Improved Algorithms for the Steiner Problem in Networks

Tobias Polzin Siavash Vahdati Daneshmand

Theoretische Informatik,
Universität Mannheim, 68131 Mannheim, Germany
email: {polzin,vahdati}@informatik.uni-mannheim.de

Abstract

We present several new techniques for dealing with the Steiner problem in (undirected) networks. We consider them as building blocks of an exact algorithm, but each of them could also be of interest in its own right. First, we consider some relaxations of integer programming formulations of this problem and investigate different methods for dealing with these relaxations, not only to obtain lower bounds, but also to get additional information which is used in the computation of upper bounds and in reduction techniques. Then, we modify some known reduction tests and introduce some new ones. We integrate some of these tests into a package with a small worst case time which achieves impressive reductions on a wide range of instances. On the side of upper bounds, we introduce the new concept of heuristic reductions. On the basis of this concept, we develop heuristics that achieve sharper upper bounds than the strongest known heuristics for this problem despite running times which are smaller by orders of magnitude. Finally, we integrate these blocks into an exact algorithm. We present computational results on a variety of benchmark instances. The results are clearly superior to those of all other exact algorithms known to the authors.

Keywords: Steiner problem; reduction tests; lower bounds; heuristics; exact algorithm

1 Introduction

The Steiner problem in networks is the problem of connecting a set of required vertices in a weighted graph at minimum cost. This is a classical $\mathcal{NP}$-hard problem with many important applications in network design in general and VLSI design in particular (see for example [15]).
The primary goal of the research presented in this paper has been the development of empirically successful algorithms. In section 2, we study some relaxations of the problem and methods for computing lower bounds using them; they are also heavily used in the following sections. In section 3, some reduction techniques are discussed, which play a central role in our approach. These techniques are also the basis of the reduction-based heuristics, which we introduce in section 4 on upper bounds. In section 5, the building blocks from the previous sections are integrated into an exact algorithm, which is shown to be successful empirically. Section 6 contains some concluding remarks.
Most of the material presented here originate from a joint work of the authors [20]. To achieve a reasonable size, only a fraction of the material there is chosen for this paper. Furthermore, we had to drop many technical details, giving priority to the main ideas and results.
Most of the background information relevant to this paper can be found in the book of Hwang, Richards and Winter [15]; we have tried to keep the notation compatible with that book. The basic definitions are repeated in the following section.

1.1 Definitions and Notations

For any undirected graph $G = (V, E)$, we define $n := |V|$, $e := |E|$, and assume that (v_i, v_j) and (v_j, v_i) denote the same (undirected) edge $\{v_i, v_j\}$. A network is here a weighted graph (V, E, c) with an edge weight function $c : E \to \mathbb{R}$. We sometimes refer to networks simply as graphs. For each edge (v_i, v_j), we use terms like cost, weight, length, etc. of (v_i, v_j) interchangeably to denote $c((v_i, v_j))$ (also denoted by $c(v_i, v_j)$ or c_{ij}). For any directed network $\vec{G} = (V, A, c)$, we use $[v_i, v_j]$ to denote the (directed) edge from v_i to v_j; and define $a := |A|$.
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 containing all terminals such that in $T_G(R)$, there is a path between every pair of terminals, and $\sum_{(v_i,v_j)\in T_G(R)} c_{ij}$ is minimized.
We define $r := |R|$. If the terminals are to be distinguished, they are denoted by $z_1, \ldots, z_r$. 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 [15]). Every instance of the undirected version can be transformed into an instance of the directed version in the corresponding bidirected network, fixing a terminal z_1 as the root. We define: $R_1 := R \setminus \{z_1\}$.

For each terminal z, one can define a neighborhood $N(z)$ as the set of vertices which are not closer to any other terminal. More precisely, a partition of V is defined ($d(v, w)$ denotes the length of a shortest path between v and w): $V = \dot{\bigcup\limits_{z\in R}} N(z)$ with $v \in N(z) \Rightarrow d(v, z) \leq d(v, t)$ (for all $t \in R$).
If $v \in N(z)$, we call z the **base** of v (written $base(v)$). In accordance with the parlance of algorithmic geometry, we call $N(z)$ the **Voronoi region** of z. We consider two terminals z_i and z_j as neighbors if there is an edge (v_k, v_l) with $v_k \in N(z_i)$ and $v_l \in N(z_j)$. Given G and R, the Voronoi regions can be computed in time $O(e + n \log n)$. Using them, a minimum spanning tree for the corresponding distance network $D_G(R)$ (we denote this tree by $T'_D(R)$) can be computed in the same time [19].

1.2 About Empirical Results in this Paper

In each of the following sections, we will report on the empirical behaviour of certain algorithms. We do not claim that algorithms can be evaluated beyond doubt by running them on a set of test instances. But when considering (exact) algorithms for an $\mathcal{NP}$-hard problem, there is no 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 needed; particularly in the context of exact algorithms, where even marginal differences (small fractions of a percent) in the value of the bounds can have a major impact on the behaviour of the algorithm.
In addition, we consider the comparability of results a critical issue, which strongly suggests using benchmark instances. There are two major benchmarks for the Steiner problem in networks: the collection in the OR-Library [4] and SteinLib [16]. The instances of OR-Library are much older, with the advantage that more comparative results exist on them. On the other hand, only one type of instances is represented (sparse and random). The library SteinLib is much more extensive, containing instances of all common types. But giving empirical results for all these instances in each section would make this paper unreasonably long, so we chose a compromise option: For the intermediary results (for example concerning upper bounds or reductions), we consider primarily the instances of OR-Library; comparable results for these components for other instances are very rare, anyway. For the final results of the complete algorithm, however, we give results for all types

of instances in SteinLib (except rectilinear instances, they are much better treated using their geometric characteristics).
Also it must be mentioned that for actual tests, we did not always implement the data structures and algorithms with the best known (worst case) time bound, especially if the extra work did not seem to pay off. So, statements concerning worst case time bounds for a component merely mean the possibility of implementation of that component with that bound.
All results in this paper are produced on a PC with a Pentium 300 MHz processor and 64 MB of main memory, using the operating system Linux 2.0.32. We always used the GNU g++ 2.7.2.1 compiler with the -O4 flag.

2 Relaxations and Lower Bounds

In this section we state some integer programming formulations of the Steiner problem and some methods for computing lower bounds on the basis of these formulations. In the context of lower bounds, (linear) relaxations of these formulations are of primary interest. Furthermore, the quality of the linear relaxations is of great importance for the success of bound-based reductions (see section 3.2) and relaxation-based upper bound calculations (see section 4.3). In this paper, we confine ourselves to those aspects which are relevant to the subject of obtaining good empirical results. An extensive theoretical investigation of various formulations and their relaxations is presented in a separate article by the authors [21]. Also, much more empirical observations are included in a thesis of the authors [20].

2.1 Formulations and Relaxations

In the following, integer programming formulations of both directed and undirected versions of the Steiner problem in networks are considered. Given a solution T ($\vec{T}$) in the underlying undirected network $G = (V, E)$ (respectively directed network $\vec{G} = (V, A)$), for each edge $(v_i, v_j) \in E$ (respectively $[v_i, v_j] \in A$) the binary variable X_{ij} (respectively x_{ij}) indicates whether the edge is in the solution (one) or not (zero).
For every integer program P, LP denotes the linear relaxation of P; and DLP denotes the dual of LP. For any (integer or linear) program Q, $v(Q)$ denotes the value of an optimal solution for Q.

2.1.1 Directed Cut Formulation

A **cut** in $\vec{G} = (V, A, c)$ is defined as a partition $C = \{\bar{W}, W\}$ of V ($\emptyset \subset W \subset V; V = W \dot{\cup} \bar{W}$). We use $\delta^-(W)$ to denote the set of edges $[v_i, v_j] \in A$ with $v_i \in \bar{W}$ and $v_j \in W$ ($\delta^+(W)$ and, for the undirected version, $\delta(W)$ are defined similarly). A cut $C = \{\bar{W}, W\}$ is called a **Steiner cut**, if $z_1 \in \bar{W}$ and $R_1 \cap W \neq \emptyset$.
The directed cut formulation was stated for the first time in [28]. (The undirected version was already introduced in [1].)

$$\boxed{P_C} \qquad \sum_{[v_i,v_j] \in A} c_{ij} x_{ij} \rightarrow \min,$$

$$\sum_{[v_i,v_j] \in \delta^-(W)} x_{ij} \geq 1 \quad (\{\bar{W}, W\} \text{ Steiner cut}), \tag{1.1}$$

$$x_{ij} \in \{0,1\} \quad ([v_i, v_j] \in A). \tag{1.2}$$

2.1.2 Spanning Tree Formulation

Here a degree-constrained reformulation of the problem in an augmented network $G_0 = (V_0, E_0, c_0)$ is used, which is produced by adding a new vertex v_0 and connecting it through zero cost edges to all non-terminals and a fixed terminal (say z_1). This leads to the following integer programming formulation, introduced in [3].

$$\boxed{P_{T_0}} \qquad \begin{array}{rcll} \sum_{(v_i,v_j)\in E} c_{ij}X_{ij} & \rightarrow & \min, & \\ \{(v_i,v_j) \mid X_{ij}=1\} & : & \text{builds a spanning tree for } G_0, & (2.1) \\ X_{0k}+X_{pq} & \leq & 1 \quad (v_k \in V\backslash R \;;\; (v_p,v_q)\in\delta(\{v_k\})), & (2.2) \\ X_{ij} & \in & \{0,1\} \quad ((v_i,v_j)\in E_0). & (2.3) \end{array}$$

The requirement (2.1) can be expressed by linear constraints. In the following, we assume that (2.1) is replaced by the following constraints.

$$\begin{array}{rcll} \sum_{(v_i,v_j)\in E_0} X_{ij} & = & n, & (2.4) \\ \sum_{(v_i,v_j)\in E_0;\; v_i,v_j\in W} X_{ij} & \leq & |W|-1 \quad (\emptyset \neq W \subset V_0). & (2.5) \end{array}$$

The constraints (2.4) and (2.5), together with the non-negativity of X, define a polyhedron whose extreme points are the incidence vectors of spanning trees in G_0 (see for example [18]). Thus, no other set of linear constraints replacing (2.1) can lead to a stronger linear relaxation.
In [21], we prove a clear relation between the linear relaxations of the two presented formulations:

Lemma 2.1 For all instances of the Steiner problem holds: $v(LP_C) \geq v(LP_{T_0})$. The ratio $\frac{v(LP_{T_0})}{v(LP_C)}$ can be arbitrarily close to zero.

