Alcom-FT Technical Report Series
ALCOMFT-TR-02-38

Partitioning Techniques for the Steiner Problem*

Tobias Polzin
Max-Planck-Institute,
Saarbrücken, Germany
polzin@mpi-sb.mpg.de

Siavash Vahdati
University of Mannheim,
Germany
vahdati@informatik.uni-mannheim.de

Abstract

Partitioning is one of the basic ideas for designing efficient algorithms, but on $\mathcal{NP}$-hard problems like the Steiner problem, straightforward application of the classical paradigms for exploiting this idea rarely leads to empirically successful algorithms. In this paper, we present a new approach that is based on vertex separators. We show several contexts in which this approach can be used profitably. Our approach is new in the sense that it uses partitioning to design reduction methods. We introduce two such methods and show their impact empirically. These methods are integrated into a program package, which presently achieves by far the best results on a wide range of benchmark instances of the Steiner problem in networks.

Keywords: Steiner problem; Reduction methods; Partitioning; Vertex separators

1 Introduction

The Steiner problem is the problem of connecting a set of terminals (vertices in a weighted graph or points in some metric space) at minimum cost. This is a classical $\mathcal{NP}$-hard problem [6] with many important applications in network design in general and VLSI design in particular. Background information on this problem can be found in [5].

Over the years, more and more sophisticated (exact) algorithms have been developed for classical $\mathcal{NP}$-hard problems like the Steiner problem (see for example [10]). The empirical success of these algorithms is mainly due to the interaction of many different components, playing together as an orchestra. In this paper, we focus on one of these components, which uses the idea of partitioning. Although the basic ideas behind the presented approach are relatively simple, the way we exploit them is new, and our approach could also be useful for other problems.

After some preliminaries in Section 1.1, we motivate the use of partitioning and point out some difficulties in applying it in Section 1.2. In Section 2, we describe an approach that uses vertex separators. We will show examples where this method is helpful. Then we show how to find the necessary separators. We describe how to use these separators in reduction methods, i.e., for transforming instances into smaller or easier ones. Finally, in Section 3, we look at the empirical impact of the presented methods by performing tests on some VLSI and geometric instances from the library SteinLib.

*This work was partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

1.1 Definitions and Basics

The Steiner problem in networks can be stated as follows (see [5] for details): Given an (undirected, connected) network $G = (V, E, c)$ (with vertices $V = \{v_1, \ldots, v_n\}$, edges E and edge weights $c_{ij} > 0$ for all $(v_i, v_j) \in E$) and a set R, $\emptyset \neq R \subseteq V$, of **required vertices** (or **terminals**), find a minimum weight tree in G that spans R (called a **Steiner minimal tree**). If we want to stress that v_i is a terminal, we will write z_i instead of v_i. A non-terminal in a Steiner tree is called a **Steiner node**. A **key node** in a Steiner tree is a node that is either a terminal or a Steiner node of degree at least 3. A **key path** is a path in a Steiner tree in which (only) the endpoints are key nodes.
A **bottleneck** of a path P_{ij} between two vertices v_i and v_j is a longest edge on P_{ij}. The **bottleneck distance** b_{ij} between v_i and v_j is the minimum bottleneck length over all paths P_{ij}. Similarly, a **Steiner bottleneck** of a path P_{ij} is a longest subpath with (only) endpoints in $R \cup \{v_i, v_j\}$; and the **Steiner bottleneck distance** s_{ij} between v_i and v_j is the minimum Steiner bottleneck length over all P_{ij}. One of the fundamental observations for designing reduction techniques for the Steiner problem is that any edge (v_i, v_j) with $c_{ij} > s_{ij}$ can be removed without changing the optimum solution value [2]. (This can be extended to the case of equality, if a path corresponding to s_{ij} does not contain (v_i, v_j).) We will also use a consequence of this observation: The optimum solution value does not change by inserting any edges (v_i, v_j) of length s_{ij} (in case of parallel edges, only one with minimum weight is kept). For any subset $S \subseteq R$, all b_{ij}, s_{ij} with $v_i, v_j \in S$ can be computed in time $O(|E| + |V| \log |V| + |S|^2)$ [1].

