \{v_a, v_b\}$ we make a case distinction:
     If $j \leq Mx^*_{(v_a,v_k)}/\phi^W(x^*)_{(\langle W \rangle, v_k)}$: $t^{i,j}_{(v_a,v_k)} = \tilde{t}^i_{(\langle W \rangle, v_k)}, t^{i,j}_{(v_b,v_k)} = 0$,
     otherwise: $t^{i,j}_{(v_a,v_k)} = 0, t^{i,j}_{(v_b,v_k)} = \tilde{t}^i_{(\langle W \rangle, v_k)}$.

   As $t^{i,j}_{(v_a,v_k)} + t^{i,j}_{(v_b,v_k)} = \tilde{t}^i_{(\langle W \rangle, v_k)}$, it can be verified that $NMx^* \geq \sum_{1 \leq i \leq N} \sum_{1 \leq j \leq M} t^{i,j}$.

   It also follows that if $\tilde{t}^i$ contained a path from a vertex v_k to $v_{\langle W \rangle}$, each $t^{i,j}$ contains a path from v_k to v_a and to v_b. As a consequence, each pair of terminals is connected in $t^{i,j}$.

2. We can find a large integer N and, for $1 \leq i \leq N$, Steiner trees $\tilde{t}^i$ in $\mathcal{ST}(S^{\phi^W})$, such that $N\phi^W(x^*) \geq \sum_{1\leq i\leq N} \tilde{t}^i$.

   The idea is as follows: We only need to consider the case that $\langle W\rangle$ is used in a tree $\tilde{t}^i$. Since there are at most two edges with positive x^*-values adjacent to v_a, we can replace all edges in the tree $\tilde{t}^i$ adjacent to $\langle W\rangle$ (except $(v_2, \langle W\rangle)$) by edges adjacent to v_1. Further, we have to take care of the edge $(v_2, \langle W\rangle)$, if it is in $\tilde{t}^i$. In this case, we create trees $t^{i,j}$ using either the edge (v_2, v_1) or the two edges (v_2, v_a) and (v_a, v_1). Again the number of trees in which we use the two alternatives is given by the ratio of $x^*_{(v_2,v_1)}$ and $x^*_{(v_2,v_a)}$.

   Let M be a large integer such that $Mx^*_{(v_2,v_l)}/\phi^W(x^*)_{(v_2,\langle W\rangle)}$ is integral for $v_l \in \{v_1, v_a\}$. For every $\tilde{t}^i$ and $1 \leq j \leq M$ we define $t^{i,j}$ with

   - $t^{i,j}_{(v_k,v_l)} = \tilde{t}^i_{(v_k,v_l)}$ for every $v_k, v_l \in V \setminus \{v_a, v_1\}$,
   - $t^{i,j}_{(v_k,v_1)} = \tilde{t}^i_{(v_k,\langle W\rangle)}$ for $v_k \in V \setminus \{v_2\}$,
   - $t^{i,j}_{(v_k,v_a)} = 0$ for $v_k \in V \setminus \{v_2\}$,
   - If $j \leq Mx^*_{(v_1,v_2)}/\phi^W(x^*)_{(\langle W\rangle,v_2)}$: $t^{i,j}_{(v_2,v_1)} = \tilde{t}^i_{(v_2,\langle W\rangle)}$, $t^{i,j}_{(v_2,v_a)} = 0$,
     otherwise: $t^{i,j}_{(v_2,v_1)} = 0$, $t^{i,j}_{(v_2,v_a)} = \tilde{t}^i_{(v_2,\langle W\rangle)}$,
   - $t^{i,j}_{(v_a,v_1)} = t^{i,j}_{(v_2,v_a)}$.

   As $x^*_{(v_a,v_1)} \geq x^*_{(v_a,v_2)}$, it can be verified that $NMx^* \geq \sum_{1\leq i\leq N} \sum_{1\leq j\leq M} t^{i,j}$.

   If there is an edge $(v_k, \langle W\rangle)$ in $\tilde{t}^i$, then in each $t^{i,j}$ there is either the edge (v_k, v_1) or (in the case that $k = 2$ and j is large enough) the two edges (v_k, v_a) and (v_1, v_a). Thus $t^{i,j}$ is a Steiner tree.

3. We can find a large integer N and, for $1 \leq i \leq N$, Steiner trees t^i in $\mathcal{ST}(S^{\phi^W})$ and Steiner trees $\bar{t}^i$ in $\mathcal{ST}(S^{\phi^{\overline{W}}})$ such that $N\phi^W(x^*) \geq \sum_{1\leq i\leq N} t^i$ and $N\phi^{\overline{W}}(x^*) \geq \sum_{1\leq i\leq N} \bar{t}^i$.

   Since $x^*(\delta(W)) = 1$, it follows that in each tree t^i there is exactly one edge in $\delta(\langle W\rangle)$ and in each tree $\bar{t}^i$ there is exactly one edge in $\delta(\langle \overline{W}\rangle)$. For each edge $(v_k, \langle W\rangle), v_k \in \overline{W}$, there are $N\phi^W(x^*)_{(v_k,\langle W\rangle)}$ trees t^i containing this edge. We assign each such tree t^i to one edge $(v_k, v_l), v_l \in W$ such that there are $Nx^*_{(v_k,v_l)}$ trees assigned to this edge. This is possible because $\phi^W(x^*)_{(v_k,\langle W\rangle)} = \sum_{v_l\in W} x^*_{(v_k,v_l)}$. We do the same for all trees $\bar{t}^i$. Now, we join the trees $t^a \setminus \{(v_k, \langle W\rangle)\}$ and $\bar{t}^b \setminus \{(v_l, \langle \overline{W}\rangle)\}$ by an edge (v_k, v_l) to a new tree $\hat{t}^i$ if they are assigned to this edge.

   It can be verified that $Nx^* \geq \sum_{1\leq i\leq N} \hat{t}^i$.

   It remains to show that there is a path between each pair of terminals z_1, z_2 in each tree $\hat{t}^i$, originating from t^a and $\bar{t}^b$, both assigned to an edge (v_k, v_l). If $z_1, z_2 \in \overline{W}$, they were connected in t^a and as t^a contained only one edge $(v_k, \langle W\rangle)$, they are still connected in $\hat{t}^i$. The case $z_1, z_2 \in W$ is similar. For $z_1 \in \overline{W}, z_2 \in W$, we can use the path between z_1 and $\langle W\rangle$ in t^a, the edge (v_k, v_l) and the path between $\langle \overline{W}\rangle$ and z_2 in $\bar{t}^b$.

4. We can find a large integer N and, for $1 \leq i \leq N$, Steiner trees t^i in $\mathcal{ST}(S^{\phi^W})$ and Steiner trees s^i in $\mathcal{ST}(S^{\phi^U})$ such that $N\phi^W(x^*) \geq \sum_{1\leq i\leq N} t^i$ and $N\phi^U(x^*) \geq \sum_{1\leq i\leq N} s^i$.

   We join the trees t^i with $\langle W\rangle$ replaced by v_a and s^i with $\langle U\rangle$ replaced by v_a to a new tree $\hat{t}^i$.

It can be verified that $Nx^* \geq \sum_{1 \leq i \leq N} \hat{t}^i$.

Since $\hat{t}^i$ contains the complete Steiner trees t^i and s^i with the respective shrunken vertex replaced by v_a, we know that $\hat{t}^i$ is a Steiner tree.

5. We can find a large integer N and, for $1 \leq i \leq N$, Steiner trees t^i in $\mathcal{ST}(S^{\phi^W})$ and Steiner trees s^i in $\mathcal{ST}(S^{\phi^U})$ such that $N\phi^W(x^*) \geq \sum_{1 \leq i \leq N} t^i$ and $N\phi^U(x^*) \geq \sum_{1 \leq i \leq N} s^i$.

   Note that $\phi^W(x^*)(\delta(v_a)) = 1$ and thus every t^i has exactly one edge adjacent to v_a. Thus there are $i' = N - N\phi^W(x^*)_{(v_a,\langle W \rangle)}$ Steiner trees t^i that do not use the edge $(v_a, \langle W \rangle)$. Let these be the Steiner trees 1 to i'. Analogously there are $i'' = N - N\phi^U(x^*)_{(v_a,\langle U \rangle)}$ Steiner trees s^i that do not use the edge $(v_a, \langle U \rangle)$. Let these be the Steiner trees $i'+1$ to $i'+i''$. First, we replace $\langle W \rangle$ and $\langle U \rangle$ by v_b in all t^i and s^i.

   For $i \leq i'$ we join t^i and the subgraph $s^i \setminus \{(v_a, \langle U \rangle)\}$ to $\hat{t}^i$.

   For $i' < i \leq i''$ we join the subgraph $t^i \setminus \{(v_a, \langle W \rangle)\}$ and s^i to $\hat{t}^i$.

   Finally, for $i > i' + i''$ we join the subgraph $t^i \setminus \{(v_a, \langle W \rangle)\}$, the subgraph $s^i \setminus \{(v_a, \langle U \rangle)\}$, and the edge (v_a, v_b) to $\hat{t}^i$.

   As $\sum_{1 \leq i \leq N} \hat{t}^i_{(v_a,v_b)} = N(\phi^W(x^*)_{(v_a,\langle W \rangle)} + \phi^U(x^*)_{(v_a,\langle U \rangle)} - 1) = N(x^*(\delta(v_a)) + x^*_{(v_a,v_b)} - 1) = Nx^*_{(v_a,v_b)}$, it can be verified that $Nx^* \geq \sum_{1 \leq i \leq N} \hat{t}^i$.

   Finally, we show that all $\hat{t}^i$ are Steiner trees. If $i \leq i'$, $\hat{t}^i$ contains the complete Steiner tree t^i with $\langle W \rangle$ replaced by v_b. Thus all terminals in $U \cup \{v_a\}$ are connected. Since s^i contains $(v_a, \langle U \rangle)$ and two subtrees connecting each terminal in $W \setminus \{v_b\}$ either to v_a or to v_b and since v_a and v_b are connected in t^i, we know that $\hat{t}^i$ is a Steiner tree. A similar argument holds for $i' < i \leq i' + i''$. For $i > i' + i''$, we know that v_a and v_b are connected directly by the edge (v_a, v_b) and every other terminal is connected either to v_a or to v_b. Thus $\hat{t}^i$ is a Steiner tree.

□

Applying these "exact" shrinks does not project the solution of the current linear program into the projected convex hull of all integer solutions, i.e., if the solution of the current linear program has not reached the value of the integer optimum, we can find a valid, violated constraint in the shrunken graphs. Unfortunately, in many cases the graphs are still too large after applying these shrinks and we have to apply some "heuristic" shrinks afterwards.

In the implementation, we use a parameter *max-component-size*, which is initially 15. If the number of vertices in a graph after applying all "exact" shrinks is not higher than *max-component-size*, we start the algorithm FIND-FACET described below, otherwise, we start a breadth-first-search from different starting positions, shrink everything except the first *max-component-size* vertices visited by the BFS, try the "exact" shrinks again and start FIND-FACET. If it turns out that we could not find a valid, violated constraint, we increase *max-component-size*. We also tried other "heuristic" shrinks by relaxing "exact" shrinks, e.g., accepting minimum Steiner cuts with value above 1, or edges that have an x-value close to 1. But we could not come up with a definitive conclusion which shrinks are best, and we believe that there is still room for improvement.

As we will see in the following, our separation algorithm finds a facet of $\mathrm{conv}(\mathcal{SG}(S^\phi))$. As shown in Theorem 4.1 of [CR94a], the lifted inequality is then a facet of $\mathrm{conv}(\mathcal{SG}(S))$.

Separation: Finding Facets

Assume we want to separate x^* from $\mathrm{conv}(\mathcal{SG}(S))$. Note that we actually separate $\phi(x^*)$ from $\mathrm{conv}(\mathcal{SG}(S^\phi))$, but this problem can be solved with the same algorithm.

As we will see, the separation problem can be formulated as a linear program with a row for every Steiner graph. Trying to solve this linear program using cutting planes, we have the problem that the number of Steiner graphs (contrary to the case of Steiner trees) is infinite and optimal Steiner graphs need not exist. Note that the same complication arises when applying local cuts to the traveling salesman problem.

The solution for the separation problem is much simpler and more elegant for the Steiner tree case than for the TSP case. The key is the following Lemma, a slight variation of Lemma 3.1.2 in [CR94a].

Lemma 33 All facets of $\mathrm{conv}(\mathcal{SG}(S))$ different from $x_{(v_a,v_b)} \geq 0$ for an edge $(v_a, v_b) \in E$ can be written in the form $a \cdot x \geq 1$ with $a \geq 0$.

Thus, if $x^* \notin \mathrm{conv}(\mathcal{SG}(S))$, we can find an inequality of the form $a \cdot x \geq 1$, $a \geq 0$, that separates x^* from $\mathrm{conv}(\mathcal{SG}(S))$. Note that if $a \geq 0$, there is a Steiner tree $t \in \mathcal{SG}(S)$ minimizing $a \cdot t$.

Thus an exact separation algorithm can be stated as follows (the name arises from the fact that the algorithm will find a facet of $\mathrm{conv}(\mathcal{SG}(S))$, as we will show later).

FIND-FACET $(G = (V, E), R, x^*)$
1 $T :=$ *incidence vector of a Steiner tree for* G, R;
2 **repeat:**
3 *solve LP*: $\min x^* \cdot \alpha,\ T\alpha \geq 1,\ \alpha \geq 0$; (basic solution)
4 **if** $x^* \cdot \alpha \geq 1$: **return** "$x^* \in \mathrm{conv}(\mathcal{SG}(S))$";
5 *find minimum Steiner tree* t *for* $(V, E, \alpha), R$;
6 **if** $t \cdot \alpha < 1$: *add* t *as a new row to matrix* T;
7 **else**: **return** " $\alpha \cdot x \geq 1$";

The algorithm terminates, since there are only a finite number of Steiner trees in $\mathcal{ST}(S)$ and as soon as the minimum Steiner tree t computed in Line 5 is already in T, we terminate because $\alpha \cdot t \geq 1$ is an inequality of the linear program solved in Line 3.