2.2 Using Relaxations for Computing Lower Bounds

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

2.2.1 The Spanning Tree Formulation and Lagrangean Relaxation

A Lagrangean relaxation LaP_{T_0} of the tree formulation P_{T_0} is described in [3], relaxing the degree constraints (2.2). After this, a subgradient optimization of the Lagrangean multiplier problem can be used, which involves calculating a minimum spanning tree in each iteration. Using this approach, the value $v(LP_{T_0})$ can be approximated fairly fast (this relaxation has the integrality property). The problem here is the value $v(LP_{T_0})$ itself. Lemma 2.1 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 modelling 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 Lagrangean 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 Lagrangean multipliers. Using data structures for efficient handling of tree bottlenecks and alternative chords (see [22, 24]) 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 [9], 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 [5], 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, in [21] we prove that 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.

2.2.2 The Cut Formulation, Dual Ascent and Row Generating

Considering the relaxation LP_C, the situation is to some degree converse to the case of LP_{T_0}. It is known that $v(LP_C)$ does not deviate from the optimum by more than 50% [13]. All our empirical investigations strongly suggest that this is an extremely tight relaxation. As an example, for all D-instances of the OR-Library $v(LP_C)$ is equal to $v(P_C)$. Even for the instances where there is a gap, the knowledge of a solution of LP_C has been usually sufficient to solve the instance exactly (without branching) through bound-based reduction techniques (section 3.2). So, the really interesting problem is how to calculate (or sufficiently approximate) a solution for LP_C.
The direct approach of solving the complete linear program using a standard LP-solver is not practical, even for the equivalent multicommodity flow relaxation [28], 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.
Therefore, we have investigated some alternative methods: dual ascent (and Lagrangean relaxation) and a row generating approach.

Dual Ascent: A fast dual ascent algorithm that generally provides fairly good lower bounds was described in [28] for the equivalent multicommodity flow relaxation. Below, we give an alternative description of it as a dual ascent algorithm for LP_C, which we call **DUAL-ASCENT**:

- Initialize the reduced costs ($\tilde{c} := c$), the lower bound ($lower := 0$) and assume all dual variables u as been set to zero.
- In each iteration, choose a terminal $z_t \in R_1$ not reachable from the root by edges of zero reduced cost. Let W, $z_t \in W$, be the smallest set such that $\{\bar{W}, W\}$ 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 [8]). Empirically, it is usually faster than this time bound would suggest.
The algorithm DUAL-ASCENT achieves already very good results. Out of the 20 D-instances of the OR-Library, a DUAL-ASCENT run yields the optimum (i.e. $v(P_C)$) for 12 instances. The average gap between lower bound and optimum is 0.4%, and the average running time is 0.4 seconds.
A critical point in this algorithm is the choice of z_t in each iteration and, for the undirected version, the choice of the root z_1. Although it has been shown [14] that $v(LP_C)$ is independent of the choice of the root vertex, the lower bound calculated by DUAL-ASCENT is not. For this reason we start DUAL-ASCENT with different roots if a strengthening of the bound is needed. This

method also improves the performance of the reductions and upper bound calculations that are done in combination with DUAL-ASCENT (see sections 3.2.2 and 4.3). Again, for the D-instances, considering up to five different roots improves the average gap to 0.07%; achieving the optimum for 16 instances. The average running time for this lower bound calculation (including the time for a combined upper bound calculation) is 0.8 seconds.
Even using this amplification, there are still cases where DUAL-ASCENT does not reach the value $v(LP_C)$. We tested different criteria for the choice of z_t in each iteration. Our standard criterion is: Choose z_t, so that W is smallest. We had some success with the following idea that tries to guide DUAL-ASCENT with the help of a heuristically constructed Steiner tree: Assume that the upper bound is already optimal. DUAL-ASCENT can reach the optimum only if in each set $\delta^-(W)$ there is exactly one edge of the corresponding Steiner tree. Of course this criterion can not always be realized, especially if the best known Steiner tree is not optimal or $v(LP_C) < v(P_C)$. Nevertheless, it is a heuristic criterion that in many cases leads to better lower (and, indirectly, upper) bounds.

A natural alternative for a better approximation of $v(LP_C)$ builds upon a Lagrangean relaxation of the multicommodity flow formulation; an approach already used in in [2] (but with the much weaker undirected relaxation; see also [15]). Relaxing the constraints which bind edge and flow variables together, the problem decomposes into (mainly) $r-1$ single pair shortest path problems, which can be solved in time $O(r(e + n\log n))$. This relaxation has the integrality property, and can be used in combination with subgradient optimization to approximate $v(LP_C)$. In [20], 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 paper and is replaced by the following approach.

Row Generating: To get an optimal solution for LP_C, one can begin with a subset of constraints of LP_C as the initial program, and successively solve the current program, find Steiner cut inequalities violated by the current solution x, add them to the program, and iterate this process by reoptimizing the program, until no Steiner cut inequality is violated anymore. This is an approach already used by many authors (see for example [7, 5, 17]).
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 $z_t \in R_1$ as the sink and the current x_{ij}-value as the capacity of the arc $[v_i, v_j]$. Although there are other (heuristical) ways to find such violated inequalities, using those corresponding to minimum cuts usually leads to better overall results. Indeed, it is even very advantageous to find in each case a minimum capacity cut with a minimum number of cut edges, an idea already used in [17]. 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), so that the overall results (especially the total number of needed reoptimizations) are clearly superior. It must be mentioned that in our implementation, the time for finding all the $r-1$ minimum cuts is dominated by the time for reoptimizing the linear programs.
For computing minimum cuts, we implemented the highest-label preflow-push algorithm with several auxiliary heuristics, including the global and the gap relabeling heuristics [6]. Although no better time bound than $O(n^2\sqrt{e})$ can be given for this algorithm, using the mentioned heuristics the empirical running times were much better described by $O(n^{1.5})$. As long as only minimum cuts from the sink-side are to be computed, only the first stage of the algorithm has to be performed. Besides, in this context several additional heuristics can be used to improve the empirical times further; for example, sinks which are reachable from the root (or another terminal) by paths of capacity no less than 1 need not be considered.
For (re-)optimizing the linear programs, we use the dual simplex routine in the callable library of CPLEX 5.0. 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 reached already in the first iteration; and the number of needed reoptimizations and the time needed per reoptimization are comparable to the case without these cuts *after* reaching this lower bound value, so that the overall times are clearly improved.
In [17], some additional groups of constraints are used to to strengthen the linear programs, including the following ones, which we call **flow-balance constraints**:

$$\sum_{[v_j,v_i]\in\delta^-(\{v_i\})} x_{ji} \leq \sum_{[v_i,v_j]\in\delta^+(\{v_i\})} x_{ij} \qquad (v_i \in V \setminus R). \tag{3.1}$$

In [21], we prove that these constraints can indeed enhance the value of the relaxation. 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 [17] cannot enhance the value of the relaxation (see [21]), a group of them can speed the process up if its violated members are added to the current program, these are constraints of the form:

$$\sum_{[v_k,v_i]\in A, v_k\neq v_j} x_{ki} \geq x_{ij} \qquad (v_i \in V \setminus R;\ [v_i,v_j] \in \delta^+(\{v_i\})). \tag{3.2}$$

To save time and space, we do some garbage collection every ten iterations, purging the constraints which 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 needed) add variables which do not price out correctly (have negative reduced costs) to the program (activate them); a correct lower bound is given when all non-active variables have non-negative reduced costs with respect to the current dual solution. We have tried several schemes for using this idea, but could not achieve decisive *additional* improvements through these schemes. The main reason is that, because of our massive usage of reduction techniques (see the next section), 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 3.2).

3 Reductions

It has been known for some time that reductions can have a decisive effect when solving $\mathcal{NP}$-hard problems in general and the Steiner problem in particular, with the PhD thesis of Cees Duin [8] being a milestone for the latter. Our work in this context has been threefold: Firstly, we designed efficient realizations of some classical tests, which would have been too time-consuming for large instances in their original form, especially for application in heuristics. Furthermore, we designed some new tests, filling some of the gaps the classical tests had left. Notice that each test is specially effective on a certain type of instances, having less (or even no) effect on some other ones, so it is important to have a large arsenal of tests at one's disposal. Finally, we integrated these tests into a packet, using some nontrivial techniques. It should be emphasized that the most impressive achievements of reductions are mainly due to the interaction of different tests, achieving results which are incomparable to those each single test could achieve on the same instance on its own.
We distinguish between two major classes of reduction tests: The **alternative-based** 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. The **bound-based** tests use a lower bound for the value of an optimal solution under the assumption 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 are successful if such a lower bound exceeds a known upper bound.