1.2 Partitioning: Advantages and Difficulties

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

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

Effectiveness: Sometimes, a method (a component of an exact algorithm) works like a charm on some group of instances; but it fails on larger instances of the same type. Methods that are based on LP-relaxations of the problem are good examples, because any LP-relaxation of polynomial size is bound to have some (integrality) errors in this context. In larger instances, such errors can accumulate and become more and more relevant. Partitioning can help against this accumulation of errors, as we will show in Section 2.3.2.

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

However, for applying the idea of partitioning to problems like the Steiner problem, classical approaches are not very helpful. Divide-and-conquer techniques are not generally applicable, because one usually cannot find independent subproblems. Dynamic programming techniques can indeed be applied, but these techniques (at least in their classical form) do not lead to empirically efficient algorithms.
In this paper, we introduce the new approach of using partitioning to design reduction methods. We partition instances by finding (small) terminal separators, i.e., vertex separators that consist of only terminals. This allows us to keep the dependence between the resulting subinstances manageable.

2 Partitioning on the Basis of Terminal Separators

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

2.1 Application Examples

Reduced Instances:

There are several reduction methods which, when successful, tend to transform instances without useful terminal separators into instances with them (see [9, 10]).
Figure 1 shows a VLSI-instance from the library SteinLib [7]. The terminals (black squares) are to be connected on the grid. Note that there are holes in the grid (corresponding to obstacles on the chip), so such instances are not geometric (rectilinear). In this figure, one does not detect any useful terminal separator. But the situation changes after applying some reduction methods: Figure 2 shows the reduced instance, which is produced after a couple of seconds: the black edges are chosen (and contracted); the grey edges remain as the reduced instance; this reduced instance is redrawn more compactly in Figure 3. Note that the visualization used is not geometric; edges which appear relatively long may actually have relatively small cost. (Our algorithm is a pure graph algorithm, which does not use the coordinates of the points anyway.) In this reduced instance, one easily detects many small terminal separators and corresponding components.

Geometric Steiner Problems

For geometric Steiner problems, an approach based on full Steiner trees has been empirically successful [14]. In geometric Steiner problems, a set of points (in the plane) is to be connected at minimum cost according to some geometric distance metric. The resulting interconnection, a Steiner minimal tree (SMT), can be decomposed into so-called FSTs (full Steiner trees) by splitting its inner terminals. The FST-approach consists of two phases. In the first phase, the FST generation phase, a set of FSTs is generated that is guaranteed to contain an SMT. In the second phase, the FST concatenation phase, one chooses a subset of the generated

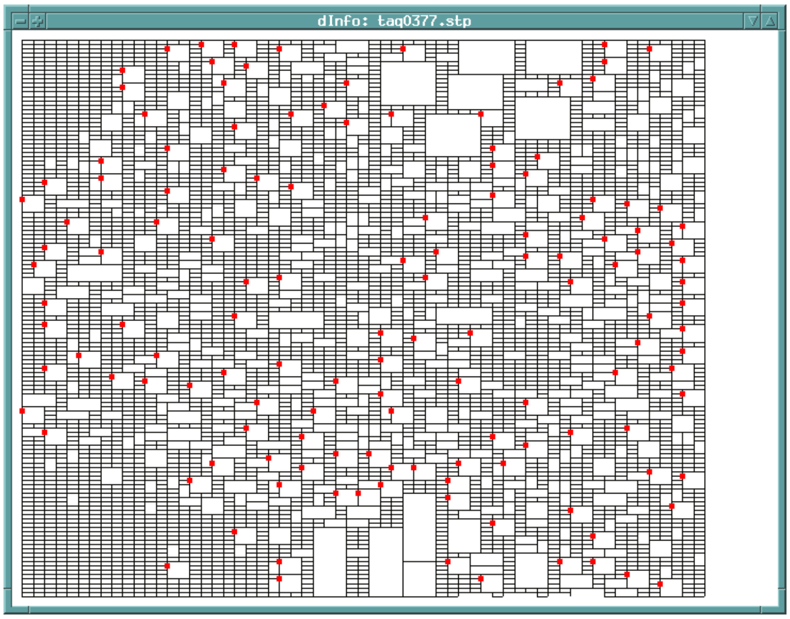

Figure 1: VLSI-instance taq0377 ($|V| = 6836,\ |E| = 11715,\ |R| = 136$)

FSTs whose concatenation yields an SMT. Although there are point sets that give rise to an exponential number of FSTs in the first phase, usually only a linear number (in $|R|$) of FSTs are generated, and empirically the bottleneck of this approach has usually been the second phase, where originally methods like backtracking or dynamic programming have been used. A breakthrough occurred as Warme [13] used the fact that FST concatenation can be reduced to finding a minimum spanning tree in a hypergraph whose vertices are the terminals and whose (hyper-)edges correspond to the generated FSTs. Although the minimum spanning tree in hypergraph (MSTH) problem is $\mathcal{NP}$-hard, a branch-and-cut approach based on the linear relaxation of an integer programming formulation of this problem has been empirically successful.