Lemma 34 If FIND-FACET does not return an inequality, $x^* \in \mathrm{conv}(\mathcal{SG}(S))$.

Proof: Consider the dual of the linear program in Line 3: $\max \sum_i \lambda_i, T^T\lambda \leq x^*$, which has the optimal value $x^* \cdot \alpha \geq 1$. We divide λ by $x^* \cdot \alpha$, with the consequence that $\sum_i \lambda_i = 1$. Now, $T^T\lambda$ is a convex combination of Steiner trees and it still holds $T^T\lambda \leq x^*$. □

Lemma 35 If FIND-FACET returns an inequality $\alpha \cdot x \geq 1$, this inequality is a valid, separating, and facet-defining inequality.

Proof: The value of the last computed minimum Steiner tree t is $t \cdot \alpha \geq 1$. Therefore, if $x \in \mathcal{SG}(S)$, the value can only be greater and it holds $x \cdot \alpha \geq t \cdot \alpha \geq 1$.

As $x^* \cdot \alpha < 1$, the inequality is separating.

From the basic solution of the linear program, we can extract $|E|$ linearly independent rows that are satisfied with equality. For each such row of the form $\alpha \cdot t \geq 1$, we add the tree t to a set S_λ and

for each row $\alpha_e \geq 0$, we add the edge e to a set S_μ. Note that $|S_\lambda| + |S_\mu| = |E|$ and the incidence vectors corresponding to $S_\lambda \cup S_\mu$ are linearly independent.

There is at least one tree t_j in S_λ. For each edge $e \in S_\mu$ we add to S_λ a new Steiner graph t_k that consists of t_j added by the edge e. Since $\alpha_e = 0$ we know that $\alpha \cdot t_k = 1$. Since the incidence vectors corresponding to $S_\lambda \cup S_\mu$ were linearly independent, replacing e with the t_k yields a new set of linearly independent vectors.

Repeating this procedure yields $|E|$ linearly independent $t_i \in S_\lambda$ with $\alpha \cdot t_i = 1$. Thus, $\alpha \cdot x \geq 1$ is a facet. □

As in [ABCC01], we can improve the running time of the algorithm by using the following fact. If we know some valid inequalities $a \cdot x \geq b$ with $a \cdot x^* = b$ then $x^* \in \mathrm{conv}(\mathcal{SG}(S)) \Leftrightarrow x^* \in \mathrm{conv}(\mathcal{SG}(S) \cap \{x \in \mathbb{R}^{|E|} \mid a \cdot x = b\})$. Thus we can temporarily remove all edges (v_i, v_j) with $x^*_{(v_i,v_j)} = 0$, since $x^*_{(v_i,v_j)} \geq 0$ is a valid inequality. Call the resulting instance S'. We use our algorithm to find a facet of $\mathrm{conv}(\mathcal{SG}(S'))$. We can use sequential lifting to obtain a facet of $\mathrm{conv}(\mathcal{SG}(S))$. For details see [ABCC01] and Theorem 4.2 of [CR94a].

Directed versus Undirected Formulations

For computing the lower bounds, we focus on the directed cut formulation, because its relaxation is stronger than the undirected variant. However, in the local cut separation algorithm we want to solve undirected Steiner graph instances, since they can be solved much faster.

The solution is to use another linear mapping that maps arc-values of a bidirected Steiner graph instance $\vec{S} = (V, A, c, R)$ to edge-values of an undirected Steiner graph instance $S = (V, E, c', R)$.

We define S by $E = \{(v_i, v_j) \mid [v_i, v_j] \in A\}$ and $c'_{(v_i,v_j)} = c_{[v_i,v_j]} = c_{[v_j,v_i]}$. For a vector $x \in \mathbb{R}^{|A|}$ we define $\psi(x) \in \mathbb{R}^{|E|}$ by $\psi(x)_{(v_i,v_j)} = x_{[v_i,v_j]} + x_{[v_j,v_i]}$.

Lemma 36 It holds:

- $x^* \in \mathrm{conv}(\mathcal{SG}(\vec{S})) \Rightarrow \psi(x^*) \in \mathrm{conv}(\mathcal{SG}(S))$,
- $\bar{x} \in \mathrm{conv}(\mathcal{SG}(S)) \Rightarrow \exists x^* \in \mathrm{conv}(\mathcal{SG}(\vec{S}))$ with $\psi(x^*) = \bar{x}$,
- if $c \cdot x^*$ is smaller than the cost of an optimal Steiner arborescence, then $\psi(x^*) \notin \mathrm{conv}(\mathcal{SG}(S))$.

Proof: Let z_1 be the root in the directed formulation. It suffices to prove the claims for the case that x^* is rational. We show the claims in turn.

If $x^* \in \mathrm{conv}(\mathcal{SG}(\vec{S}))$, we can find a large integer N and directed Steiner trees $t^i \in \mathcal{ST}(\vec{S})$ such that $Nx^* \geq \sum_{1 \leq i \leq N} t^i$. Clearly $N\psi(x^*) \geq \sum_{1 \leq i \leq N} \psi(t^i)$. Furthermore, $\psi(t^i)$ are Steiner graphs, since each directed path in t^i from the root z_1 to a terminal z_k gives an undirected path between z_1 and z_k in $\psi(t^i)$.

If $\bar{x} \in \mathrm{conv}(\mathcal{SG}(S))$, we can find a large integer N and undirected Steiner trees $t^i \in \mathcal{ST}(S)$ such that $N\bar{x} \geq \sum_{1 \leq i \leq N} t^i$. Let $\tilde{t}^i$ be the directed tree obtained by rooting t^i at z_1. Clearly $\tilde{t}^i$ is a directed Steiner tree and $\psi(\tilde{t}^i) = t^i$. Let $x' = N^{-1} \sum_{1 \leq i \leq N} \tilde{t}^i$. We know that $x' \in \mathrm{conv}(\mathcal{SG}(\vec{S}))$ and $\psi(x') \leq \bar{x}$. Thus there exists $x^* \geq x'$ with $x^* \in \mathrm{conv}(\mathcal{SG}(\vec{S}))$ and $\psi(x^*) = \bar{x}$.

Note that we have defined the objective function c' of the undirected Steiner graph instance such that $c' \cdot \psi(x) = c \cdot x$ for all $x \in \mathbb{R}^{|A|}$. Assume $\psi(x^*) \in \mathrm{conv}(\mathcal{SG}(S))$. We know that there is $x' \in \mathrm{conv}(\mathcal{SG}(\vec{S}))$ with $\psi(x') = \psi(x^*)$. Thus there is a Steiner tree $t \in \mathcal{SG}(\vec{S})$ with $c \cdot t \leq c \cdot x' = c' \cdot \psi(x') = c' \cdot \psi(x^*) = c \cdot x^*$. □

For lifting the undirected edges to directed arcs, one can use the computation of optimal Steiner arborescences. For the actual implementation, we used a faster lifting using a lower bound to the value of an optimal Steiner arborescence, provided by the fast algorithm DUAL-ASCENT (Section 2.9.2). For producing facets for the directed Steiner problem, one could compute optimal Steiner arborescences in the algorithm FIND-FACET.

2.12.3 Concluding Remarks

We presented two theoretically interesting and empirically successful approaches for improving lower bounds for the Steiner tree problem: vertex splitting and local cuts. Vertex splitting is a new technique and improves the lower bounds much faster than the local cut method, but the local cut method has the potential of producing tighter bounds. Vertex splitting, although inspired by a general approach, is not directly transferable to other problems, while local cuts are a more general paradigm. On the other hand, the application needs some effort, e.g., developing proofs for shrinks and implementation using exact arithmetic. A crucial point is the development of heuristic shrinks, where a lot of intuition comes into play and we believe that there is room for improvement. Although the local cut method was originally developed for the traveling salesman problem, its application is much clearer for the Steiner tree problem.

Both methods are particularly successful if there are some local deficiencies in the linear programming solution. On constructed pathological instances the lower bounds are still improved significantly, but the progress is not fast enough to solve such instances efficiently.

Another interesting observation is that the power of the vertex splitting approach can be improved by looking at multiple roots simultaneously. In fact, we do not know any instance where repeated vertex splittings would not bring the lower bound to the integer optimum if multiple roots are used. It remains an open problem to find out if this is always the case.

2.13 Some Experimental Results

In this section, we present summarized results of some different approaches for computing lower bounds on benchmark instances from SteinLib [Ste97] (see Section A for a description of the instance groups). Note that the stand-alone 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 a part of an exact algorithm. Usually, one would first try reduction techniques (even those involving bound calculations) before starting time-consuming lower bound calculations, becauce reductions can not only accelerate the following computations, but also make them more effective. On the other hand, measuring running times and lower bound gaps on reduced instances would make a comparison with other implementations difficult.

In Table 2.2, we report running times and the average gap between the lower bound (rounded up to the next integer) and the known optimum. The main purpose of this table is to provide a rough overview of the power of the different methods. We give results for all groups of instances from SteinLib where all optimal values are known. To ensure comparability, we did not use reductions before starting the lower bound computation. But this meant that the row generating method could not be carried out on all instances in reasonable time; a stroke in the table means that the corresponding computation was aborted for some of the instances of the group because of running time constraints.

An experimental comparison of the directed cut relaxation LP_C with the (stronger) MSTH-based relaxation LP_{FST} is also interesting (a theoretical study was already presented in Section 2.8.2).

instance	DUAL-ASCENT		DUAL-ASCENT, 10 roots		LP_C, row generation	
group	time (s)	gap (%)	time (s)	gap (%)	time (s)	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.

In Table 2.3 we compare the average gaps to integer optimum and computation times for the relaxations LP_{FST} and LP_C on the geometric instances from SteinLib. For the first approach, we used GeoSteiner 3.1 [WWZ01], a software package developed by Warme, Winter and Zachariasen for solving Euclidean and rectilinear Steiner problems and the MSTH problem, by taking as $v(LP_{FST})$ the value of the last linear program before any branching was performed. A comparison of the running times for the whole concatenation phase is given in Section 5.4.1. For these comparisons, we have excluded some TSPFST instances that could not be solved by GeoSteiner in one day (results of our program on such instances can be found in Section 5.4.1). These tests were performed on a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor and 1 GB of main memory (for details see Appendix A).

instance	LP_{FST} (GeoSteiner)		LP_C (our program)	
group	time (s)	gap (%)	time (s)	gap (%)
ES1000FST	99.2	0.0078	80.1	0.0079
TSPFST	129.6	0.009803	28.6	0.009806

Table 2.3: Comparison of LP_{FST} and LP_C.

Studying the data (detailed results on single instances can be found in [PV01d]), one observes:

- Both relaxations yield almost always the same value. Only on a couple of instances, LP_{FST} is tighter than LP_C by a relatively small margin.
- Both relaxations are fairly tight on the considered instances. The average gap to integer optimum is in both cases less than 0.01%.
- The average running times for computing $v(LP_C)$ have been smaller, but this does not say much about which method is faster on a specific instance.

However, note that when solving the instances to optimality, our program would also use our improvements of the relaxation LP_C. For example, on the ES1000FST instances the usage of $LP_{C'}$ (Sections 2.6.5 and 2.11.1) reduces the average gap from 0.008% to 0.001%, which is a substantial

improvement in the context of an exact algorithm. The impact of our improvement techniques for relaxations (Section 2.12) is even more drastic: Using the vertex splitting technique (Section 2.12.1) in combination with LP_C reduces the average gap to 0, meaning that the integer optimum is reached for all instances of this group. The running times for the lower bound computation were increased only by about 10%.

In Table 2.4, we present the impact of both vertex splitting and local cuts on the largest benchmark instances ever solved (some additional experimental results can be found in [APV03b]). Here we have chosen the approach of first applying some reduction methods. Note that without the reductions, the impact of these techniques would be even more impressive, but then these instances could not be handled in reasonable time. In all cases, the lower bound reached the value of the integer optimum (and a tree with the same value was found). Without the lower bound improvement techniques, the exact solution of the instances would take much longer (or was not even possible in case of d15112).

instance	original size		red.	reduced size		LP_{C+FB}		+ vertex splitting		+ local cuts	
	$\|V\|$	$\|R\|$	time	$\|V\|$	$\|R\|$	value	time	value	time	value	time
d15112	51886	15112	5h	22666	7465	1553831.5	20.4h	1553995	21.9h	1553998	21.9h
es10000	27019	10000	988s	4061	1563	716141953.5	251s	716174280	284s	—	
fnl4461	17127	4461	995s	8483	2682	182330.8	5299s	182361	6353s	—	
lin37	38418	172	28h	2529	106	99554.5	1810s	99560	1860s	—	

Table 2.4: Results on large benchmark instances. A dash means that the instance was already solved to optimality without local cuts. For the instance d15112, we used the program package GeoSteiner-3.1 to translate the TSPLIB [Rei91] instance into an instance of the Steiner problem in networks with rectilinear metric. No benchmark instance of this size has been solved before. The SteinLib instances es10000 and fnl4461 were obtained in the same way. Warme et al. solved the es10000 instance using the MSTH approach [WWZ00] and local cuts. They needed months of cpu time. The instance fnl4461 was the largest previously unsolved geometric instance in SteinLib. The SteinLib instance lin37 originates from some VLSI-layout problem, is not geometric, and was not solved by other authors.

Chapter 3

Reductions to Simplify Problem Instances

3.1 Introduction

Informally, reductions are here methods to reduce the size of a given instance without destroying the optimal solution. It has been known for some time that reductions can play an important role as a preprocessing step for the solution of $\mathcal{NP}$-hard problems. In particular, the importance of reductions for the Steiner problem has been widely recognized and a large number of techniques were developed [HRW92]; a milestone was the PhD thesis of Cees Duin [Dui93].