3.1 Alternative-based Reductions

In this subsection we present a collection of alternative-based tests, including some new versions of classical tests and some new tests, which can all be realized in time $O(e + n \log n)$.
In the context of alternative-based reductions, the notion of bottleneck Steiner distances (also called special distances) is often helpful. Any path P between two vertices v_i and v_j in a network G can be broken down into one or more **elementary paths** between v_i, successive terminals and v_j. The **Steiner distance** between v_i and v_j along P is the length of the longest elementary path in P. The **bottleneck Steiner distance** $b(v_i, v_j)$ between v_i and v_j is the minimum Steiner distance taken over all paths between v_i and v_j in G.

3.1.1 PT$_m$ and Related Tests

The test PT$_m$ (Paths with many Terminals) was introduced in [10]:

PT$_m$ test: Every edge (v_i, v_j) with $c(v_i, v_j) > b(v_i, v_j)$ can be removed from G.

The PT$_m$ test is one of the most effective classical exclusion tests, but it is too time-consuming for large instances in its original form. Here we consider a fast realization of this test which also uses inaccurate information. The modifications follow the same principal ideas as in [8]. Later we will simply refer to this modified version as the PT$_m$ test. Empirically, 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 readily observes that the bottleneck Steiner distance $b(z_i, z_j)$ can be computed by determining a bottleneck on the fundamental path between z_i and z_j in the spanning tree $T'_D(R)$, which can be constructed in time $\mathrm{O}(e + n \log n)$. Each such bottleneck can be trivially computed in time $O(r)$, leading to a total time $O(mr)$ for m inquiries ($m \in O(\min\{e, r^2\})$). 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 [24]) to achieve time $\Theta(r^2)$ for all inquiries. One can go further and solve the problem as an off-line variant for all m inquiries in time $O(m\alpha(m, r))$ using the Eval-Link-Update data structure [22]. 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 $O(e + n \log n)$ for the whole test: One can sort the edges of $T'_D(R)$ and then process them as links in increasing cost order, building a binary tree (whose internal nodes represent the edges of $T'_D(R)$) using a suitable auxiliary union-find data structure. This transforms the problem to an instance of the off-line nearest-common-ancestor problem, which is solvable, for example, in $O(m)$ using a depth-first search strategy [22]. This leads to a total time $O(m + r \log r)$ for all m inquiries.
For non-terminals v_i and v_j, one can use an upper bound for the bottleneck Steiner distance $b(v_i, v_j)$ 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 $O(e + n \log n)$, as described in [8]. After that, one works with the upper bound $\hat{b}(v_i, v_j) := \min_{a,b \in \{1,\ldots,k\}} \{\max\{\ d(v_i, z_{i,a})\ ,\ b(z_{i,a}, z_{j,b})\ ,\ d(z_{j,b}, v_j)\}\}$ instead of $b(v_i, v_j)$. But we do not precompute the $\hat{b}$-values, because very often not all the k^2 combinations have to be checked; for example if the test condition turns out to be already satisfied during the computation (or, of course, if v_i or v_j were a terminal). More importantly, many additional observations can be used to do without $\hat{b}(v_i, v_j)$ altogether. For example the lower bound $\tilde{b}(v_i, v_j) := \max\{d(v_i, base(v_i)), d(v_j, base(v_j))\}$ (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 $\tilde{b}(v_i, v_j) = \hat{b}(v_i, v_j)$. If v_i and v_j belong to different Voronoi regions and $c(v_i, v_j) < \tilde{b}(v_i, v_j)$, then the test cannot be successful for (v_i, v_j). Furthermore, precomputing the $\hat{b}$-values (which can need time $\Theta(n^2)$) would destroy the total time $O(e + n \log n)$ for performing this test on all edges.
An additional observation leads to a simple, very fast test, which is sometimes extremely powerful:

Observation 3.1 Let $\hat{B}$ be the length of the longest edge in $T'_D(R)$. Every edge (v_i, v_j) with $c(v_i, v_j) > \hat{B}$ can be removed from the network.

Proof: Suppose there is a Steiner minimal tree T containing an edge (v_i, v_j) with $c_{ij} > \hat{B}$. 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 $\hat{B}$. This path contains an elementary path P connecting C_i and C_j, whose length is at most $\hat{B}$. Reconnecting C_i and C_j by P yields a graph spanning all terminals and shorter than T, a contradiction. □

Notice that using this test, one can eliminate some edges which could not be eliminated by the PT_m test (even in its original form).

Since the tests above only consider paths with at least one terminal, they miss some of the edges the simple test LE (Long Edges) [2, 15] would eliminate. On the other hand, after execution of other tests the graph is often sparse. So a weakened version of LE, which simply searches for shorter paths from both ends of an edge, can be effective. With the additional restriction that during the examination of each edge not more than a constant number of edges are visited in search for an alternative path, one gets the total time $\Theta(e)$ for this modified test, which we call **Triangle**. This test is sometimes a nice complement to the PT_m test (as described above), especially if the proportion of terminals to all vertices is not high.

A test like PT_m can actually be extended to the case of equality, but removing edges in case of equality can change the (restricted) bottleneck Steiner distances, which makes a recalculation of these distances after each deletion necessary. We have 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, so they are dropped here. 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 only with a measure like this.

3.1.2 NTD_k

The test NTD_k (Non-Terminals of Degree k) was introduced in [10]:

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 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 $(\Delta, \Delta \times \Delta, b)$. If this condition is satisfied, one can remove v_i and incident edges, introducing for each two vertices v_j and v_k adjacent to v_i an edge (v_j, v_k) with length $c_{ij} + c_{ik}$ (and keeping only the shortest edge between each two vertices).

The special cases with k (degree of v_i) in $\{1, 2\}$ can be implemented with total time $O(n)$ (for examination of all non-terminals). For $k \in \{3, \ldots, 7\}$ we use the $\hat{b}$-values instead of the exact bottleneck Steiner distances, as described in section 3.1.1. Again, empirically only a marginal difference of effectiveness is observed between the original and the modified version. As before, we do not precompute the $\hat{b}$-values, so the modified version has total time $O(e + n \log n)$.

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

3.1.3 NV and Related Tests

The test NV (Nearest Vertex) is a classical inclusion test [2, 15]:

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 with 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 an observation given below. For this purpose, we use Voronoi regions again, saving some extra information while computing the regions. Let $distance(z_i)$ be the length of a shortest path from z_i to another terminal z_j over the edge (z_i, v_i'), computed as follows: Each time an edge (v_i, v_j) with $v_i \in N(z_i), v_j \in N(z_j), z_j \neq z_i$ is visited, it is checked whether v_i is a successor of v_i' in the shortest paths tree with root z_i (simply done through marking the successors of v_i'). In such a case $distance(z_i)$ is updated to $\min\{distance(z_i), d(z_i, v_i) + c(v_i, v_j) + d(v_j, z_j)\}$. Now we have:

Observation 3.2 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 observation 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 there exists a terminal z_j 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 observation, the test NV can be performed for all terminals in time $O(e + n \log n)$. 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):

Observation 3.3 Let z_i be a terminal, and (v_1, v_1') and (v_2, v_2') the shortest and second shortest edges which leave the Voronoi region of z_i ($v_1, v_2 \in N(z_i)$, $v_1', v_2' \notin N(z_i)$; we call such edges **links**). The edge (v_1, v_1') belongs to at least one Steiner minimal tree, if $c(v_2, v_2') \geq d(z_i, v_1) + c(v_1, v_1') + d(v_1', z_j)$, where $z_j = base(v_1')$.
Proof: Suppose that the edge (v_1, v_1') is not in any Steiner minimal tree. Consider such a tree T and the path between z_i and z_j in T. An edge on this path must leave the Voronoi region of z_i. Removing this edge and inserting (v_1, v_1') and two shortest paths to z_i and to z_j, we get a subgraph that includes (v_1, v_1'), spans all terminals and is no longer than T, a contradiction. □

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

The classical test SE (Short Edges) [10, 15] is a more powerful inclusion test. We have observed that even this test can be implemented with time $O(e + n \log n)$. But although this test is more effective than NV and SL in a single application, the difference almost vanishes when the reduction tests are iterated. Therefore, we only use the much simpler, empirically faster tests NV and SL in our actual implementations.

3.1.4 Path Substitution (PS)

We have designed another alternative-based reduction test that is more general than the previous tests in two ways: The test PS examines several edges along a path, instead of examining elementary graph objects (like single edges and vertices). If the test is successful, some of these edges can be deleted at once. The other more general aspect is a consequence of the first: Searching for alternatives for a path, it is not sufficient anymore 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 strong requirements as conditions.
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 observation 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 degree 2 or 4 are more or less straightforward.