Note that by building the union of (the edge sets of) the FSTs generated in the first phase, we get a normal graph and the FST-concatenation problem is reduced to solving the classical Steiner problem in this graph. In [11], we report that our general graph algorithm often performs the second phase significantly faster than the MSTH-based method, especially for the more time-consuming instances. This is mainly due to the use of reduction techniques, where the methods presented here also play an important role, because usually the instances have many useful terminal separators with corresponding small components already at the beginning of the second phase, and this is even amplified after application of other reduction methods.

Combination of Steiner Trees

During the solution process, our program generates many heuristical solutions (Steiner trees). It would not be the best idea to just keep the best of them and throw the others away, because one could possibly combine parts of them to construct a new, even better solution. A simple

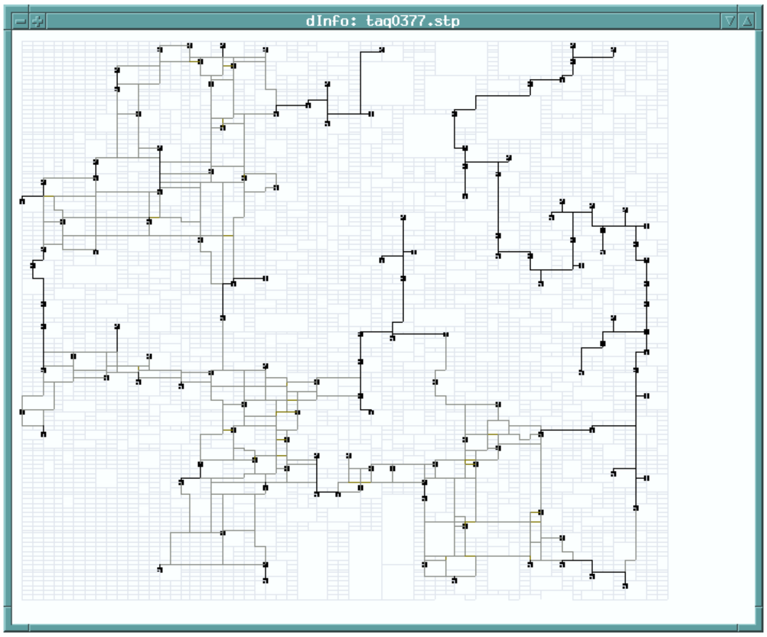

Figure 2: taq0377, reduced ($|V| = 193,\ |E| = 312,\ |R| = 67$)

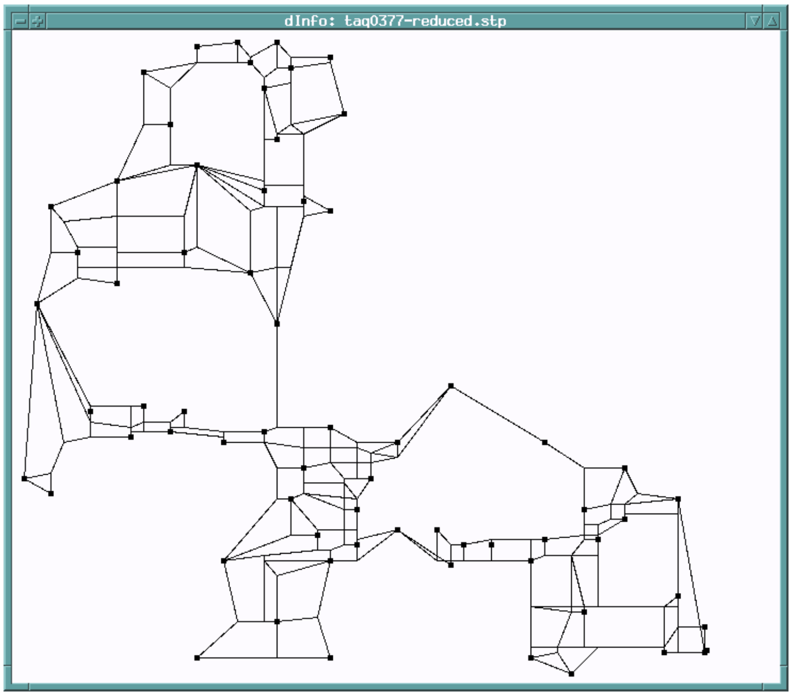

Figure 3: taq0377, reduced, redrawn ($|V| = 193,\ |E| = 312,\ |R| = 67$)