More precisely, a reduction method (also called a **reduction test**) is an algorithm that tries to transform an instance I of the problem into an instance I' of "smaller" size (in this context: with fewer vertices *or* edges), such that an optimal solution for I can be reconstructed (efficiently) from an optimal solution for I'. The motivation is that the reduced instance I' is (hopefully) simplified in some sense. The algorithm tests, using some information from the instance, whether a certain condition (**test condition**) is satisfied. If this is the case, a transformation (called **test action**) is performed. This action can be deleting an edge or removing a non-terminal (possibly after inserting a clique for vertices adjacent to it); such tests are called **exclusion tests**. It can also be the contraction of an edge (into a terminal); such tests are called **inclusion tests**.

We find it helpful to distinguish between two major approaches for designing reduction tests:

alternative-based: Such tests 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 (absence of a proper alternative). For example, the simple exclusion test Long Edges checks for each edge (v_i, v_j) (test object) whether $c(v_i, v_j) > d(v_i, v_j)$ (test condition); if this is the case, the edge (v_i, v_j) can be deleted (test action). The test is valid, because every feasible solution containing the edge (v_i, v_j) could be transformed into one of smaller cost by replacing (v_i, v_j) with a shortest path between v_i and v_j.

bound-based: Such tests 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 perform their action 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. As an example, consider a dual feasible solution of value $lower$ for an LP relaxation of the problem (say LP_C) and the corresponding reduced costs $\tilde{c}_{ij}$ ($\tilde{c}_{ij} = c_{ij} - \sum_{W,[v_i,v_j]\in\delta^-(W)} u_W$ for LP_C, see Section 2.9). An arc $[v_i, v_j]$ can be excluded (the variable x_{ij} can be fixed to zero) if $lower + \tilde{c}_{ij} > upper$, where $upper$ is the cost of a feasible (heuristic) solution for the original problem. Such tests are often called "variable fixing by sensitivity analysis". The validity of the test follows from basic observations on linear programs; in Lemma 48 we will prove a much more general result.

Since each reduction method is specially effective on a certain type of instances, and has less (or even no) effect on some others, it is important to have a large arsenal of different methods at one's disposal. Our contribution in this field has been:

- development of fast reduction methods, both variants of classical methods and new ones,
- introduction of new approaches for designing reduction methods and realizing effective and efficient techniques based on them,
- efficient implementation and integration of these methods into various heuristic and exact algorithms, not only as preprocessing.

In Section 3.2, we present a collection of (fast) alternative-based reductions, including efficient variants of classical tests and some new tests, which can all be realized with worst-case time $O(m + n \log n)$. The achieved efficiency greatly enhances the applicability of reductions, especially for integration into heuristics (Section 4.3).

In Section 3.3, we present a collection of bound-based reductions. On the one hand, we present fast (for example $O(m + n \log n)$ time), but still relatively effective bound-based tests. On the other hand, we develop an extremely powerful reduction scheme by integrating, for the first time, advanced tests into the optimization process of LP relaxations, which is a major component of our exact algorithm (Chapter 5).

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 Section 3.4, we present such an extended reduction test, which generalizes various tests from the literature. We introduce the new approach of combining alternative- and bound-based arguments, 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.

In the solution process, particularly as the result of our other reduction techniques, we frequently encounter graphs of (locally) low connectivity; but the standard methods based on partitioning are not helpful for exploiting this situation. In Section 3.5, we present the new approach of using partitioning to design reduction methods. As we will show, the resulting methods have been quite effective in the context of Steiner problem, and the approach can also be useful for other problems.

In Section 3.6 we outline the integration of different reduction methods into a reduction package. This integration is of great importance, because the most impressive achievements of reductions are mainly due to the interaction of different tests.

Finally, in Section 3.7 we present some experimental results for different combinations of our reduction methods. It turns out that our reduction package is by far both faster and stronger than other existing reduction programs: On the one hand, it needs much less time to achieve each intermediate degree of reduction. On the other hand, the final reduced instances are generally much smaller.

3.1.1 Additional Definitions for Reductions

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

In a (Steiner) tree T, non-terminals of degree at least 3 (in T) and terminals are considered as **key nodes**. A **key path** is a path in T in which (only) the endpoints are key nodes. The unique path in T between two nodes v_i and v_j is called the **fundamental path** between v_i and v_j. A **tree bottleneck** between two nodes v_i and v_j in T is a longest subpath on the fundamental path between v_i and v_j in which only the endpoints may be key nodes.

3.2 Alternative-Based Reductions

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

For the classical tests, we adopt the notation in the book [HRW92] (and not the notations in original works), because the coverage of reductions in that book is more comprehensive and systematic.

3.2.1 Deletion of Edges by Using Distance Measures

To identify unnecessary edges, many tests based on the existence of alternatives have been suggested. For example, the simple test LE (Long Edges) [Bea84] excludes edges (v_i, v_j) with $c(v_i, v_j) > d(v_i, v_j)$. Duin und Volgenant [DV89] introduced the test PT$_m$ (Paths with many Terminals), which generalizes most of these tests.

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.
Since the idea behind this test is central, we repeat the proof.

Lemma 37 The test PT$_m$ is valid.

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 such that the Steiner distance between v_i and v_j along P is s_{ij}. On this path there is an elementary path P' that connects the two components of T. The cost of T will decrease if (v_i, v_j) is replaced by P', a contradiction. □

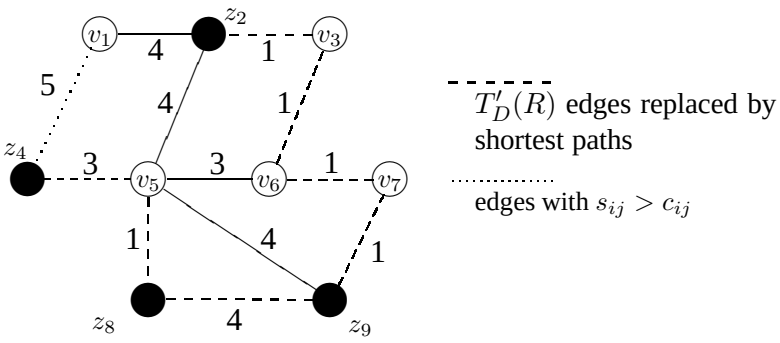


Figure 3.1: Example for the PT$_m$ test.

In Figure 3.1, the edge (v_1, z_4) can be removed with this test, because $c(v_1, z_4) = 5 > 4 = s(v_1, z_4)$. To see that $s(v_1, z_4) = 4$, consider the path $(v_1, z_2, v_3, v_6, v_7, z_9, z_8, v_5, z_4)$.

The test PT$_m$ is one of the most effective classical exclusion tests, but in its original form it is too time consuming for large instances, because it requires the calculation of the s-values for all edges, which needs time $O(n(m + n \log n))$ and space $\Theta(n^2)$ [Dui00]. Here we consider a fast realization of this test, which also uses alternative information. The modifications follow the same principal ideas as used by Duin [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 $T'_D(R)$ (Section 1.2), which can be constructed in time O$(m + n \log n)$. Each such bottleneck can be trivially computed in time $O(r)$, leading to a total time $O(qr)$ for q queries ($q \in O(\min\{m, r^2\})$).

Observing that one actually has a static-tree variant of the bottleneck problem, one can use a strategy based on depth-first search (as described in [VJ83]) to achieve time $\Theta(r^2)$ for all queries. One can go further and solve the problem as an off-line variant for all q queries 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 all queries: 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 queries.

For arbitrary 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 is 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 $\check{s}(v_i, v_j) := \max\{d(v_i, base(v_i)), d(v_j, base(v_j))\}$ (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 $\check{s}(v_i, v_j) = \hat{s}(v_i, v_j)$. If v_i and v_j belong to different Voronoi regions and $c(v_i, v_j) < \check{s}(v_i, v_j)$, 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 observation leads to a simple, very fast test, which is sometimes very powerful:

Lemma 38 Let $\hat{S}$ be the length of a 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 H spanning all terminals with $c(H) < c(T)$, 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 these variants consider only paths with at least one terminal, they miss some of the edges the simple test LE 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 helpful. 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**.

A test like PT$_m$ can actually be extended to the case of equality with restricted bottleneck Steiner distances: 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)). 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. For example, consider the network in Figure 3.1 without the edge (z_8, z_9). Both edges (z_2, v_5) and (v_5, z_9) satisfy the test condition with equality, but after one of them is removed, the condition is not satisfied for the other anymore. Because of such difficulties, in the literature it was assumed that the test PT$_m$ cannot be performed in the case of equality without recalculation of the necessary information. We observed that the few problematic cases can be efficiently identified, so that in all other cases the test actions can be performed even in case of equality (without recalculation). The details are rather technical, with a long list of case differentiations (see [PV97]). But it must be mentioned that this observation 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 and can be reactivated with a measure like this.

3.2.2 Substitution of Non-Terminals

To identify non-terminals that are not key nodes in at least one Steiner minimal tree, the following test NTD$_k$ (Non-Terminals of Degree k) was introduced by Duin and Volgenant [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 $(\Delta, \Delta \times \Delta, s)$.

Later (in Lemma 49), we will prove a much more general test condition, which also shows the validity of the above test.

If the test 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 3.2.1. 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 delicate 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.

3.2.3 Contraction of Edges

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_k, v_l) with $v_k \in N(z_i), v_l \in N(z_j), z_j \neq z_i$ is visited, it is checked whether v_k 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 $\min\{distance(z_i), d(z_i, v_k) + c(v_k, v_l) + d(v_l, z_j)\}$. Now we have:

Lemma 39 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 40 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 between v_1 and z_i and between v'_1 and z_j, we get a subgraph H that includes (v_1, v'_1) and spans all terminals with $c(H) \leq c(T)$, a contradiction. □
This test can also be performed for all terminals in total time $O(m + n \log n)$. An application of these tests is shown in Figure 3.2.

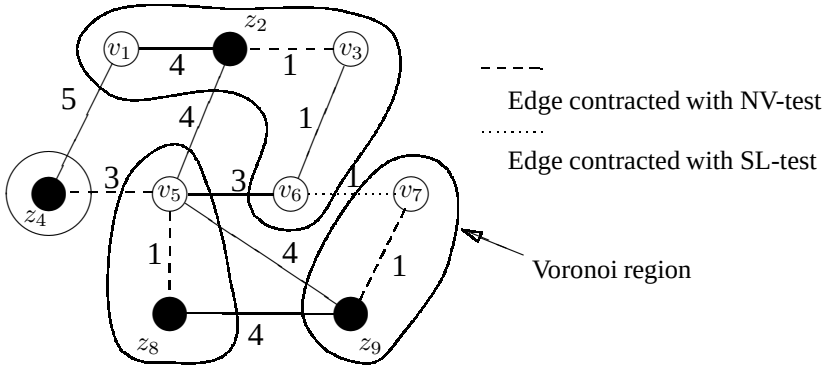


Figure 3.2: Example for the application of the NV and SL tests.

The classical test SE (Short Edges) [DV89, HRW92] is a more powerful inclusion test. We observed for the first time that even this test can be implemented with time $O(m + n \log n)$, using the

same methods and data structures as in section 3.2.1. But although this test is more effective than NV and SL in a single application, the difference diminishes when the reduction tests are iterated. Therefore, we did not include this test in our packet of fast tests.

3.2.4 Exclusion of Paths

We designed another alternative-based reduction test that is more general than the previous tests in two ways: It 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 once. The other more general aspect is a consequence of the first: Searching for alternatives 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 41 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 obtain a subgraph H that does not contain (v_0, v_1) and spans all terminals with $c(H) \leq c(T)$, 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 $v_0^i, i \in \{1, \ldots, degree(v_0) - 1\}$, 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. We call this fast version **PS** (for Path Substitution). 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.

3.3 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 section), the computation of (sharp) lower bounds is a common part of the standard algorithms for the exact solution of such a problem. The information gained during such computations can be used to reduce the instance further. Besides, by using fast heuristics for generating bounds small worst-case running times can be guaranteed even for this kind of tests.

3.3.1 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 $radius(z)$ 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 3.2.1. The following lemma can be used to eliminate a non-terminal.

Lemma 42 Let T be a Steiner minimal tree and assume that v_i is a Steiner node in T. Then $d(v_i, z_{i,1}) + d(v_i, z_{i,2}) + \sum_{t=1}^{r-2} radius(z_t)$ 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 $z_l = base(v_i)$) 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 43 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 42. □

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

Lemma 44 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 42. □

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

Proof: Observe that the graph $\hat{G}$ is identical to that defined in Section 2.9.3, page 49, on primal-dual approaches, so the same assertions about the bounds follow. [1] □

This lemma can be extended to test conditions; 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 $\hat{G}$ 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 ratio of terminals to all vertices is large. Additionally, this test was the basis for the development of the strong PRUNE-heuristics, which are presented in Section 4.3.

3.3.2 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:

[1] In [PV01c] we gave a detailed, more tedious proof without using the notion of primal-dual algorithms.

Lemma 46 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 cost of any (directed) Steiner tree in $G' = (V, A, c')$ with $c' := c - \tilde{c}$. For any $\tilde{x}$ representing a feasible Steiner tree for G of cost $c \cdot \tilde{x}$, it holds that $lower' + \tilde{c} \cdot \tilde{x} \leq c \cdot \tilde{x}$.

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

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 cost of an optimum Steiner tree with additional constraints, for example, that the tree contains a certain non-terminal. The resulting tests follow the same line as the tests IRA and IRAe, which were introduced by Duin [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 3.2.1. 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 47 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 arcs $[v_a, v_b] \in A$.

Proof: Let $v_1, v_2, \ldots, v_n$ be the vertices in V sorted by their distance 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 $l > k$, and 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 arc $[v_a, v_b]$ we have:

$$\begin{aligned}
\sum_{W,[v_a,v_b]\in\delta^-(W)} u_W &= \sum_{b\leq k<a} u_{W_k} \\
&= \sum_{b\leq k<a} (d'(v_{k+1}) - d'(v_k)) \\
&= \max\{0, d'(v_a) - d'(v_b)\} \\
&= \max\{0, \min\{d(v_a, z_t), d(z_1, z_t)\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} \\
&\leq \max\{0, \min\{c_{ab} + d(v_b, z_t), d(z_1, z_t) + c_{ab}\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} \\
&= c_{ab}.
\end{aligned}$$

□

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 any involved arc $[v_a, v_b]$ 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 arcs 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 ratio of terminals to all vertices is not very large.