Observation 3.4 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 which are not contained in P. Let $d_0(v_i, v_j)$ be the length of a shortest path between v_i and v_j that does not contain (v_0, v_1), and $d_P(v_i, v_j)$ (for v_i and v_j in P) the length of the subpath of P between v_i and v_j.
The edge (v_0, v_1) can be deleted if for all $i \in \{1, \ldots, l\}$ there are functions f^i and g^i such that:
I) for all v_i^k adjacent to v_i and for $k_0 = f^i(k)$: $d_P(v_0, v_i) \geq d_0(v_0^{k_0}, v_i^k)$,
II) for all $v_0^{k_0}$ adjacent to v_0 and for $k = g^i(k_0)$: $d_P(v_0, v_i) \geq d_0(v_0^{k_0}, v_i^k)$, $c(v_0, v_0^{k_0}) \geq c(v_i, v_i^k)$.
Proof: Suppose all Steiner minimal trees contain the edge (v_0, v_1). Consider such a tree T, and let $t \geq 1$ be the smallest index such that there is an edge (v_t^k, v_t) in T. Notice that the degree of v_0 in T must be greater than 1 and that all edges between v_0 and v_t on P must be in T. There are two cases:
1) In T, v_0 has degree three. Choose k such that (v_t^k, v_t) is in T. Let $k_0 = f^t(k)$. Remove the edges on the path $(v_0, v_1, \ldots, v_{t-1}, v_t)$ from T. The resulting components can be reconnected without reinserting (v_0, v_1) by a path between $v_0^{k_0}$ and v_t^k which is not longer.
2) In T, v_0 has degree two. Choose k_0 such that $(v_0^{k_0}, v_0)$ is in T. Let $k = g^t(k_0)$. Remove the edges on the path $(v_0^{k_0}, v_0, v_1, \ldots, v_{t-1}, v_t)$ from T. The resulting components can be reconnected without reinserting (v_0, v_1) by a path between $v_0^{k_0}$ and v_t^k and the edge (v_t^k, v_t). Again the inserted edges together are not longer than the removed edges.
In both cases, we have a subgraph that does not contain (v_0, v_1), spans all terminals and is not longer than T, a contradiction. $\square$

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 is 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 observation 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 $O(e)$ can be guaranteed, without impeding its reduction potential noticeably.
This version of the test is usually effective only for some sparse graphs (including some VLSI-instances). On such instances, 5-10% of edges could frequently be removed using this test alone.

3.2 Bound-based Reductions

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

3.2.1 Using Voronoi Regions

The Voronoi regions can be used to determine a lower bound for the value of an optimal solution under some additional assumptions (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 next terminals to v_i, as described in section 3.1.1. The following observation can be used to eliminate a non-terminal.

Observation 3.5 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 which 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:

Observation 3.6 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 observation 3.5.

One can also define a test performing the same actions as NTD_k when it is successful, using the following observation:

Observation 3.7 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 observation 3.5.

Intuitively, one expects that a 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:

Observation 3.8 Consider the auxiliary network $G' = (R, E', d')$, in which two terminals are adjacent if and only if they are neighbors in the original network, defining:
$d'(z_i, z_j) := \min\{\min\{d(z_i, v_i), d(z_j, v_j)\} + c(v_i, v_j) \quad | \quad v_i \in N(z_i), v_j \in N(z_j)\}$.
The weight of a minimum spanning tree for G' is a lower bound for the weight of any Steiner tree for the original instance (G, R).

Proof: We will prove the observation by transforming a Steiner minimal tree $T_G(R)$ into a spanning tree T' in G' without increasing the cost. For guiding this transformation we construct an auxiliary tree T'' by contracting all edges of $T_G(R)$ that are entirely in one Voronoi region. We consider T'' as a rooted tree with an arbitrary root z_r. Beginning with isolated terminals as T', each step of the transformation removes the path from one leave of T'' to its parent and inserts an edge of G' into T'. Throughout the transformation the following invariant (†) holds: In each component of T', there is exactly one terminal that has not been removed from T''. In the beginning (†) holds trivially.
Each step of the transformation is performed as follows: Choose any leave z_i of T'' such that all vertices $v_i \in N(z_i) \setminus \{z_i\}$ have at most one successor in T''. Notice that there is always such a z_i in T'', because the number of leaves is greater than the number of non-terminals with more than one successor. From z_i we move in the direction of the root until we reach a terminal z_l. The path from z_i to z_l in T'' is denoted by P_i''. The corresponding path in $T_G(R)$ is denoted by P_i. Now we look at the bases of the vertices on P_i. Let v_j be the last vertex on P_i whose base z_j is connected to z_i in T', and v_k the first whose base z_k is not connected to z_i in T'. The invariant (†) guarantees that such vertices v_j and v_k exist, because not all bases $z_i, \ldots, z_l$ of the vertices of P_i can belong to the same component of T', since z_i and z_l have not been removed from T''. We denote with P_i' the part of P_i between z_i and v_k. The length of P_i' is at least $d'(z_j, z_k)$, because $d'(z_j, z_k) \leq d(z_j, v_j) + c(v_j, v_k) \leq d(z_i, v_j) + c(v_j, v_k)$. Remove the subpath of P_i'' beginning from z_i until a vertex in T'' with degree greater than 2 or z_l is reached. The edge (z_j, z_k) is inserted in T', so (†) remains valid.
After all terminals except z_r have been removed from T'', $r-1$ edges have been inserted into T' without creating a cycle, so T' is a spanning tree at the end.
We show now that each two paths P_a' and P_b' corresponding to terminals z_a and z_b are edge-disjoint (in the following simply denoted as disjoint). There are two cases:
I) z_a is a successor of z_b in T'' (or vice versa): If there are common edges in P_a' and P_b', there must be common edges or (if they are contracted) at least common vertices in P_a'' and P_b''. But the paths P_a'' and P_b'' are disjoint and have at most one vertex in common, namely z_b. Hence, common edges of P_a' and P_b' must lie entirely inside the Voronoi region of z_b and have been contracted in T'' into z_b. When z_a is chosen, neither z_a nor z_b has been removed from T''. Because of (†), at this time z_a and z_b are not connected in T'. So, for P_a the vertex v_j (as defined above) must be outside $N(z_b)$, and there is no edge in P_a' that is entirely inside $N(z_b)$. Therefore, the paths are disjoint.
II) z_a and z_b are successors of a terminal z_c in T'': Assume that z_a is chosen before z_b. There are disjoint paths in $T_G(R)$ from a vertex v_d to z_a, z_b and z_c ($v_d = z_c$ is possible). Let $z_d := base(v_d)$. Suppose that the paths P_a' and P_b' are not disjoint. So, they must both contain edges of the path between v_d and z_c. Thus, when z_a is chosen, z_d must be connected to z_a in T'. Because z_a has not been removed from T'', it follows from (†) that z_d was removed from T'' before. This is only possible if $v_d \neq z_d$. When z_d was chosen, v_d had only one successor in T''. Thus, the path between z_d and z_b has been removed even sooner. This means that z_b had to be chosen before z_a, a contradiction.
Since each edge of T' corresponds to a path in $T_G(R)$ with at least the same length and all these paths are disjoint, the spanning tree T' is not longer than $T_G(R)$, and the observation follows. □

This observation can be extended to a test condition; for example, for any non-terminal v_i, the weight of such a spanning tree minus the length of its longest edge plus $d(v_i, z_{i,1}) + d(v_i, z_{i,2})$ is a lower bound for the weight of any Steiner minimal tree that contains v_i. The resulting test is very fast: The network G' can be determined without much extra work while computing the Voronoi regions, and a minimum spanning tree for it can be computed in time $O(e + r\log r)$.
For computing upper bounds in this context, we use a modified path heuristic with time $O(e + n\log n)$, which is described in section 4.1. So, all these tests can be performed in time $O(e+n\log n)$; 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 the 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 specially successful if the proportion of terminals to all vertices is high. Besides, this test was the basis for the development of the strong PRUNE-heuristics, which are presented in section 4.2.

3.2.2 Using Dual Ascent

The information provided by the algorithm DUAL-ASCENT (section 2.2.2), namely the lower bound *lower* and the reduced costs can be used to design another bound-based reduction test. Here we use an extremely simple, but very helpful observation, which we will exploit frequently later on:

Observation 3.9 Let $G = (V, A, c)$ be a (directed) network (with a given set of terminals) and $\tilde{c} \leq c$. Let $lower'$ be a lower bound for the value of any (directed) Steiner tree in $G' = (V, A, c')$ with $c' := c - \tilde{c}$. For each $\tilde{x}$ representing a feasible Steiner tree for G, it holds: $lower' + \tilde{c}^T\tilde{x} \leq c^T\tilde{x}$.
Proof: $c^T\tilde{x} = c'^T\tilde{x} + \tilde{c}^T\tilde{x} \geq lower' + \tilde{c}^T\tilde{x}$. □

Now consider the reduced costs provided by DUAL-ASCENT as $\tilde{c}$: One can readily 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 $\vec{T}$, $lower + \sum_{[v_i,v_j]\in A} \tilde{c}_{ij}\tilde{x}_{ij}$ represents a lower bound on the weight of $\vec{T}$.
This observation can be used to compute lower bounds for the value of an optimal Steiner tree under certain assumptions, for example, that the tree contains a certain non-terminal. The resulting tests are basically identical to the tests IRA and IRAe, which are introduced in [8], 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 $\sum_{[v_i,v_j]\in A} \tilde{c}_{ij}\tilde{x}_{ij}$ 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 3.1.1. 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 $O(e + n \log n)$ 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. We call this collection of tests **DA** (standing for Dual Ascent).
Handling 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 gained a substantial improvement in 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 $O(e + n \log n)$ of the other tests hitherto presented, somewhat unsatisfactory, especially because the other parts of the test can indeed be performed in time $O(e + n \log n)$.
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 jointly 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 $O(e + n \log n)$ uses the observation that it is possible to increase many dual variables around a terminal at once.