3.3.3 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 goal, we use the row generation method described in Section 2.11.

Every iteration of the row generation method provides a dual feasible solution for the underlying relaxation (for example LP_C or $LP_{C'}$) and appropriate reduced costs. Using this information, the same reduction techniques as described in Section 3.3.2 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 arcs 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 also be exploited for even stronger reductions in combination with the extended reduction techniques that will be described in the next section.

Note that using a lower bound and reduced costs produced by an LP solver like CPLEX in the context of reduction techniques is a delicate matter. 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 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 values, 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 large factor for the scaling.

The dual side is more interesting. The basic idea is to round the dual values to integers 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 major problem is that the dual values may not be feasible any more. Here we can use a nice extension of Lemma 46.

Lemma 48 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 48.1 If $\bar{\mu} \geq A^T\bar{\lambda} - c$, then 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 values in the context of bound-based reductions, we do the following:

1. We round the dual values to integers $\bar{\lambda}$. We choose the scaling factor such that the current difference between upper and lower bound corresponds to an integer $rest$ that uses nearly the full bit length of the computer's integers. We can ensure that all numbers computed in the bound-based reductions are not greater than $2\ rest+1$. On the one hand, this prevents 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 values associated to constraints containing x_{ij} a little so that the violation gets smaller. This is done in a heuristical way.

4. Using exact arithmetic, we compute $\bar{\mu} := \max\{0, A^T\bar{\lambda} - c\}$. From Corollary 48.1 it follows that $lower := b \cdot \bar{\lambda} - 1 \cdot \bar{\mu}$ is an exact lower bound for the cost of an optimum Steiner tree.

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

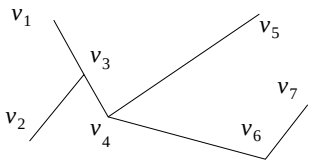
6. Now, we can perform bound-based reductions as described in the previous subsection with $lower$ and $\tilde{c}$, using that $lower + \tilde{c} \cdot \tilde{x}$ is a lower bound for the cost of a Steiner tree represented by $\tilde{x}$.

3.4 Extension and Combination of Reduction Techniques

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 this section, we begin with a generic framework for extended reduction tests, which also generalizes various tests from the literature. The generic algorithm is then substantiated by presenting the applied test conditions and criteria for guiding and truncation of expansion. 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.

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 3.3). This means that for every leaf 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 3.3: Depiction of some central notions for extended reductions.

3.4.1 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 [Win95, Dui00, UdAR02]. The following function EXTENDED-TEST describes in pseudocode a general framework for many of these approaches. The argument of EXTENDED-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-OUT(T, L) contains the specific test conditions (see Section 3.4.2): RULE-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(G, R, 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 non-empty extension ⊆ {(v_i, v_j) :
                        not RULE-OUT(T ∪ {(v_i, v_j)}, L(T) ∪ {v_j})} :
9                   if not EXTENDED-TEST(G, R, 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-OUT is correct, the correctness of EXTENDED-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-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 [UdAR02] 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 [UdAR02].

- In Section 3.2.4, 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-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 non-trivial. 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-OUT), using the results of Duin [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.

3.4.2 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 49 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 with 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:

Lemma 50 If for any $v_i, v_j \in T$, the length of a tree bottleneck between v_i and v_j in T is larger than the s_{ij} in G, then there is an optimum 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 no larger cost. □

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

Lemma 51 Let $\{l_1, l_2, \ldots, l_k\} = L(T)$ be the leaves of T. Then $lower_{constrained} := lower' + min_i\{\tilde{d}(z_1, l_i) + \tilde{c}(\vec{T}_i) + \sum_{j \neq i} min_{z_p \in R^{z_1}} \tilde{d}(l_j, z_p)\}$ 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 46. □

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

Lemma 52 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-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.

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

3.4.3 Criteria for Expansion and Truncation

The basic truncation criterion is the number of backtracking steps, where there is an obvious trade-off 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 deg of a leaf is large, considering all $2^{(deg-1)} - 1$ 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) by Duin [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.

3.4.4 Implementation Issues

Precomputing (Steiner) Distances: A crucial issue for the implementation of the test is the calculation of bottleneck Steiner distances as described in Section 3.1.1. An exact calculation of all s_{ij} would make the test impractical even for medium-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 $O(m + n \log n)$, as described in Section 3.2.1. As we described there, the required time for each query can be made small, or even constant if all necessary queries are known in advance. Although the resulting approximate values $\hat{s}_{ij}$ produce quite satisfactory results for the original test (PT_m in Section 3.2.1), 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 $\tilde{s}$-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. For more details, see [PV01b].

Tree Bottlenecks: The tree bottleneck test of Lemma 50 can be very helpful, because every distance between tree nodes 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. 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 51 is the DUAL-ASCENT algorithm described in Section 3.3.2. 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 3.3.3. 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 3.2.2) 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 52. 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 by Duin [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.

3.4.5 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 3.4.4.

1. For a fast preprocessing we use the linear time expansion test that eliminates paths, as described in Section 3.2.4.

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.

Some experimental results of a selection of these methods can be found in Section 3.7.

3.5 Partitioning as a Reduction Technique

Partitioning is one of the basic ideas for designing efficient algorithms, but for $\mathcal{NP}$-hard problems like the Steiner problem, straightforward application of the classical paradigms for exploiting this idea rarely leads to empirically successful algorithms. In this section, we present the new approach of using partitioning to design reduction methods. As we will show, this method has been quite effective in the context of Steiner problem, and it can 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) in their advanced stages 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 3.5.4.

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.

The approach chosen here for partitioning is based on certain separating sets (vertex separators), these are sets of vertices whose removal makes the graph disconnected (remember that we assumed our graphs are connected). We consider here (small) separating sets that contain only terminals (**terminal separators**), although the basic ideas can be extended to general vertex separators. This choice allows us to keep the dependence between the resulting subinstances manageable.

3.5.1 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 (particularly those described in Section 3.4) that, when successful, tend to transform instances without useful terminal separators into instances with them.

Figure 3.4 shows a VLSI-instance from the library SteinLib [Ste97]. 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 3.5 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 3.6. Note that the visualization used is not geometric; edges that 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.

Geometric Steiner Problems

As described in Section 2.8, the bottleneck of the FST approach to geometric Steiner problems is usually the second phase. As previously described, 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. Such graphs usually have many useful terminal separators with corresponding small components already at the beginning of the second phase, this is

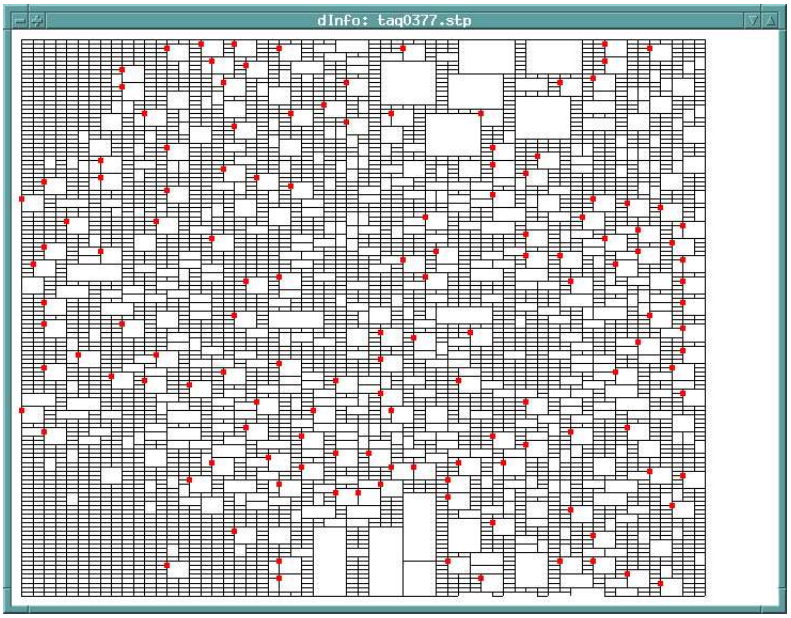


Figure 3.4: VLSI-instance taq0377 ($|V| = 6836,\ |E| = 11715,\ |R| = 136$).

outstandingly given in the Euclidean case due to the nature of geometric techniques used in the first phase [WWZ00]. This property 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.

3.5.2 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

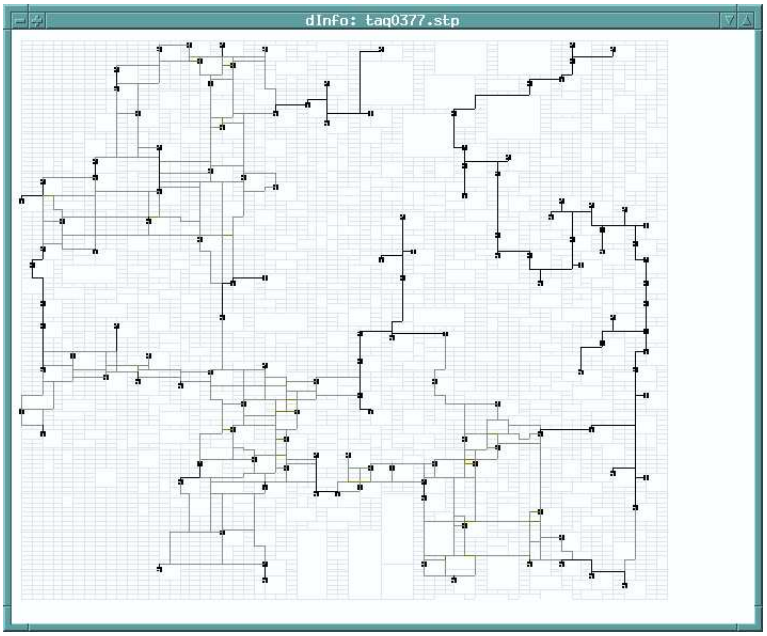


Figure 3.5: taq0377, reduced ($|V| = 193,\ |E| = 312,\ |R| = 67$).

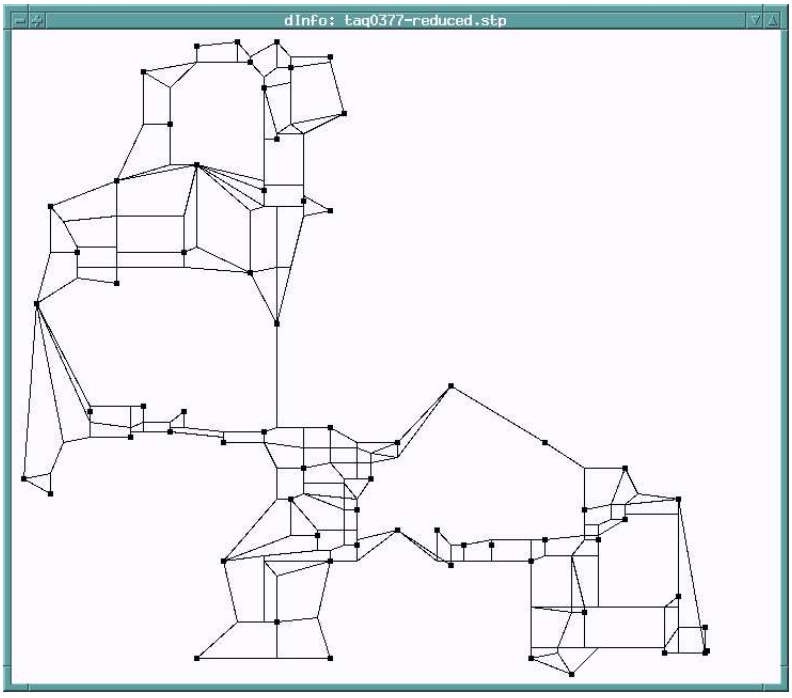


Figure 3.6: taq0377, reduced, redrawn ($|V| = 193,\ |E| = 312,\ |R| = 67$).

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.

3.5.3 Reduction by Case Differentiation

In this section, we describe a reduction method that exploits small terminal separators $S \subset R$ to reduce a given instance.

warm-up ($|S| = 1$): The case $|S| = 1$ corresponds to articulation points (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 3.16 on page 103: 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 also Lemma 59 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 3.7, 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 G_2, 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 the 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 3.8, where only one edge from G_2 is left undecided (the other edges can be either contracted (black) or deleted (light grey)). In Figure 3.9, the reduced instance is redrawn; as one sees, the processed component has almost disappeared.

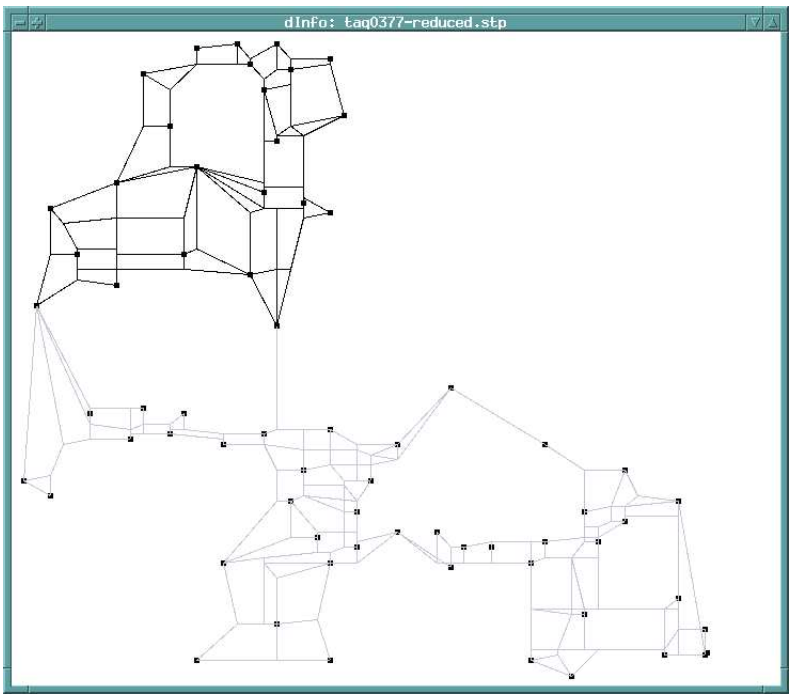


Figure 3.7: $|S| = 2$.