Observation 3.10 Choose a terminal $z_t \in R_1$. Define $d'(v_i) := \min\{d(v_i, z_t), d(z_1, z_t)\}$. For all Steiner cuts $\{\bar{W}, W\}$ set the dual variable $u_W := \max\{0, \min_{v_j \notin W}\{d'(v_j)\} - \max_{v_j \in W}\{d'(v_j)\}\}$. Then $\sum_{W,[v_a,v_b]\in\delta^-(W)} u_W = \max\{0, d'(v_a) - d'(v_b)\} \leq c_{ab}$ for all edges $[v_a, v_b] \in A$.
Proof: Let $v_1, v_2, \ldots, v_n$ be the vertices of V sorted by their distances to z_t in ascending order. Consider a Steiner cut $\{\bar{W}, W\}$. Obviously $u_W = \max\{0, d'(v_h) - d'(v_i)\}$ for $h = \min\{j \mid v_j \notin W\}$, $i = \max\{j \mid v_j \in W\}$. If there are two vertices v_h and v_i with $h < i, v_h \notin W$, and $v_i \in W$,

then $u_W = 0$. So if $u_W > 0$, there must be a vertex v_k with $v_l \in W$ for all $l \leq k$ and $v_l \notin W$ for all $l > k$; so we can denote W by W_k: $W_k = \{v_1, \ldots, v_k\}$; $u_{W_k} = d'(v_{k+1}) - d'(v_k)$.
For any edge $[v_a, v_b]$ we have: $\sum_{W,[v_a,v_b]\in\delta^-(W)} u_W = \sum_{b\leq k<a} u_{W_k} = \sum_{b\leq k<a}(d'(v_{k+1}) - d'(v_k)) = \max\{0, d'(v_a) - d'(v_b)\} = \max\{0, \min\{d(v_a, z_t), d(z_1, z_t)\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} \leq \max\{0, \min\{c_{ab} + d(v_b, z_t), d(z_1, z_t) + c_{ab}\} - \min\{d(v_b, z_t), d(z_1, z_t)\}\} = c_{ab}$. □

It follows immediately that u is feasible for DLP_C.
Since the dual variables u are not used explicitly in the reduction process, it is sufficient to work with the reduced costs and the calculated lower bound; so the updating process for one terminal can be performed very fast, because we just need a shortest paths tree rooted at z_t which spans z_1. Then the reduced costs for an edge $[v_a, v_b]$ are decreased by $\max\{0, d'(v_a) - d'(v_b)\}$ and the lower bound is increased appropriately. After each such updating there may still be terminals that are not reachable from the root by edges of zero reduced cost, so the updating can be repeated with other terminals, but then with respect to the remaining reduced costs. We guide this calculation by the structure of a heuristic solution: The terminals are sorted according to non-decreasing distances from the root in this solution and considered one at a time.
Note that using this method, an edge can be visited by several terminals. To limit the effort, we simply abort the calculation of a shortest paths tree if it reaches a vertex which has already been visited by 5 terminals. This way, the running time for setting the lower bound and reduced costs is $O(e + n \log n)$. The other operations of the test can be performed in the same time, as described before. To construct a heuristic solution, we use a heuristic described in section 4.1, which has the same running time. So the whole test can be performed in total time $O(e + n \log n)$. We call this test **LDA** (Limited Dual Ascent). Despite its very small (worst case) time, it is fairly effective, especially if the proportion of terminals to all vertices is not very high.

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. Using a combination of DUAL-ASCENT and the Lagrangean relaxation of the multicommodity flow formulation (which was briefly described in section 2.2.2), in [20] we devised a reduction method using a sensitivity analysis on Lagrangean multipliers, which can also be used in combination with the row generating strategy. Although the resulting test is sometimes quite effective, its details are rather lengthy and technical, so we decided not to include it in this article, and consequently did not use it for the results reported here.

3.2.3 Using the Row Generation Strategy

Every iteration of the row generation method described in section 2.2.2 provides a dual feasible solution for LP_C (or LP_C plus the additional constraints (3.1)) and appropriate reduced costs. Using this information, the same reduction techniques as described in section 3.2.2 can be used. The only enhancement here is that edges are allowed to be deleted even in one direction temporarily. Note that this can amplify the effect of subsequent reductions considerably. In the linear program itself, the deletion of edges is realized by fixing the corresponding variables to zero.
In many cases the mentioned reductions during the row generation make further alternative-based reductions possible. But it would be a bad idea to delay these reductions until the row generation terminates, because they could possibly accelerate the computation and raise the optimal value of the relaxation. On the other hand, it would be problematic to abort the row generation, do the alternative-based reductions and then start it again, because the constraints generated in the meantime could not be used (directly) anymore. 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).

3.3 Integration and Implementation of Tests

To study the effect of different combinations and orderings of the tests, we designed an interpreter for command-lines, where each test is encoded by a character. We also implemented a direct control of loops (through parentheses), their termination criteria, switching of parameters, etc. The main observation is that the (alternative-based) tests are not very sensitive to the order in which they are executed. On the other hand, the ordering has often an impact on the total time for reductions; in this sense the ordering cited in [15] is a suitable one (although not necessarily the only one, as long as a fast version of PT_m is performed first).

For the implementation, we have chosen a kind of adjacency-list representation of networks (with all edges in a single array), but we sometimes switch to other auxiliary representations (all linear in 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 $O(e+n\log n)$ of the fast tests.

With the additional postulation 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 asymptotical time bound for the whole reduction process as for the first iteration ($O(e + n\log n)$, 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 $O(e)$. We always perform such a transformation after each run of the program, checking the feasibility and value of the solution in the original instance.

3.4 Empirical Results

In this subsection, we present some empirical results on the larger instances of the OR-Library for a packet of reduction techniques with the worst case time $O(e+n\log n)$, namely PT_m, NTD_k, NV, SL, VR and LDA. Using the same argument as in the previous subsection, the same time bound can be given for the whole reduction process. Of course even more reduction could be achieved using the other techniques in addition, especially those explicitly working with relaxations (like DA and RG), but such a packet would have rather the character of an exact algorithm (and actually it solves almost all instances of the OR-Library to optimality), so such results are reported in the section 5 (exact algorithms). The results given here should underline the applicability of reductions in fast heuristics, a subject which we will elaborate in the next section.

A stroke in the tables means that the instance has been solved to optimality by the reductions.

instance	original size			size after reductions				time
	n	r	e	n	r	e	remaining edges in %	(in sec.)
D1	1000	5	1250		—		0	0.1
D2	1000	10	1250		—		0	0.1
D3	1000	167	1250		—		0	0.1
D4	1000	250	1250		—		0	0.1
D5	1000	500	1250		—		0	0.1
D6	1000	5	2000		—		0	0.1
D7	1000	10	2000		—		0	0.1
D8	1000	167	2000	140	63	230	11.5	0.1
D9	1000	250	2000		—		0	0.1
D10	1000	500	2000		—		0	0.1
D11	1000	5	5000		—		0	0.1
D12	1000	10	5000		—		0	0.1
D13	1000	167	5000		—		0	0.1
D14	1000	250	5000		—		0	0.1
D15	1000	500	5000		—		0	0.1
D16	1000	5	25000		—		0	0.2
D17	1000	10	25000		—		0	0.3
D18	1000	167	25000	807	94	2430	9.7	0.3
D19	1000	250	25000	692	96	2017	8.1	0.4
D20	1000	500	25000		—		0	0.1
						average:	1.5	0.14

Table 1: Results of a fast reduction packet (OR-Library, D-instances)

instance	original size			size after reductions				time
	n	r	e	n	r	e	remaining edges in %	(in sec.)
E1	2500	5	3125		—		0	0.1
E2	2500	10	3125		—		0	0.1
E3	2500	417	3125	91	56	139	4.4	0.1
E4	2500	625	3125		—		0	0.1
E5	2500	1250	3125		—		0	0.3
E6	2500	5	5000		—		0	0.2
E7	2500	10	5000		—		0	0.6
E8	2500	417	5000	359	144	618	12.3	0.3
E9	2500	625	5000	150	82	234	4.7	0.2
E10	2500	1250	5000		—		0	0.5
E11	2500	5	12500		—		0	0.6
E12	2500	10	12500		—		0	0.7
E13	2500	417	12500	608	189	1078	8.6	0.6
E14	2500	625	12500		—		0	0.3
E15	2500	1250	12500		—		0	0.5
E16	2500	5	62500		—		0	0.9
E17	2500	10	62500		—		0	1.1
E18	2500	417	62500	2031	247	6028	9.6	1.1
E19	2500	625	62500	1110	171	2867	4.6	1.2
E20	2500	1250	62500		—		0	0.6
						average:	2.2	0.54

Table 2: Results of a fast reduction packet (OR-Library, E-instances)

4 Upper Bounds

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 which achieve good solution values only after long runs. In this section, we describe some of the methods we used in our attempt to achieve both goals simultaneously.

4.1 Path Heuristics

The repetitive shortest paths heuristics belong to the empirically most successful classical heuristics for the Steiner problem in networks ([15], [27], [25]). But naive implementation of these heuristics (simply starting SPH from scratch every time) leads to intolerable running times. So, as a first step, we designed an empirically efficient realization of such a heuristic and also a modified version which guarantees short running times. We contrived these variants only as components of our other algorithms, not as standalone heuristics.

Studying a repetitive shortest paths heuristic such as SPH-V [15] one readily observes that the actions can be divided into two phases (see also [8, 11]): In the first phase, one can compute shortest paths from each terminal to all vertices; this can be done e.g. in $O(r(e + n \log n))$. Using the information from the first phase, each run of the SPH in the second phase (constructing a Steiner tree by successively connecting the current tree (a single vertex at the beginning) to the closest terminal not in the tree by a shortest path) can easily be realized in time $O(rn)$. Our concern here is achieving further empirical acceleration.

With regard to the first phase, we observe that the shortest paths need not be always computed completely:

Observation 4.1 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 which is not longer than P. So, when computing shortest paths from z, we need not consider v and any vertex which 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'. $\square$

As a consequence, one can stop computing the shortest paths tree from a terminal z in the first phase as soon as the Voronoi region of z and the neighboring terminals (as defined in 1.1) have been spanned, because the shortest path between z and every vertex v visited afterwards contains a vertex $v' \in N(z')$ with z' a neighbor of z and $d(z, z') \leq d(z, v)$, since z' has already been spanned by the shortest paths tree. Furthermore, no shortest path via an intermediary terminal needs to be considered. These observations often lead to a considerable reduction in the empirical times, especially if the graph has many terminals and is not dense (the latter is almost always the case after reductions). Note that for graphs with few terminals, repetitive SPH is fast anyway.
For building the Steiner trees in the second phase, we prefer a realization which uses the concept of neighborhoods: Using the information from the first phase, we manage for each vertex v a list of neighboring terminals, sorted by (increasing) distances to v. A priority queue manages candidates for expansion of the tree, using the distance to the nearest terminal not in the tree as the key for insertion. Each time a vertex v is extracted from the queue, two cases can arise: Either the terminal corresponding to the key is not yet in the tree, in this case the tree is expanded by the corresponding shortest path (and the queue is updated); or it is already in the tree, in this case the neighbor list of v is scanned further until either a terminal not in the tree is visited (which delivers the key for reinsertion of v into the queue) or the end of the list is reached (meaning that v can be ignored). Although the worst case time of this implementation ($O(rn \log n)$) is slightly worse than $O(rn)$ of the straightforward implementations, it is usually much faster, and the worst case time is dominated by the first phase anyway.

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 $O(e+n \log n)$. Motivated by the fact that the for SPH relevant vicinity of each terminal often gets smaller with growing number of terminals, one can simply force the first phase not to perform more than $O(e + n \log n)$ operations. But then it is not guaranteed anymore that the relevant neighborhood of each terminal is really captured. To remedy this defect, we simultaneously use the graph G' of Mehlhorn's fast implementation of DNH [15, 19], 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 (say at most ten) of Steiner trees are computed in the second phase, using different terminals as starting points.
This heuristic can be implemented with time $O(e + n \log n)$ and guarantees a performance ratio of 2. Although it was designed only to be used as a component of other algorithms (especially in combination with reductions), it yields reasonable results even on its own: For the D-instances of the OR-Library, the average gap from optimum is just 1.6%, much better than the 5% of DNH. The average running time of 0.2 seconds for these instances shows that this improvement is not paid with long running times.

4.2 Heuristic Reductions

Working with reductions, one often gets the impression that some of the tests are too cautious. Sometimes one has nice ideas for strengthening a test, which turn out to be not universally valid. Of course even the strangest exception is enough to make a reduction test completely useless for (direct) integration into an exact algorithm. But with respect to heuristics, the situation is fundamentally different: Here a much stronger orientation towards the frequent case can be adopted.
The idea used here is to support the normal (exact) reduction tests through some heuristic ones. It must be emphasized that the goal is not reducing the graphs by brute force, but only giving an impulse in situations where the exact reduction process is blocked, in order to activate it again. In this context, it is particularly advantageous if it can be assumed that the performed actions 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. This test is kept very cautious to

make a comprehensible proof possible. Furthermore, one observes readily that in case the used upper bound is not optimal, the test could potentially perform more (exact) reductions if a better upper bound were 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 asymptotical time for all iterations together is the same as for the first one, namely $O(e + n \log n)$. 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 $T'_D(R)$ yields also as a side effect a guaranteed performance ratio of 2. We call this whole procedure **PRUNE**.
The idea of not eliminating the nodes of a Steiner tree can be further utilized by using a (good) heuristic solution instead of $T'_D(R)$ for guiding the heuristic reductions. We use the implementation of SPH-V described in section 4.1 (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**.

4.3 Relaxations and Upper Bounds

Computing lower bounds is not the only motivation for dealing with relaxations; the gained information can also be used (among other things) to obtain upper bounds.
Consider the (directed) cut formulation P_C of the Steiner problem: Given an optimal solution $\hat{x}$ of its linear relaxation LP_C, the complementary slackness conditions state that each edge $[v_i, v_j]$ with $\hat{x}_{ij} > 0$ has zero reduced cost. Assuming that there is some similarity between some optimal solutions of the integer program P_C and its linear relaxation LP_C, its thoroughly motivated to search an (optimal) solution in a subgraph containing the edges with reduced cost zero.
The algorithm DUAL-ASCENT, attempting to construct an optimal solution for DLP_C, adjusts the reduced costs favourably. 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 [28, 25]. 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 which 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, however, the PRUNE heuristics run extremely fast on the auxiliary graphs, so that this kind of computation of upper bounds should be performed after each call to DUAL-ASCENT.
Although the empirical solution quality of this heuristic is striking, 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 ob-

servation 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 mentioned cases, this approach often improves the solution value, leading frequently 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 $\tilde{c}^T\hat{x} \leq optimum - lower$ (see observation 3.9) strongly suggests that normally there can not be many edges with large reduced costs in an optimal solution. This motivates another heuristic, **SLACK-PRUNE**, which basically follows the same scheme as GUIDED-PRUNE, but uses the criterion of the test DA for eliminating vertices. The guiding solution is computed by a call of PRUNE in the auxiliary graph described above, since the needed 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 $O(e \cdot \min\{e, nr\})$. But in combination with reductions, the empirical times are much smaller than the above term could suggest.

Like in DUAL-ASCENT, dual feasible solutions and corresponding reduced costs for LP_C are calculated during the row generating algorithm (section 2.2.2). This information can be used to generate auxiliary graphs similar to those in ASCEND-AND-PRUNE. But in this case there are not necessarily paths with reduced cost zero from the root to all terminals. The auxiliary graph in this context contains all vertices with the property that there is a path from the root over this vertex to another terminal not longer (with respect to reduced costs) than the longest shortest path from the root to another terminal. This auxiliary graph can be used as in ASCEND-AND-PRUNE. A classical method for utilizing the information provided by linear relaxations is to use an ordinary heuristic in the original network with modified edge costs $c'_{ij} = c_{ij}(1 - x_{ij})$ (where x is the primal solution of the current linear program). But this is not a generally 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, standalone heuristics. But as a complement to the row generating strategy, they are frequently effective, especially in the advanced stages of the algorithm.

4.4 Combination of Steiner Trees

During the reduction process and especially while solving instances exactly, one usually gets several distinct heuristic solutions. In general, it is not the best idea to simply keep the best solution and forget the others. It is possible that solutions with a worse value are better locally, and one can try to keep the best part of each solution.
We have developed several techniques for realizing the idea above. One simple and effective way is to consider the graph consisting of the union of the edge sets of two (or more) Steiner trees. In this graph, one can call a (powerful) heuristic again or even try an exact solution. Such graphs have frequently several (nontrivial) biconnected components, which makes the (exact) solution considerably faster. Using such schemes, we frequently get improvements in solution values (as far as they were not optimal anyway). The instances generated through such combinations (in the following called combination-instances) are almost always solved to optimality through (fast) reductions, so that these improvements are gained at no significant extra cost.
For the results reported in this paper, 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.

4.5 Empirical Results

In this subsection, we present some empirical results for a selection of our heuristics on the large instances of the OR-Library. These are all heuristics with a worst case time describable by a polynomial of low order, as explained in the previous subsections. We leave it to the reader to compare the empirical running times and solution qualities given below to those of other heuristics in the literature (for good results see [8, 11, 12, 23]).
In the table 3, the average gap from optimum (in %) and the average running time (in seconds) are given.

algorithm	D-instances		E-instances	
	gap (%)	time (sec.)	gap (%)	time (sec.)
PRUNE	0.11	0.3	0.46	1.2
GUIDED-PRUNE	0.08	0.2	0.13	0.9
ASCEND-AND-PRUNE	0	0.2	0.07	0.6
SLACK-PRUNE	0	0.3	0	2.8

Table 3: Results of the PRUNE heuristics on the large instances of the OR-Library

5 An Exact Algorithm

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

5.1 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: The bound-based reductions depend on upper and lower bounds; and the computation of upper and lower bounds profits from reductions, both in terms of running time and quality of results. The idea behind reduction tests is also the central part of the reduction-based heuristics for computing upper bounds. Further we use the structure of heuristic solutions (corresponding to good upper bounds) to guide the computation of lower bounds; and the information gained during the computation of lower bounds is used to guide the computation of upper bounds. All in all, there is a mutual dependence between the three major components: reductions, upper bounds, and lower bounds. This is not a drawback, but an advantage: The scenario is that performing (alternative-based) reductions accelerates the computation of upper and lower bounds and enhances their qualities; the information gained during the computation of bounds is used to reduce the instance further (using bound-based reductions), and then the whole pattern repeats. We call this whole process the reduction process, beginning with fast reductions and switching to more and more powerful ones as the process advances. This strategy is not only a major reason for the short solution times our algorithm very often achieves, but also allows solving instances which 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 which otherwise could not be solved (without branching) using the same techniques for computing upper and lower bounds.
For the empirical results given in this paper, we use the following components: For computing lower bounds, we use the relaxation LP_C (see section 2.1) through the algorithm DUAL-ASCENT and, in advanced stages, row generation (section 2.2.2) or, if the proportion of terminals to all vertices is high, the Lagrangean relaxation LaP_{T_0} (section 2.2.1). To reduce the instances, we use all described

alternative-based techniques (section 3.1). Besides, we use the bound-based techniques DA (section 3.2.2) and, in combination with row-generation, the test RG (section 3.2.3). For computing upper bounds we use our PRUNE heuristics (sections 4.2, 4.3), including the combination of Steiner trees (section 4.4). 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.