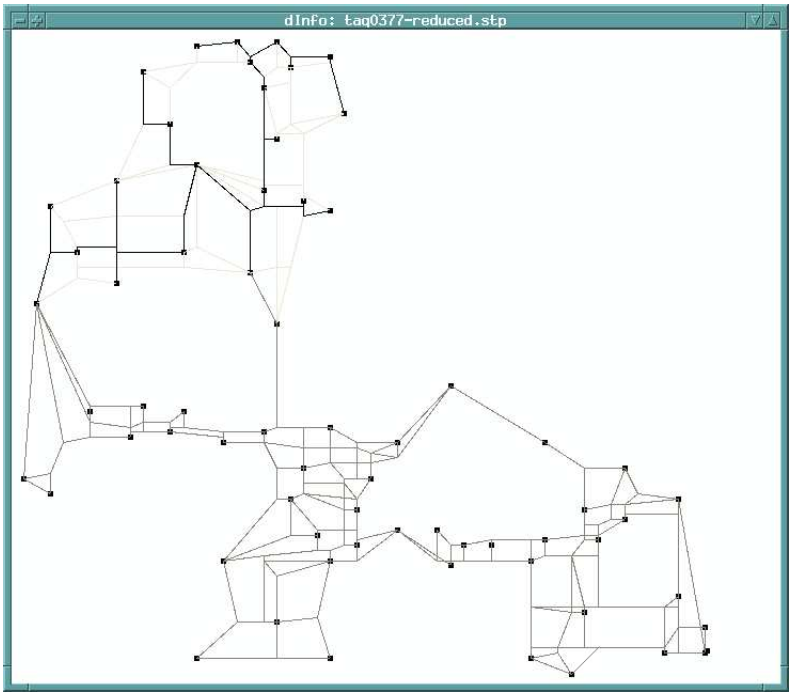


Figure 3.8: ($|V| = 133,\ |E| = 214,\ |R| = 45$).

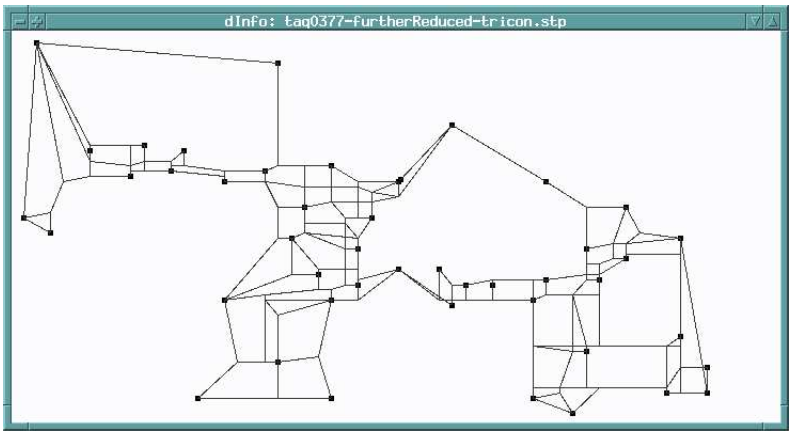


Figure 3.9: ($|V| = 133,\ |E| = 214,\ |R| = 45$).

general case ($|S| = k$): As described in Section 3.5.2, 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 $\left\{ {k \atop i} \right\}$. So there are $\sum_{i=1}^{k} \left\{ {k \atop i} \right\} = B(k)$ cases, where $B(k)$ denotes the k-th Bell number. 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 53 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 that for any subset $S \subseteq R$, all b_{ij}, s_{ij} with $v_i, v_j \in S$ can be computed in time $O(|E| + |V| \log |V| + |S|^2)$ for a graph (V, E) (Section 3.2.1 and [Dui93]).

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.

3.5.4 Reduction by 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 3.4 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 3.3.2: Any linear relaxation can provide a dual feasible solution of value $lower$ and reduced costs $\tilde{c}$. We can use a fast method to compute a constrained lower bound with respect to $\tilde{c}$. From Lemma 46 follows that the sum of the two bounds 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 from Section 2.2.1. 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 3.10). But in the optimal solution of the LP relaxation LP_C variables can have fractional values. In Figure 3.11, 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 3.12). 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: $upper(G'_2) < lower_{constrained}(G''_2)$.

Lemma 54 The test condition is valid, i.e., no Steiner minimal tree in G satisfies the constraint if $upper(G'_2) < lower_{constrained}(G''_2)$.

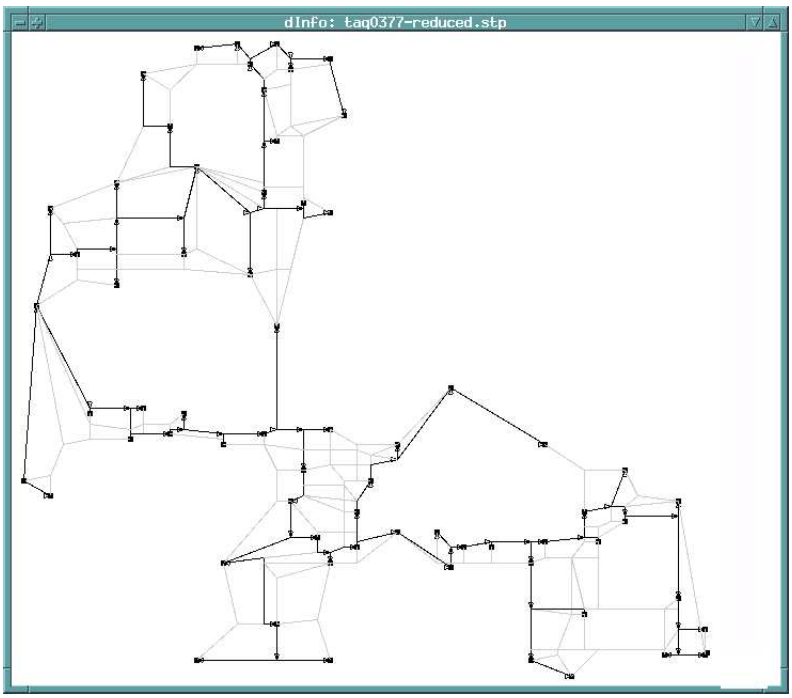


Figure 3.10: $v(P_C) = 6393$.

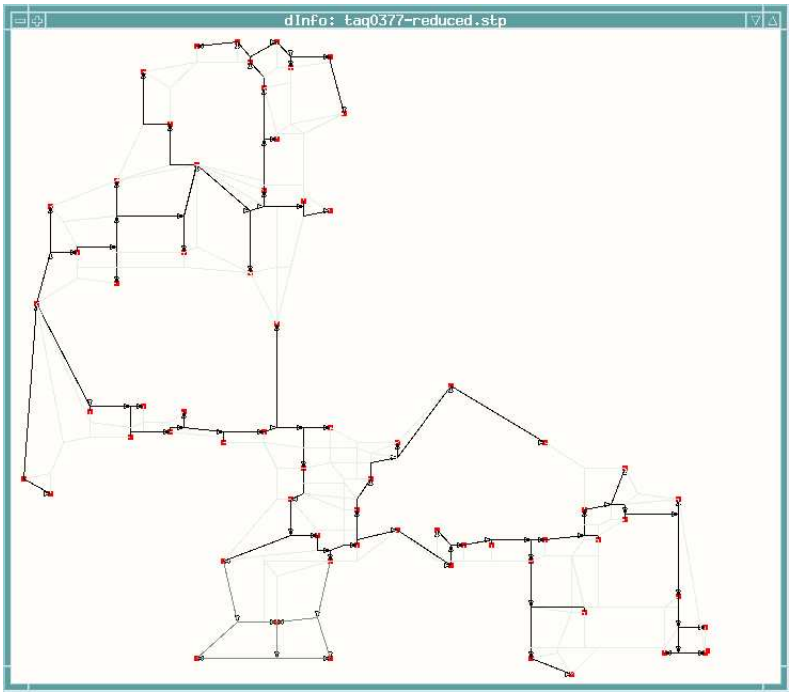


Figure 3.11: $v(LP_C) = 6392.5$.

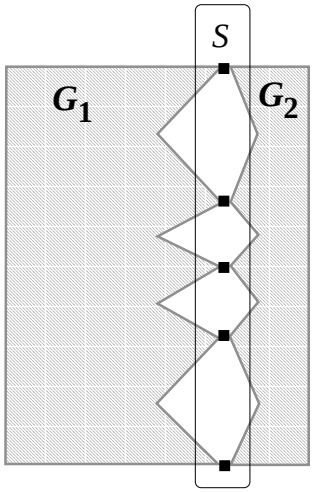


Figure 3.12: A terminal separator S and corresponding subgraphs G_1 and G_2.

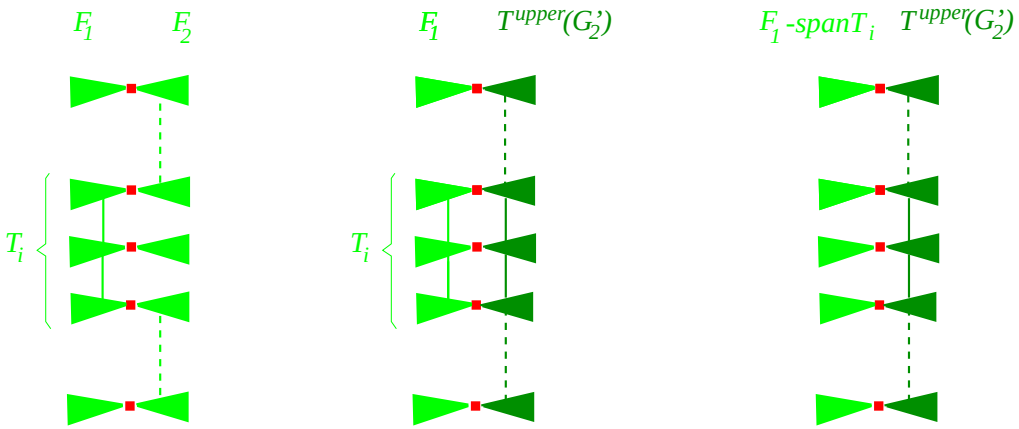


Figure 3.13: Construction of $T^{upper}(G')$ from $T^{opt}_{con}(G)$.

Proof:

Consider $T^{opt}_{con}(G)$, an (unknown) optimum Steiner tree of cost $opt_{con}(G)$ 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 3.13, 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 3.13, 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 3.13, 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 value does not change by inserting any edges (v_i, v_j) of length s_{ij} into G (see Lemma 37), so the optimum solution values in G' and G are the same.

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 optimum solution value. □

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 3.14, where a separator of size 4 and the corresponding component G_2 are shown (black edges). Figure 3.15 shows the result of application of the reduction method presented (black edges contracted, grey edges remaining); in Figure 3.16 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 3.17, 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.

3.6 Integration and Implementation of Tests

The most impressive achievements of reductions are mainly due to the interaction of different tests. Often the actions of some tests prepare the ground for the success of some others and vice versa, so looping over a sequence of tests can be quite effective.

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] for classical tests is a suitable one (although not necessarily the only one, as long as a fast version of PT_m is performed first).

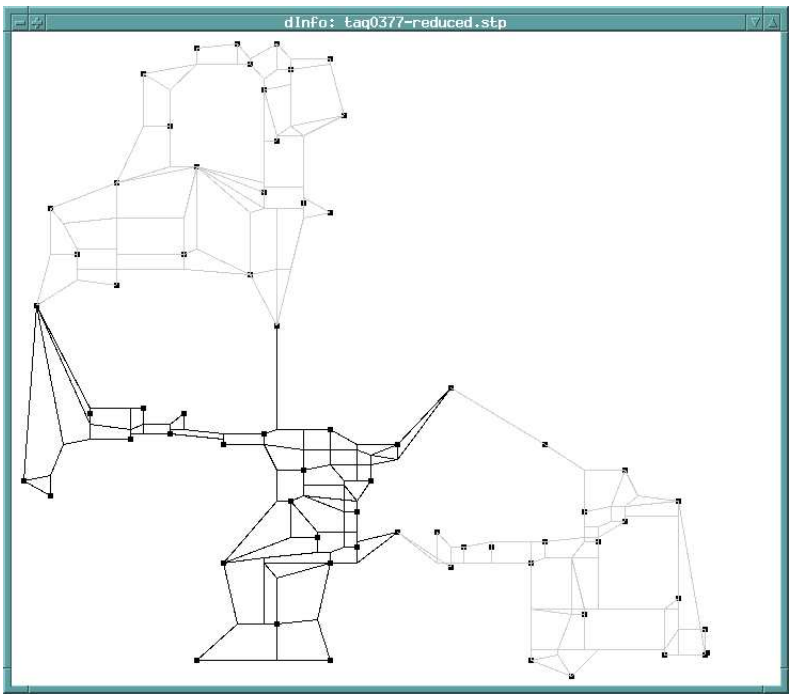


Figure 3.14: $|S| = 4$.

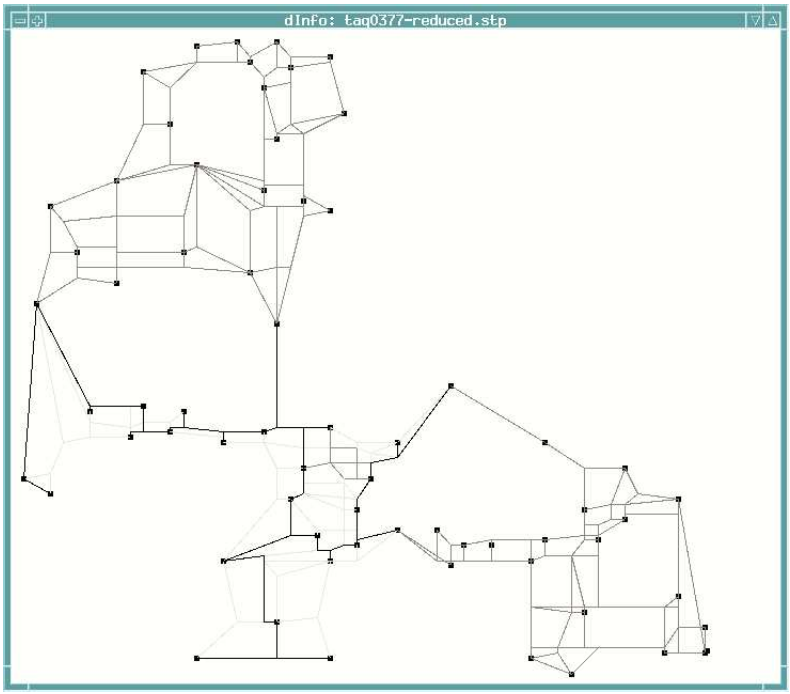


Figure 3.15: ($|V| = 121,\ |E| = 200,\ |R| = 41$).

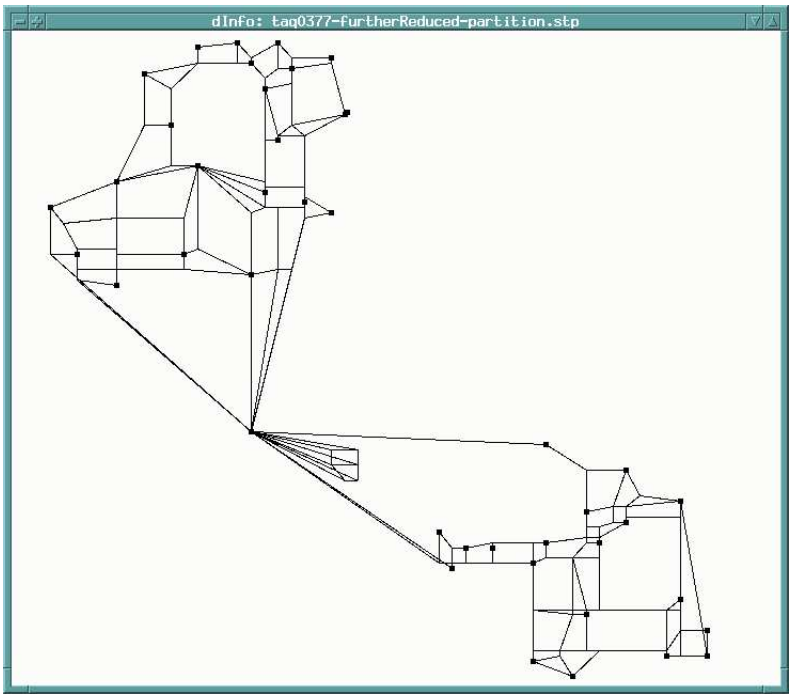


Figure 3.16: ($|V| = 121,\ |E| = 200,\ |R| = 41$).

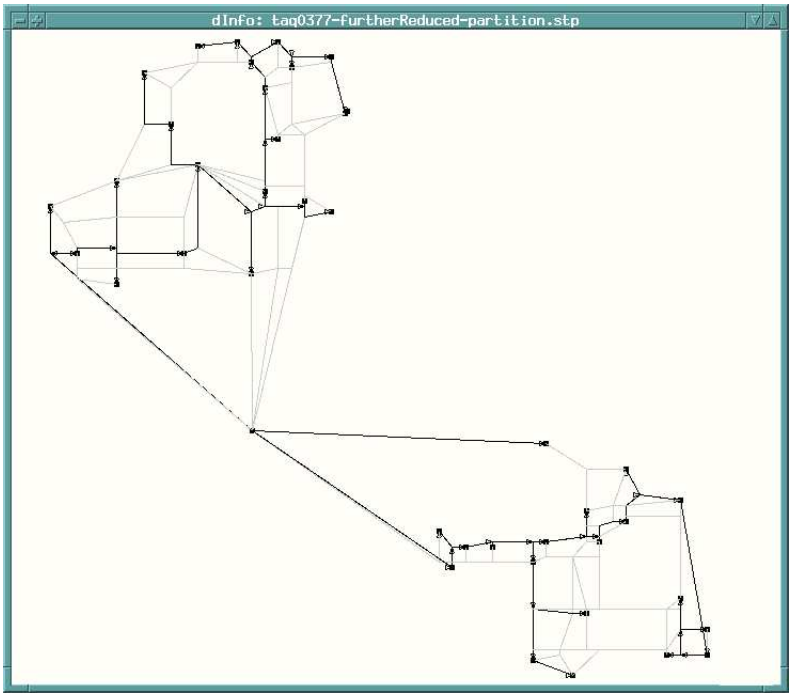


Figure 3.17: $v(LP_C) = 6393$.

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 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 all iterations as for the first one ($O(m + n \log n)$, 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.

3.7 Some Experimental Results

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

Very fast tests: 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, some extended reduction tests are also used. Note that using extended reductions, there is an obvious trade-off between effectiveness and running time, which can be configured by increasing or decreasing the parameter for the maximum backtracking depth. A more detailed study on extended reduction techniques is given in [PV01b].

+ Partitioning-based reductions : Here, we additionally use the partitioning-based reductions, presented in Section 3.5. Note that they also require exact solution of small subinstances. More results on the impact of these techniques can be found in [PV01e].

The results are summarized in Table 3.1. Note that the columns of this table represent independent runs on the original instances.

Although reductions are used in most articles reporting experimental results for the Steiner problem, there are only a few other works where the reported reduction results reach a quality that come any close to ours. For classical reductions, some of the best results are reported in the PhD thesis of Cees Duin [Dui93], who had developed with Ton Volgenant several of the most important classical tests. He modified these tests so that they are applicable at least to medium-size 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 eliminate more than 98% of the edges, while his eliminate 76%. (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 eliminated 62% of the edges.

For VLSI instances and using some extended reduction techniques, the best other results are presented by Uchoa, Poggi de Aragão, and Ribeiro [UdAR02]. With their reduction techniques only 6.2% of the edges remain, taking 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 group	fast tests		+ dual ascent		+ extended tests		+ partitioning	
	time in sec.	remaining edges in %	time in sec.	remaining edges in %	time in sec.	remaining edges in %	time in sec.	remaining 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.

It should be stressed that although the results in Table 3.1 are impressive in many cases, they still do not reveal the whole power of reductions. Some of the strongest reductions are achieved when using the row-generating method (RG test from Section 3.3.3), for example in the context of an exact algorithm. Even if (despite using interleaved reductions and improvement methods like those in Section 2.12) the relaxation does not reach the integer optimum at the end, often the reductions performed in the meantime make a reactivation of the reduction process possible. Repeating this procedure often postpones or eliminates the need for branching. Some results including the RG test are presented in the context of exact algorithms, in Section 5.4.

Chapter 4

Heuristics and Upper Bounds

4.1 Introduction

Since we cannot expect to solve all instances of an $\mathcal{NP}$-hard problem like the Steiner problem in times that are small (polynomial) with respect to the size of instance, methods for computing "good", but not necessarily optimal solutions (heuristics) are well motivated on their own. But the value of such a solution provides also an upper bound for the value of any optimal solution, which can be used, for example, in bound-based reductions (Chapter 3) or even in exact algorithms (Chapter 5).

For the Steiner problem, the number of papers on heuristics is enormous; there are many tailored heuristics, and also every popular meta-heuristic has been tried (often in many variants). A comprehensive overview of articles published before 1992 can be found in [HRW92]. But the best results have been published after that (or even have not been published in a journal yet). 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 this chapter, we describe some of the methods we used in our attempt to achieve both goals simultaneously.

In Section 4.2, we describe some efficient realizations of shortest paths heuristics, which are among the most successful classical heuristics for the Steiner problem. Our variants are not designed to be used as stand-alone heuristics, but as components of other algorithms, especially in combination with reductions. We present also experimental results, partly to make it possible to compare some of the best-known classical heuristics for the Steiner problem with our new, strong heuristics which will be presented in the following sections.

In Section 4.3, we use the idea of combining heuristic and exact reductions to design some of the most powerful heuristics for the problem, which we call PRUNE heuristics.

In Section 4.4, the approach of PRUNE heuristics is further elaborated by using the information gained from relaxations.

In Section 4.5, we outline our methods for exploiting a collection of available Steiner trees to compute a possibly better one.

Finally, in Section 4.6, we present some experimental results, comparing our PRUNE heuristics to the strongest other methods from the literature.

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 construct a Steiner tree in a way similar to that of Prim's algorithm for building minimum spanning trees: Start with a subtree that initially consists of one vertex. In each step, connect the subtree by a shortest path to a closest terminal not in the current tree. Repeat this until all terminals are connected. A final pruning phase computes a minimum spanning tree for the vertices used in the heuristic tree and removes non-terminals with degree one from the resulting tree until all its leaves are terminals. The worst-case running time of this heuristic is $O(r(m + n \log n))$, and it has a performance ratio of 2. To improve the quality of the heuristic, one can repeat it with different start vertices (repetitive SPH).

The algorithm of Dijkstra for the computation of shortest paths uses tentative distances τ_i for every vertex $v_i \in V$. A tentative distance τ_i represents an upper bound for the distance between the current subtree and v_i. In [PW02] Poggi de Aragão and Werneck used a simple, but very helpful observation: The tentative distances need not be reset every time a terminal is connected to the current subtree, since the old tentative distances are still valid upper bounds for the distances to the extended subtree. One can simply re-insert every vertex v_i into the priority queue in Dijkstra's algorithm (with $\tau_i = 0$) as soon as it becomes part of the current subtree. Using this technique, the empirical running times are improved drastically, but unfortunately the worst-case running time is still $O(r(m + n \log n))$. Furthermore, as SPH is usually used repetitively, very similar information is computed again and again 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 to a 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 subsections. Note that we use these variants only as components of our other algorithms, not as stand-alone heuristics.

4.2.1 A Two-Phase Realization of the Repetitive Shortest Paths 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 55 Let P be a shortest path between a terminal z and a vertex v, such that there is a vertex v' on P with $z' := \mathit{base}(v') \neq z$ 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 that 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 Section 1.2) 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 $d(z, z') \leq d(z, v)$. 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 instance has many terminals and is not dense (the latter is almost always the case after applying reductions). Note that for instances 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 close 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 re-insertion 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 Small 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 DNH (see Section 1.2), which we also compute in the first phase. 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 for Path Heuristics

Although we designed the variants of the path heuristic described in the previous two subsections 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 the average running times and the average gaps to the optimal solution value for instance groups from SteinLib [Ste97] (see Appendix A for a description of the instance groups). The following algorithms are tested:

one-phase SPH: improved version of the shortest path heuristic with worst-case running time $O(r(m + n \log n))$, using the method of [PW02];

one-phase SPH, repeated: repetitive variant of the above, starting from up to 100 different vertices (terminals are preferred as starting points);

two-phase SPH, repeated: fast two-phase variant with worst-case running time $O(m+n \log n)$ from Section 4.2.2, again with up to 100 different starting points;

DNH: fast version of the distance network heuristic with worst-case running time $O(m + n \log n)$ from Section 1.2, for comparison. Note that there is not much point in repeating DNH, because the constructed tree $T'_D(R)$ will always have the same cost.

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

We present this table also to make it possible to compare (skillful implementations of) some of the best-known classical heuristics for the Steiner problem with our new, strong heuristics which will be presented in the next sections.

instance group	one-phase SPH		one-phase SPH, repetitive		two-phase SPH, repetitive		DNH	
	time (s)	gap (%)	time (s)	gap (%)	time (s)	gap (%)	time (s)	gap (%)
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 for path heuristics.

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, properties which it shares with DNH. For this reason, the repetitive variant is usually used. Here, we see that using the fast two-phase variant can often 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 55, and in spite of the rigid running time bound, all necessary shortest paths 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 variants reflect only different decisions in situations when ties between paths of equal length are broken. A different situation happens for example for the WRP3 and WRP4 instances, where many vertices have similar distances to many terminals. In the one-phase variant this leads to comparatively high running times, as distance values have to be re-checked 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 extensions, a precomputed shortest path is not available and one has to use paths from G' and the solution quality deteriorates.

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 execution of the one-phase SPH are similar (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 a paper of ours [PV01c] 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.

4.3.1 Heuristic Reductions

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 3.3.1). 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**.

4.3.2 Guiding the Heuristic

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 (directed) 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.

4.4.1 Searching in Auxiliary (Sub-) Graphs

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.

4.4.2 Searching in the Original Graph

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 46) 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 condition of the test DA (Section 3.3.2) 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.

4.4.3 Using the Row Generating Strategy

As in DUAL-ASCENT, dual feasible solutions and corresponding reduced costs for LP_C are calculated during the row generating algorithm (Section 2.11). 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 $c_{ij}(1 - \bar{x}_{ij})$ (where $\bar{x}$ 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 (non-trivial) 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 here, 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 our PRUNE heuristics on instance groups from SteinLib [Ste97] (see Appendix A for a description of the instance groups). All of our heuristics have a worst-case time describable by a polynomial of low order, as explained in the previous sections. Other strong heuristics 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 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 do 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. In Table 4.2, we compare the average running times and the average gaps to the optimal solution value. A stroke 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 (s)	gap (%)	time (s)	gap (%)	time (s)	gap (%)	time (s)	gap (%)
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 (strong) upper bound calculations.

From Table 4.2, one can see that SLACK-PRUNE is generally 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, while producing tighter upper bounds. Remember also that in [RUW02] the preprocessing time was not recorded. The quality of the solutions produced by SLACK-PRUNE is remarkable; even for larger incidence instances (I160, I320), which were constructed to be difficult for the known reduction techniques, the solutions are satisfactory. Note that the percentage of pruned vertices is a simple parameter for the trade-off 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 effectively.