5.2 Branch-and-Bound

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

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

As the branching variable, we choose the non-terminal with the largest degree in the best available Steiner tree. The intuitive motivation for this choice is an intensification of the search in an area where a good solution has been found (in case of inclusion) and a diversification of the search to other areas (in case of exclusion). This strategy also supports the building of several blocks (biconnected components). It is known [15] that in case several blocks exist, the problem can be solved by solving the instances corresponding to each intermediate block separately, which generally reduces the total running time substantially. Although it usually cannot be assumed that the original instance is not biconnected, this often changes later during the reduction process and after branching. We use this fact frequently in our algorithms: Whenever a more time-consuming part is to be performed, we check whether the graph is biconnected. If this is not the case, we solve the corresponding subinstances separately and transform the gained information back to information for the original instance. Here one can use the following observation to identify the blocks which must be considered:

Observation 5.1 Let T be a Steiner tree with all leaves being terminals in a network G. A block of G is intermediate if and only if it contains an edge of T.

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

5.3 Empirical Results

Here we report on what the already presented components achieve together, acting as an "orchestra". As stated in section 1.2, results for different types of instances from the benchmark SteinLib are presented, including the instances of the OR-Library. All results are produced by a single run of the

same program with the same parameter values. For each instance, we give the number of nodes in the branch-and-bound tree (B) and the total time till the exact solution of the instance (see tables 5-9 on pages 24-25). We set a time limit of one hour on each run. Within this time, we have solved almost all considered instances, including many which (to our knowledge) have not been solved before. Indeed, the largest time our program needed for a previously solved instance in these sets has been 74 seconds (for E18 of the OR-Library). Only six instances have not been solved within one hour (see table 9); for these instances we give the gap (in percent) between the upper and the lower bound after one hour. Two of them could be solved allowing longer runs; for them we give the time for the exact solution in brackets (although optimal Steiner trees were found already in less than one hour). The other four could not be solved even in one day.
Again, 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 [3, 5, 7, 8, 17]). As an orientation, we compare the average times (in seconds) of this algorithm for the exact solution of the OR-Library instances (these are the only instances used by the majority of the authors) to those of other exact algorithms in the literature (table 4). Note that the differences in the speed of the used computers almost vanish when compared to the differences between the running times of our algorithm and the other ones.

instance-group	[3] Cray X-MP	[5] SG Indigo	[7] VAX 8700	[8] i486	[17] Sun Sparc 20	here Pentium-II
C	67	2946	3066	16	16	0.2
D	556	3545	14260	176	117	0.3
E*	—	—	31504	—	1020	1.4

Table 4: Results of different exact algorithms on the instances of the OR-Library

6 Concluding Remarks

We have presented several algorithmic contributions for solving the Steiner problem in networks. The empirical results strongly recommend the chosen approach based on reductions and underline the utility of the techniques presented in this paper. In particular, the reduction-based heuristics have proven to be extremely strong and robust. Also, the running times of the exact algorithm are often surprisingly small; and for many instances, there is not much room left for improvements. But this is not always the case:
On some instances, fast reductions come to a halt at a time when the used relaxations are still not strong enough; this is the case for some of the mc-instances (table 6 on page 24), where the algorithm has gone into branching to solve the instances exactly. But the results on the other groups of instances seem to indicate that such cases rarely arise naturally.
And there are of course the very large instances, like those VLSI-instances with more than 30000 vertices (see table 9 on page 25). Even for these instances, the methods in this paper are capable of producing fairly good results quite quickly (gaps of 2-4% between upper and lower bounds in 1-3 minutes). But if such instances have to be solved exactly, methods like row generation come to their current limits, because even linear programs with a number of variables or constraints linear in the number of vertices seem to be too large then to allow small running times. A natural approach for a faster utilization of relaxations would be improving DUAL-ASCENT further; but our investigations indicate that not much further progress is possible using heuristical criteria, and some basically new ideas have to be developed.
In the short term, considerable empirical progress (at least for some groups of instances) could be probably achieved by developing further reduction techniques. For some VLSI-instances, an approach like that used in [26] for Euclidean and rectilinear Steiner problems can prove to be fruitful.

Acknowledgement: We would like to thank the referees for their comments.

*Excluding E18. This instance has been only solved by [17] (68000 seconds) and by us (74 seconds).

instance	size			opt.	B	time
	n	r	e			(in sec.)
C01	500	5	625	85	1	0.01
C02	500	10	625	144	1	0.01
C03	500	83	625	754	1	0.01
C04	500	125	625	1079	1	0.02
C05	500	250	625	1579	1	0.02
C06	500	5	1000	55	1	0.03
C07	500	10	1000	102	1	0.03
C08	500	83	1000	509	1	0.04
C09	500	125	1000	707	1	0.06
C10	500	250	1000	1093	1	0.03
C11	500	5	2500	32	1	0.06
C12	500	10	2500	46	1	0.04
C13	500	83	2500	258	1	0.10
C14	500	125	2500	323	1	0.04
C15	500	250	2500	556	1	0.03
C16	500	5	12500	11	1	0.10
C17	500	10	12500	18	1	0.08
C18	500	83	12500	113	1	2.99
C19	500	125	12500	146	1	0.16
C20	500	250	12500	267	1	0.06
D01	1000	5	1250	106	1	0.03
D02	1000	10	1250	220	1	0.02
D03	1000	167	1250	1565	1	0.04
D04	1000	250	1250	1935	1	0.06
D05	1000	500	1250	3250	1	0.06
D06	1000	5	2000	67	1	0.08
D07	1000	10	2000	103	1	0.05
D08	1000	167	2000	1072	1	0.19
D09	1000	250	2000	1448	1	0.12
D10	1000	500	2000	2110	1	0.12
D11	1000	5	5000	29	1	0.08
D12	1000	10	5000	42	1	0.08
D13	1000	167	5000	500	1	0.14
D14	1000	250	5000	667	1	0.13
D15	1000	500	5000	1116	1	0.11
D16	1000	5	25000	13	1	0.26
D17	1000	10	25000	23	1	0.21
D18	1000	167	25000	223	1	1.90
D19	1000	250	25000	310	1	1.55
D20	1000	500	25000	537	1	0.18
E01	2500	5	3125	111	1	0.07
E02	2500	10	3125	214	1	0.07
E03	2500	417	3125	4013	1	0.26
E04	2500	625	3125	5101	1	0.24
E05	2500	1250	3125	8128	1	0.45
E06	2500	5	5000	73	1	0.17
E07	2500	10	5000	145	1	0.16
E08	2500	417	5000	2640	1	0.53
E09	2500	625	5000	3604	1	0.51
E10	2500	1250	5000	5600	1	0.71
E11	2500	5	12500	34	1	0.25
E12	2500	10	12500	67	1	0.61
E13	2500	417	12500	1280	1	4.34
E14	2500	625	12500	1732	1	0.80
E15	2500	1250	12500	2784	1	0.68
E16	2500	5	62500	15	1	0.71
E17	2500	10	62500	25	1	0.64
E18	2500	417	62500	564	1	74.1
E19	2500	625	62500	758	1	14.0
E20	2500	1250	62500	1342	1	0.71

Table 5: Instances of the OR-Library

instance	size			opt.	B	time
	n	r	e			(in sec.)
p401	100	5	4950	155	1	0.03
p402	100	5	4950	116	1	0.02
p403	100	5	4950	179	1	0.03
p404	100	10	4950	270	1	0.01
p405	100	10	4950	270	1	0.03
p406	100	10	4950	290	1	0.02
p407	100	20	4950	590	1	0.04
p408	100	20	4950	542	1	0.03
p409	100	50	4950	963	1	0.02
p410	100	50	4950	1010	1	0.01
p455	100	5	4950	1138	1	0.06
p456	100	5	4950	1228	1	0.13
p457	100	10	4950	1609	1	0.03
p458	100	10	4950	1868	1	0.05
p459	100	20	4950	2345	1	0.04
p460	100	20	4950	2959	1	0.05
p461	100	50	4950	4474	1	0.04
p601	100	5	180	10230	1	0.02
p602	100	5	180	8083	1	0.01
p603	100	5	180	5022	1	0.01
p604	100	10	180	11397	1	0.01
p605	100	10	180	10355	1	0.01
p606	100	10	180	13048	1	0.01
p607	100	20	180	15358	1	0.01
p608	100	20	180	14439	1	0.01
p609	100	20	180	18263	1	0.01
p610	100	50	180	30161	1	0.01
p611	100	50	180	26903	1	0.01
p612	100	50	180	30258	1	0.01
p613	200	10	370	18429	1	0.03
p614	200	20	370	27276	1	0.03
p615	200	40	370	42474	1	0.03
p616	200	100	370	62263	1	0.02
p619	100	5	180	7485	1	0.01
p620	100	5	180	8746	1	0.01
p621	100	5	180	8688	1	0.01
p622	100	10	180	15972	1	0.01
p623	100	10	180	19496	1	0.02
p624	100	20	180	20246	1	0.01
p625	100	20	180	23078	1	0.02
p626	100	20	180	22346	1	0.02
p627	100	50	180	40647	1	0.01
p628	100	50	180	40008	1	0.01
p629	100	50	180	43287	1	0.01
p630	200	10	370	26125	1	0.01
p631	200	20	370	39067	1	0.02
p632	200	40	370	56217	1	0.75
p633	200	100	370	86268	1	0.01