Chapter 5

Exact Algorithms

5.1 Introduction

For an $\mathcal{NP}$-hard problem like the Steiner problem, it is usually not considered as surprising that (not very large) instances can be constructed that defy existing exact algorithms. On the other hand, it should not be surprising that relatively large instances can be solved exactly in fairly small times. An instructive case is the development in the context of the Steiner problem in recent years: Instances that were assumed to be completely out of the reach of exact algorithms some years ago can now be solved in reasonable times; and whole instance classes that were considered as of a "hard type" can now be routinely solved. But this does not happen by itself, say due to faster computers (although more powerful computers can help): The improvements in the running times are by orders of magnitude larger than the gains in the speed of the used computers. Major improvements happen when somebody comes up with new ideas that work well in practice.

For the Steiner problem in networks, there are many published works on exact solution building upon every classical approach (enumeration algorithms, dynamic programming, branch-and-bound, branch-and-cut) and the presented algorithms are sometimes quite involved. Again, we point to [HRW92] for a survey. Later major contributions (beside our work) were presented by Duin [Dui93] (advanced reductions and heuristics in a branch-and-bound framework), Koch and Martin [KM98] (improved branch-and-cut), and Uchoa et al. [UdAR02] (further improved reductions, again with branch-and-cut).

In this chapter, we describe how the components presented in the previous chapters are integrated into an exact algorithm, acting as an "orchestra".

In Section 5.2, we present an algorithm which exploits small width in (sub-) graphs, and show how it can be used profitably in combination with our other techniques in a more general context.

In Section 5.3, we describe the interaction between different components and how we take advantage of it to design a very powerful reduction process. Then we outline how this reduction process is used in a branch-and-bound framework.

In Section 5.4, we present some summarized experimental results and comparisons to the results of other authors. Detailed experimental results are included in Appendix A.

Finally, in Section 5.5 we make some concluding remarks, including some paths for further developments and transfer of the concepts to other problems.

5.2 Using (Sub-) Graphs of Small Width

In this section, we present a practical algorithm for solving the Steiner problem in graphs with a small width parameter (a formal definition of the used width concept is given in the following). 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 $\mathcal{NP}$-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 3.5) we can already profit from small width in subgraphs of a 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 and instances of geometric Steiner problems after FST generation (see Section 3.5.1).

The width concept here is closely related to path-width, as we will show in Section 5.2.4. For an overview of subjects concerning path-width and the more general notion of tree-width 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 tree-width 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 tree-width 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 path-width is described in [PT01], which is practical for a range of path-widths similar to the one considered here. We also adapted that approach to the Steiner problem, but the experimental results were not as good as with the one presented here.

A theoretically more powerful concept is branch-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 indicated that no further 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 3.5), and the Dynamic Programming approach for subgraphs presented here.

5.2.1 The Basic Approach

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 non-empty subsets, i.e., $\dot{\bigcup}_{1 \le i \le t} P_i = Q$ 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) $v_s \in Q$:

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

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.

5.2.2 A Dynamic Programming Realization

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 BORDER-DP (DP stands for Dynamic Programming), written in pseudocode:

```
BORDER-DP(G, R)        (assuming an ordering of the vertices)
1   s := 0; q := 0; opt := ∞;        (q : 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 : q := q + 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 q = |R| and |Pnew| = 1 :        (feasible Steiner tree)
17                          opt := min(opt, c(s,Pnew));
18  return opt;
```

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 $b = \log n / \log \log n$.

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. In [PV02b], we describe also some heuristics to recognize partitions that cannot lead to an optimal Steiner tree; and we give some examples how the presented algorithm works together with other components of our program to solve some previously unsolved benchmark instances.

5.2.3 Ordering the Vertices

In Section 5.2.4, 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 path-width b. The problem of deciding whether the path-width 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 non-terminals. 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,
- membership of v_s in R (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, thereby 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.

5.2.4 Relation to Path-Width

In this section, we show that every path-decomposition with path-width 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, we repeat the definition of path-width: A **path-decomposition** of a graph $G = (V, E)$ is a sequence of subsets of vertices $(U_1, U_2, \ldots, U_p)$, such that

1. $\bigcup_{1 \leq i \leq p} U_i = V$,
2. $\forall (v, w) \in E \quad \exists i \in \{1, \ldots, p\} : \quad v \in U_i \wedge w \in U_i$,
3. $\forall i, j, k \in \{1, \ldots, p\} : \quad i \leq j \leq k \Rightarrow U_i \cap U_k \subseteq U_j$.

The **path-width** of a path-decomposition $(U_1, U_2, \ldots, U_p)$ is $\max\{|U_i| \mid 1 \leq i \leq p\} - 1$. The **path-width** of a graph G is the minimum path-width over all possible path-decompositions of G.

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

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

Lemma 56 Every path-decomposition $U = (U_1, \ldots, U_p)$ can be transformed to a bijective path-decomposition $(U'_1, \ldots, U'_n)$ of no larger path-width.

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

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 57 Every (bijective) path-decomposition U can be transformed to a minimal (bijective) path-decomposition of no larger path-width.

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

Lemma 58 Let $(U_1, \ldots, U_n)$ 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: $U_s = \{v_s\} \cup B_{s-1}$.

Proof:

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

□

It follows that every path-decomposition of G can be transformed to a path-decomposition $U = (U_1, \ldots, U_n)$ of no larger path-width such that for an ordering of vertices according to the $start$ function of U it holds: $U_s = \{v_s\} \cup B_{s-1}$. 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 U by setting $U_s = \{v_s\} \cup B_{s-1}$. In each case, we have: $\max\{|U_s| \mid 1 \leq s \leq n\} - 1 = \max\{|B_{s-1}| \mid 1 \leq s \leq n\}$.

5.3 Building an Orchestra: An Exact Algorithm

In this section we describe the synthesis of an exact algorithm from the components described before.

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:

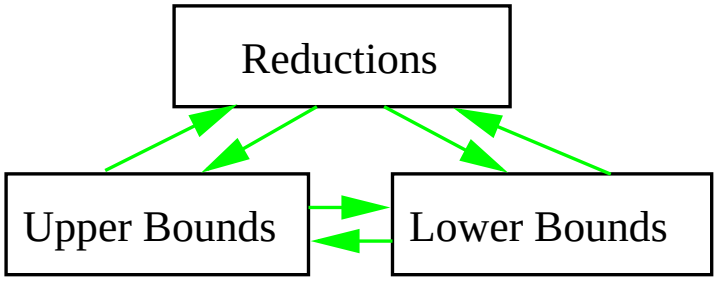


Figure 5.1: Interaction of the different components.

The bound-based reductions depend on upper and lower bounds (Section 3.3); 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 (Section 4.3). Further we use the structure of heuristic solutions (corresponding to good upper bounds) to guide the computation of lower bounds (see for example Section 2.9.2); and the information gained during the computation of lower bounds is used to guide the computation of upper bounds (Section 4.4). All in all, there is a mutual dependence between the three major

components: reductions, upper bounds, and lower bounds. But the interaction goes even further: The exact algorithms can be used again profitably in each of the components, namely for improving lower bounds (Section 2.12.2) and upper bounds (Section 4.5) and performing advanced reductions (Sections 3.4.2 and 3.5.3). This interdependence 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 not only is a major reason for the short solution times our algorithm very often achieves, but also enables us to solve instances 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 2.11) or, if the proportion of terminals to all vertices is high, the Lagrangian relaxation LaP_{T_0} (Section 2.10.1). To reduce the instances, we use all described alternative-based techniques (Section 3.2). In addition, we use the bound-based techniques DA (Section 3.3.2) and, in combination with row-generation, the test RG (Section 3.3.3). Furthermore, we use the extended reduction techniques (Section 3.4) and the partitioning-based reductions (Section 3.5). 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, as described in Section 5.2, whether we can solve the (sub-) instance with the algorithm BORDER-DP.

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.

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). In case several blocks exist, the problem can be solved by solving the instances corresponding to each relevant block separately (see Section 3.5.3), 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 59 Let T be a Steiner tree with all leaves being terminals in a network G. A block of G must be considered if and only if it contains an edge of T.

Proof: It is easy to see that a block must be considered if and only if there are two terminals z_k and z_l such that every path between z_k and z_l contains an edge in B, such blocks are called intermediate. Obviously, for any intermediate block B, 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 Summarized Experimental Results

In this section, we present some summarized results of our exact algorithm on the instances of SteinLib [Ste97]. Detailed results for each instance are reported in Appendix A.

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 instance groups, which have frequently been used by other authors. Note the the machine used by us is of more recent date and consequently faster than the others (although it is only half as fast as a (not uo-to-date) PC, see Appendix A). However, as obvious from the times given in the table, our improvement of the running times is of a much larger magnitude than could be explained by the relative machine speeds in each case.

instance-group	[Bea89] Cray X-MP	[BL98] SG Indigo	[CGR92] VAX 8700	[Dui93] i486	[KM98] Sun Sparc 20	[Uch01]* UltraSparc II	here Sunfi re
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.

*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 by him in zero time.

†Excluding instances not solved by [KM98].

We could solve every instance from SteinLib that has also been solved by any other author, usually much faster. Furthermore, we could solve 33 instances from different instance classes of SteinLib that have not been solved by other authors. See Appendix A for details.

5.4.1 Results on Geometric Instances

As described in Section 2.8, the bottleneck of the FST approach to geometric Steiner problems has usually been the second phase. The hitherto most successful algorithm for this phase is the one integrated in the program package GeoSteiner [WWZ01], which uses a branch-and-cut approach based on an MSTH formulation of the problem (Section 2.8.1). In Table 5.2, we compare the the average running times of GeoSteiner and our program for the second phase on the geometric instances from SteinLib. These tests (for both programs) were performed on a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor (for details see Appendix A). Detailed results on single instances can be found in [PV03, PV01d].

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 for the exact solution.

Additionally, we are able to solve some instances (fl1400, fl3795, fnl4461) that have not been solved before. In Table 5.3, we present the results of our program for the exact solution of the instances not solved by GeoSteiner in one day. We needed no branching for any of the instances in this section.

instance	size			optimum	time (s)
	$\|R\|$	$\|V\|$	$\|E\|$		(our program)
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. Each of the components (methods for computing lower and upper bounds and simplifying instances) achieves impressive results on its own. Also, the resulting exact algorithm solves many instances of different types in unprecedented small times, and in many cases, 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

[‡]Excluding instances not solved by GeoSteiner.

constructed to fool the currently available techniques. Here, the algorithm has gone into branching to solve the instances exactly. Currently, insightfully constructed, "pathological" instances cannot be solved in reasonable times even if they are of "medium" size (several hundreds of vertices), so new techniques are required here. On the other hand, the results on the other groups of instances indicate that such cases rarely arise naturally.

Regarding classes that originate from some application or were constructed "neutrally", even instances with tens of thousands of vertices can usually be solved in reasonable time. However, very large instances can still be a challenge. With respect to them, there are two main problems:

First, even though the running times and memory consumptions of many components can be described by polynomials of low degree, they can grow to a critical value for very large instances. Particularly, the DUAL-ASCENT procedure which is used relatively early in the reduction process, can be a bottleneck, especially because of the $\Theta(rn)$ memory it requires in our favorite implementation. Of course, new effective and efficient reduction techniques could be a solution. But in the short term, alternative (possibly distributed) implementations, which are tailored to very large instances, can remedy this problem.

A second problem is more severe. As already discussed in Section 3.5.4, when the instances get larger, the deviations of the linear (relaxed) solution from the integer one, but also the errors heuristics like DUAL-ASCENT are bound to make while approximating this solution, can accumulate. As a result, the bound-based methods can get weaker and eventually lose their impact. In the following subsection, we outline also some possible approaches to deal with this problem.

5.5.1 Some Paths for Further Improvements

Here we mention briefly some paths for further algorithmic improvements, especially for the solution of large or (currently) hard instances.

Branch-and-cut: Currently, we do not use branch-and-cut with (explicitly) globally valid cuts to improve the quality of the lower bounds. We found it more advantageous to use the calculated lower bounds and reduced costs for bound-based reductions and then try all reduction tests on the reduced graph. This process is often sufficient to solve a given instance without branching. Even when branching is necessary, the whole reduction process can make use of vertex branching, which is not the case in a branch-and-cut setting, because many operations performed by the reduction methods (e.g., substitution of a vertex by a clique over vertices adjacent to it, see Section 3.2.2) are difficult to translate profitably into the linear program maintained by a branch-and-cut algorithm. On the other hand, in our approach the generated constraints are discarded after every RG (row generation) test, so for the next such test, they have to be computed from scratch. For larger instances where most of the time is spent while solving linear programs, a better integration of the reductions into a branch-and-cut framework could be an improvement.

Using stronger relaxations: We have designed a collection of very strong relaxations (see the hierarchy in Section 2.7), but currently we can use them only restrictively (Section 2.11.1). If techniques could be developed to use stronger relaxations from the hierarchy efficiently for large instances, a major improvement could occur.

Improving the improvement methods: As already discussed in Section 2.12.3, the techniques for improving linear relaxations could be further improved. In particular, the potential of heuristic shrinks is perhaps still not fully exploited.

Adaptation to the FST approach: Although our program already achieves the currently best results for the second phase of this approach (Section 5.4.1), we do not use all information from the first phase, since we discard the knowledge about individual FSTs. This information could be used profitably, for example for the computation of lower bounds or in reduction methods. However, the resulting algorithm would not be a pure network algorithm (which only gets a weighted graph and the list of terminals as the input) anymore. But the success of the FST approach on geometric Steiner problems already justifies a tailored algorithm, and this approach could also be applicable to other Steiner problems, in particular in the context of phylogeny (Section 1.3.3).

Distributed computing: Many components in our program can be implemented distributedly in a natural manner, in particular the extended reduction techniques (Section 3.4) and the partitioning-based reductions (Section 3.5). In this way, the solution of very large instances of different types (for example VLSI instances) with hundreds of thousands of vertices should be well within reach.

5.5.2 Reliability and Verification

Another algorithmic challenge is the development of methods for reliable computation [Meh02], i.e., algorithms that come up with some kind of evidence that they produce correct results. Such algorithms return additional output (often called a witness) that enables a (fast) verification of the results. A weak version of a witness is mentioned in Section 3.6: After all reductions are performed, our algorithm transforms a tree in the reduced instance (which often consists of a single terminal) to one for the unreduced instance, thus certifying that a tree with the same value exists in the original instance (this is than checked by our program). Of course, this does not say much about the optimality of this tree.

An approach for certifying the optimality would be to present (the constraints of) a relaxation for the original instance with the same solution value as the weight of the produced Steiner tree. But since the reductions can change (improve) the value of relaxations (which is one their advantages after all), it is not easy to come up with such a relaxation. Note that also a solution and a certificate (for example a dual solution) should be reconstructed, because for larger instances current LP solvers could not solve the corresponding LPs even with a linear number of variables and constraints in the the size of original instance in reasonable time.

As a formal verification of the whole program is unlikely, certifying algorithms as components would already be an advantage. Major difficulties arise in the context of reduction techniques. Here we mention some of them:

- For the efficient implementation of reduction tests, we frequently use the following pattern: 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 a distance network, approximations of distances and bottleneck Steiner distances, a lower bound and corresponding 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., for the PT_m test (including the equality case) in Section 3.2.1) or that invalid parts are marked such that they do not cause harm (e.g., using neighbor lists for tagging some precomputed data as invalid in case an edge is deleted in Section 3.4.4). 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 3.2.1), it is easy to verify that a path exists with Steiner distance not

larger than the length of the edge by following the parent pointers in the shortest path tree and the edges in the tree $T'_D(R)$. For edges that are deleted later, this is no longer possible directly, as many edges on such paths may have been deleted.

- For many certifying algorithms the required times for producing the witness and verification of the result is (at least asymptotically) dominated by the running time of the main 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 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 alternative path can take time $O(n)$, while the test condition itself can be checked in nearly constant time.

Our methods with respect to other paradigms of reliable computation, namely "exact arithmetic" and "test and repair" have already been discussed in the previous chapters, in particular in the context of using the output of LP solvers for reductions (Section 3.3.3).

5.5.3 Transfer of Concepts to Other Problems

Many of methods developed or improved in this work in the context of the Steiner problem could also be useful for other hard combinatorial optimization problems. In the following, we briefly mention some concepts which appear most promising:

Relaxations and lower bounds: The techniques we used to compare and develop relaxations could also be helpful for the development or choice of relaxations for similar problems. Also the approaches we used for improving relaxations seem quite promising. In particular, the method of graph transformation introduced by us (Section 2.12.1) could be adapted to related problems.

Reduction: The approach of using the reduction process as the motor of the algorithm has proved to be quite powerful. Also our extended and combined reduction techniques (Section 3.4) and our approach of using partitioning as the basis for reductions (Section 3.5) could be used for other problems. In fact, upon our suggestion Hisao Tamaki has already integrated the latter technique successfully into his record-setting algorithm for computing short traveling salesman tours [Tam03].

Heuristics and upper bounds: Our reduction-based heuristics (Section 4.3) have proved to be quite powerful. The approach can also be applied to any other problem when similarly promising paths for reductions exist.

Finally, the interaction between the components has been a major reason for the efficiency of our exact algorithm; in fact, we are not aware of any other algorithm for a similar problem where this interaction has been exploited to such an extent. On the other hand, designing similarly effective and efficient techniques for each component and integrating them in a likewise efficient manner could take (if possible at all) many man-years of work.

Appendix A

Experimental Results of the Program Package

Here we report detailed results of our program package on the instances of the benchmark library SteinLib [Ste97]. In Table A.1, we give a brief description of the instance classes of SteinLib. For more comprehensive information, see [Ste97, KMV01]. The column "instances" gives the number of instances in the class. The column "status" shows whether all instances of this class have been solved by other authors and by us ("solved"), or only by us ("*solved*" in italics), or if there are some "unsolved" instances.

In Tables A.3-A.12, we report the results of our program package on each single instance of SteinLib, except for the instance groups I080, B, C, P, and ES10FST–ES500FST, which are too easy: All of them can be solved by our program exactly in very small times (typically less than one second).

For each instance, we give the value of the optimal solution (if reached) 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 63 instances have not been solved within the time limit. For these instances we give the computed lower and upper bounds after five hours (in italics). However, 9 of these instances could be solved using stronger extended reductions (see Section 3.4), and 13 could be solved by longer runs. For them we give the time for the exact solution in Tables A.15 and A.16. In total, there are only 41 unsolved instances left. These instances are from the instance groups SP, PUC, and I640, which were constructed with the aim of being difficult for known techniques.

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 to defy 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 2.11) 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.

We solved all instances that (to our knowledge) have been solved before. Furthermore, we solved 33 instances from different instance classes in SteinLib that have not been solved by other authors (see Table A.2).

For many classes, our exact solution program performs several orders of magnitude faster than programs of other authors, see Section 5.4 for a summarized comparison. It is acknowledged to be the strongest program for solving general Steiner problem instances at the time being [CD01, Ste97].

class name	instances	$\|V\|$	status	description
D	20	1000	solved	sparse with random weights and varying $\|E\|$ and $\|R\|$, from OR-Library [Bea90].
E	20	2500	solved	
X	3	52-666	solved	complete with Euclidean weights.
ES1000FST	15	2532–2984	solved	originally rectilinear instances, derived with GeoSteiner [WWZ01] from 1000 (rsp. 10000) random points in the plane or from TSPLIB [Rei91], see Section 2.8.
ES10000FST	1	27019	solved	
TSPFST	76	89–17127	*solved*	
I080	100	80	solved	so-called incidence networks, constructed to be difficult for known reduction 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 graphs [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 graphs with holes (not geometric) from VLSI design, SteinLib instance groups `alue`, `alut`, `diw`, `dmxa`, `gap`, `msm`, and `taq`.
LIN	37	53–38418	*solved*	grid graphs with holes (not geometric) from VLSI design.
WRP3	63	84–3168	*solved*	wire routing problems from industry [ZR03].
WRP4	62	110–1898	solved	
1R	27	1250	solved	2D cross grid graphs [Fre97].
2R	27	2000	solved	3D cross grid graphs [Fre97].

Table A.1: Classes of problem instances in SteinLib

All results presented here (and indeed all results in this work except in two cases explained below) were produced single-threaded on a Sunfire 15000 with 900 MHz SPARC III+ CPUs, using the operating system SunOS 5.9. We used the GNU g++ 2.95.3 compiler with the -O4 flag and the LP solver CPLEX version 8.0. As the Sunfire is a multi-processor machine with shared memory, it is slower than a single processor system with the same processor. In two cases (when comparing our program with the program Geosteiner [WWZ01] in Sections 2.13 and 5.4.1), we used (for both programs) 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, gcc 2.96 compiler and CPLEX 7.0. This machine was approximately twice faster than the Sunfire.

instance group	Table	instances
LIN	A.6	lin31, lin32, lin33, lin34, lin35, lin36, lin37
WRP	A.5	wrp3-83
TSPFST	A.7	fl1400fst, fl3795fst, fnl4461fst
PUC	A.13	bipe2p, bipe2u, cc5-3p, cc5-3u, hc8p, hc8u
SP	A.14	w3c571
I640	A.12	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.2: Instances solved by us and not solved by others.

instance	size			optimum	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			optimum	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.3: Results on the D and E-instances. Type: Sparse with random weights and varying $|E|$ and $|R|$, from OR-Library.

instance	size			optimum	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			optimum	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.4: Results on the 1R and 2R-instances. Type: 2D (respectively 3D) cross grid graph [Fre97].

instance	size			optimum	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			optimum	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.5: Results on the WRP-instances. Type: Wire routing problems from industry [ZR03]. Instances not solved here could be solved using stronger reductions, see Table A.15.

instance	size			optimum	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			optimum	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			optimum	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.6: Results on the VLSI and LIN-instances. Type: Grid graph with holes (not geometric) from VLSI design. Instances not solved here could be solved using stronger reductions, see Table A.15.

instance	size			optimum	time
	$\|V\|$	$\|E\|$	$\|R\|$		
es10000fst	27019	39407	10000	716174280	1398.9

instance	size			optimum	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			optimum	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			optimum	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.7: Results on the ES10000, ES1000 and TSP-instances. Type: Originally rectilinear instances, derived with GeoSteiner from 1000 (respectively 10000) random points in the plane or from TSPLIB.

instance	size			optimum	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.8: Results on the X-instances. Type: Complete with Euclidean weights.

instance	size			optimum	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.9: Results on the MC-instances. Type: Constructed difficult instances.

instance	size			optimum	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			optimum	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.10: Results on the I160-instances. Type: Incidence networks, constructed with the aim of being difficult for known reduction techniques.

instance	size			optimum	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			optimum	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.11: Results on the I320-instances. Type: Incidence networks, constructed with the aim of being difficult for known reduction techniques. Instances not solved here could be solved using longer runs, see Table A.16.

instance	size			optimum	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			optimum	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.12: Results on the I640-instances. Type: Incidence networks, constructed with the aim of being difficult for known reduction techniques. Instances i640-211, i640-34[1-5] could be solved using longer runs, see Table A.16.

instance	size			optimum	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			optimum	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.13: Results on the PUC-instances. Type: Constructed difficult instances: hypercubes, from code covering, and bipartite graphs [RdAR+01].

instance	size			optimum	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.14: Results on the SP-instances. Type: Constructed difficult instances, combination of odd wheels and odd circles, difficult for linear programming approaches.

instance	size			optimum	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.15: Instances solved using stronger extended reductions.

instance	size			optimum	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.16: Instances solved by longer runs.

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.

[ABCC03] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical Programming*, 97:91–153, 2003.

[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.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, 1993.

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

[APV03a] E. Althaus, T. Polzin, and S. Vahdati Daneshmand. Improving linear programming approaches for the Steiner tree problem. In K. Jansen, M. Margraf, M. Mastrolilli, and J. D. P. Rolim, editors, *Experimental and Efficient Algorithms, WEA 2003*, volume 2647 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2003.

[APV03b] 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.

[Aro03] S. Arora. Approximation schemes for NP-hard geometric optimization problems: a survey. *Mathematical Programming*, 97:43–69, 2003.

[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.

[BFR99] H.-J. Bandelt, P. Forster, and A. R¨ohl. Median-joining networks for inferring intraspecific phylogenies. *Molecular Biology and Evolution*, 16:37–48, 1999.

[BKJ83] K. Bharath-Kumar and J. M. Jaffe. Routing to multiple destinations in computer networks. *IEEE Transactions on Communications*, 31:343–351, 1983.

[BKM01] M. D. Biha, H. Kerivin, and A. R. Mahjoub. Steiner trees and polyhedra. *Discrete Applied Mathematics*, 112:101–120, 2001.

[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.

[CC02] M. Chlebík and J. Chlebíková. Approximation hardness of the Steiner tree problem on graphs. In M. Penttonen and E. Meineche Schmidt, editors, *Algorithm Theory - SWAT*, volume 2368 of *Lecture Notes in Computer Science*, pages 170–179. Springer, 2002.

[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.

[Cie98] D. Cieslik. *Steiner minimal tress*. Kluwer Academic Publishers, 1998.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[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*, 64:209–229, 1994.

[CR94b] S. Chopra and M. R. Rao. The Steiner tree problem II: Properties and classes of facets. *Mathematical Programming*, 64:231–246, 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.

[CVS03] Concurrent Versions System. `http://www.cvshome.org/`, 2003.

[DSR00] D-Z. Du, J. M. Smith, and J. H. Rubinstein, editors. *Advances in Steiner Trees*. Kluwer Academic Publishers, 2000.

[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.

[DW71] S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1971.

[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.

[For03] P. Forster. Personal communication, 2003.

[Fre97] C. 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.

[GGW93] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings 3rd Symposium on Integer Programming and Combinatorial Opt.*, pages 57–74, 1993.

[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 fractional Steiner forest and related problems. In *Proceedings of 43rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 500–, 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.

[GW95] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.

[GW96] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problem. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS, 1996.

[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. ExpLab – 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.

[HRS00] R. Hassin, R. Ravi, and F. S. Salman. Approximation algorithms for a capacitated network design problem. In K. Jansen and S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 167–176, 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.

[JMS03] K. Jain, M. Mahdian, and M. R. Salavatipour. Packing Steiner trees. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[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.

[KKPS00] J. K¨onemann, G. Konjevod, O. Parekh, and A. Sinha. Improved approximations for tour and tree covers. In K. Jansen and S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 184–193, 2000.

[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.

[Kle94] P. N. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Information Processing Letters*, 52(6):303–307, 1994.

[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.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[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.

[NRK01] R. Novak, J. Rugelj, and G. Kandus. Steiner tree based distributed multicast routing in networks. In X. Cheng and D.-Z. Du, editors, *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 327–351. Kluwer Academic Publishers, Dordrecht, 2001.

[PR00] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In *Automata, Languages and Programming*, pages 49–60, 2000.

[PS02] H.-J. Pr¨omel and A. Steger. *The Steiner Tree Problem: A Tour through Graphs, Algorithms and Complexity*. Vieweg, 2002.

[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.

[PV00a] 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.

[PV00b] T. Polzin and S. Vahdati Daneshmand. Primal-Dual Approaches to the Steiner Problem. Technical Report 14/2000, Universit¨at Mannheim, 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.

[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.

[RV99] S. Rajagopalan and V. V. Vazirani. On the bidirected cut relaxation for the metric Steiner tree problem. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 742–751, 1999.

[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.

[R¨oh98] H. R¨ohrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut f¨ur Informatik, Saarbr¨ucken, 1998.

[Ste97] SteinLib. `http://elib.zib.de/steinlib`, 1997. T. Koch, A. Martin, and S. Voß.

[Tam03] H. Tamaki. Personal communication, 2003.

[Tar79] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26:690–715, 1979.

[Tho97] M. 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.

[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.

[Voß90] S. Voß. *Steiner-Probleme in Graphen*. Hain-Verlag, Frankfurt/M., 1990.

[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.

[Zel93] A. Z. Zelikovsky. A faster approximation algorithm for the Steiner tree problem in graphs. *Information Processing Letters*, 46:79–83, 1993.

[ZR03] M. Zachariasen and A. Rohe. Rectilinear group Steiner trees and applications in VLSI design. *Mathematical Programming*, 94:407–433, 2003.

Index