Table 7: SteinLib, p-instances

instance	size			opt.	B	time
	n	r	e			(in sec.)
mc2	120	60	7140	71	23	8.89
mc3	97	45	4656	47	25	16.2
mc7	400	170	760	3417	1	0.05
mc8	400	188	760	1566	1	0.10
mc11	400	213	760	11689	1	0.04
mc13	150	80	11175	92	8	11.7

Table 6: SteinLib, mc-instances

instance	size			opt.	B	time
	n	r	e			(in sec.)
berlin52	52	16	1326	1044	1	0.02
brasil58	58	25	1653	13655	1	0.01
world666	666	174	221445	122467	1	1.46

Table 8: SteinLib, complete Euclidean

instance	size			opt.	B	time
	n	r	e			(in sec.)
diw0234	5349	25	10086	1996	1	1.57
diw0250	353	11	608	350	1	0.01
diw0260	539	12	985	468	1	0.02
diw0313	468	14	822	397	1	0.01
diw0393	212	11	381	302	1	0.01
diw0445	1804	33	3311	1363	1	0.29
diw0459	3636	25	6789	1362	1	0.99
diw0460	339	13	579	345	1	0.03
diw0473	2213	25	4135	1098	1	0.30
diw0487	2414	25	4386	1424	1	0.29
diw0495	938	10	1655	616	1	0.06
diw0513	918	10	1684	604	1	0.08
diw0523	1080	10	2015	561	1	0.04
diw0540	286	10	465	374	1	0.01
diw0559	3738	18	7013	1570	1	5.35
diw0778	7231	24	13727	2173	1	3.17
diw0779	11821	50	22516	4440	1	1298
diw0795	3221	10	5938	1550	1	0.90
diw0801	3023	10	5575	1587	1	0.59
diw0819	10553	32	20066	3399	1	2.56
diw0820	11749	37	22384	4167	1	159

instance	size			opt.	B	time
	n	r	e			(in sec.)
gap1307	342	17	552	549	1	0.03
gap1413	541	10	906	457	1	0.04
gap1500	220	17	374	254	1	0.02
gap1810	429	17	702	482	1	0.06
gap1904	735	21	1256	763	1	0.11
gap2007	2039	17	3548	1104	1	1.26
gap2119	1724	29	2975	1244	1	0.24
gap2740	1196	14	2084	745	1	0.13
gap2800	386	12	653	386	1	0.03
gap2975	179	10	293	245	1	0.01
gap3036	346	13	583	457	1	0.11
gap3100	921	11	1558	640	1	0.09
gap3128	10393	104	18043	4292	1	23.3

instance	size			opt.	B	time
	n	r	e			(in sec.)
dmxa0296	233	12	386	344	1	0.02
dmxa0368	2050	18	3676	1017	1	1.12
dmxa0454	1848	16	3286	914	1	0.24
dmxa0628	169	10	280	275	1	0.02
dmxa0734	663	11	1154	506	1	0.06
dmxa0848	499	16	861	594	1	0.06
dmxa0903	632	10	1087	580	1	0.92
dmxa1010	3983	23	7108	1488	1	1.19
dmxa1109	343	17	559	454	1	0.04
dmxa1200	770	21	1383	750	1	0.29
dmxa1304	298	10	503	311	1	0.03
dmxa1516	720	11	1269	508	1	0.08
dmxa1721	1005	18	1731	780	1	0.15
dmxa1801	2333	17	4137	1365	1	0.85

instance	size			opt.	B	time
	n	r	e			(in sec.)
alue2087	1244	34	1971	1049	1	0.18
alue2105	1220	34	1858	1032	1	0.26
alue3146	3626	64	5869	2240	1	22.5
alue5067	3524	68	5560	2586	1	10.3
alue5345	5179	68	8165	3507	1	1136
alue5623	4472	68	6938	3413	1	1298
alue5901	11543	68	18429	3912	1	653
alue6179	3372	67	5213	2452	1	4.17
alue6457	3932	68	6137	3057	1	4.49
alue6735	4119	68	6696	2696	1	18.8
alue6951	2818	67	4419	2386	1	22.1
alue7065	34046	544	54841	$\leq$23905	1	*2.4%*
alue7066	6405	16	10454	2256	1	1666
alue7080	34479	2344	55494	$\leq$62553	1	*1.9%* §
alue7229	940	34	1474	824	1	0.10

instance	size			opt.	B	time
	n	r	e			(in sec.)
msm0580	338	11	541	467	1	0.08
msm0654	1290	10	2270	823	1	0.16
msm0709	1442	16	2403	884	1	0.18
msm0920	752	26	1264	806	1	0.32
msm1008	402	11	695	494	1	0.17
msm1234	933	13	1632	550	1	0.10
msm1477	1199	31	2078	1068	1	0.20
msm1707	278	11	478	564	1	0.02
msm1844	90	10	135	188	1	0.01
msm1931	875	10	1522	604	1	0.10
msm2000	898	10	1562	594	1	0.09
msm2152	2132	37	3702	1590	1	0.55
msm2326	418	14	723	399	1	0.04
msm2492	4045	12	7094	1459	1	3.24
msm2525	3031	12	5239	1290	1	0.47
msm2601	2961	16	5100	1440	1	1.69
msm2705	1359	13	2458	714	1	0.14
msm2802	1709	18	2963	926	1	0.24
msm2846	3263	89	5783	3135	1	292
msm3277	1704	12	2991	869	1	0.27
msm3676	957	10	1554	607	1	0.11
msm3727	4640	21	8255	1376	1	1.02
msm3829	4221	12	7255	1571	1	4.56
msm4038	237	11	390	353	1	0.03
msm4114	402	16	690	393	1	0.04
msm4190	391	16	666	381	1	0.03
msm4224	191	11	302	311	1	0.01
msm4312	5181	10	8893	2016	1	10.6
msm4414	317	11	476	408	1	0.03
msm4515	777	13	1358	630	1	0.10

instance	size			opt.	B	time
	n	r	e			(in sec.)
alut0787	1160	34	2089	982	1	0.11
alut0805	966	34	1666	958	1	1.79
alut1181	3041	64	5693	2353	1	358
alut2010	6104	68	11011	3307	1	29.1
alut2288	9070	68	16595	3843	1	1078
alut2566	5021	68	9055	3073	1	604
alut2610	33901	204	62816	$\leq$12280	1	*3.8%*
alut2625	36711	879	68117	$\leq$35583	1	*3.7%* §
alut2764	387	34	626	640	1	0.01

instance	size			opt.	B	time
	n	r	e			(in sec.)
taq0014	6466	128	11046	5326	1	1556
taq0023	572	11	963	621	1	0.14
taq0365	4186	22	7074	1914	1	2.77
taq0377	6836	136	11715	6393	1	*0.9%* (6486)
taq0431	1128	13	1905	897	1	0.13
taq0631	609	10	932	581	1	0.07
taq0739	837	16	1438	848	1	0.53
taq0741	712	16	1217	847	1	0.41
taq0751	1051	16	1791	939	1	0.31
taq0891	331	10	560	319	1	0.01
taq0903	6163	130	10490	5099	1	*1.4%* (16056)
taq0910	310	17	514	370	1	0.02
taq0920	122	17	194	210	1	0.01
taq0978	777	10	1239	566	1	0.07

Table 9: SteinLib, VLSI-instances

§ Due to the memory requirements, we had to use a PC with 256 MB of main memory and a Pentium-II 450 MHz processor for these two instances.

References

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

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

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

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

[5] J. E. Beasley and A. Lucena. A branch and cut algorithm for the Steiner problem in graphs. *Networks*, 31:39–59, 1998.

[6] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[7] 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.

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

[9] C. W. Duin and T. Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17:353–364, 1987.

[10] C. W. Duin and T. Volgenant. Reduction tests for the Steiner problem in graphs. *Networks*, 19:549–567, 1989.

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

[12] H. Esbensen. Computing near-optimal solutions to the Steiner problem in a graph using a genetic algorithm. *Networks*, 26:173–185, 1995.

[13] M. X. Goemans and D. J. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Mathematical Programming*, 60:145–166, 1993.

[14] M. X. Goemans and Y. Myung. A catalog of Steiner tree formulations. *Networks*, 23:19–28, 1993.

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

[16] T. Koch and A. Martin. SteinLib. `ftp://ftp.zib.de/pub/Packages/mp-testdata/steinlib/index.html`, 1997.

[17] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[18] T. L. Magnanti and L. A. Wolsey. Optimal Trees. In M. O. Ball et al., editor, *Handbooks in Operations Research and Management Science*, volume 7, chapter 9. Elsevier Science, 1995.

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

[20] T. Polzin and S. Vahdati Daneshmand. Algorithmen für das Steiner-Problem. Master's thesis, Universität Dortmund, 1997.

[21] T. Polzin and S. Vahdati Daneshmand. A comparison of Steiner tree relaxations. Technical Report 5/1998, Universität Mannheim, 1998. (to appear in Discrete Applied Mathematics).

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

[23] M. G. A. Verhoeven. *Parallel Local Search*. PhD thesis, Eindhoven University of Technology, 1996.

[24] 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.

[25] S. Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40:45–72, 1992.

[26] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. Technical Report 98/11, Dept. of Computer Science, University of Copenhagen, 1998.

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

[28] R. T. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287, 1984.