but effective method is to build the instance corresponding to the union of the edges of different Steiner trees and find a (heuristical or even exact) solution in that instance. Such instances often have small terminal separators, so that the methods described here can be applied.

2.2 Finding Terminal Separators

It is well known that the vertex connectivity problem can be solved by network flow techniques in the so-called split graph, which is generated by splitting each vertex into two vertices and connecting them by edges of low capacity; original edges have high (infinite) capacity. In this way, k-connectedness (finding a vertex separator of size less than k or verifying that no such separator exists) can be decided in time $O(\min\{k|V||E|, (k^3 + |V|)|E|\})$ [3] (this bound comes from a combination of augmenting path and preflow-push methods). In case of undirected graphs (as in the present application), the job can be done in a sparse graph with $O(k|V|)$ edges which can be constructed in time $O(|E|)$ [8], so $|E|$ can be replaced in the above bound by $k|V|$.

However, the application here is less general: we search for vertex separators consisting of terminals only, so only terminals need to be split. Besides, we are interested in only small separators, where k is a very small constant (usually less than 5, say at most 10), so we can concentrate on the (easier) augmenting flow methods. More importantly, we are not searching for a single separator of minimum size, but for many separators of small (not necessarily minimum) size. These observations have lead to the following implementation: we build the (modified) split graph (as described above), fix a terminal as source, and try different terminals as sinks, each time solving a minimum cut problem using augmenting path methods. In this way, up to $\Theta(|R|)$ terminal separators can be found in time $O(|R||E|)$. We accelerate the process by using some heuristics. A simple observation is that vertices that are reachable from the source by paths of non-terminals need not be considered as sinks. Similar arguments can be used to heuristically discard vertices that are reachable from already considered sinks by paths of non-terminals.

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

2.3 Reduction Methods

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

2.3.1 Case Differentiation

warm-up ($|S| = 1$): The case $|S| = 1$ corresponds to articulation points (and biconnected components), which can be found in linear time $O(|E|)$. It is well known [5] that the subinstances corresponding to the biconnected components can be solved independently. An example can be found in Figure 13: There is an articulation point in the middle; the upper and the lower components can be solved independently and the small component in the middle can obviously be discarded.

base case ($|S| = 2$): The case $|S| = 2$ corresponds to separation pairs (and triconnected components). All (non-trivial) triconnected components can be found in linear time $O(|E|)$ [4]. Consider Figure 4, where a separator S of size 2 and the corresponding components G_1 and G_2 are shown (black edges for G_2). Note that the two subinstances are not independent anymore. Now, for any Steiner minimal tree T, two cases are possible :

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

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

Since we do not know T in advance, for a direct solution we must also consider both cases for its complement G_1. But if G_2 is relatively small, the solution of the complementary subinstance can be almost as time-consuming as the solution of the original instance, meaning that not much is gained (or time may even be lost, because now we have to solve it twice). A classical approach would search for components of almost equal size, but we choose a different approach. The idea is to solve only the small component twice, and then take edges that are common to both solutions and discard edges that are included in neither. The result is shown in Figure 5, where only one edge from G_2 is left undecided (the other edges are either contracted (black) or deleted (light grey)). In Figure 6, the reduced instance is redrawn; as one sees, the processed component has almost disappeared.

general case ($|S| = k$): As described in Section 2.2, we can find up to $\Theta(|R|)$ separators of size at most k in time $O(k|R||E|)$. The basic approach is the same as for the case $|S| = 2$; but a larger number of cases must be considered now. We put each subset of terminals in S that are connected by one subtree of T in G_1 into one group. There can be $i = 1, \ldots, k$ such groups. For each i, we must count the number of ways of partitioning a set of k elements into i non-empty subsets, which is a Stirling number of the second kind $\left\{ {k \atop i} \right\}$. So there are $\sum_{i=1}^{k} \left\{ {k \atop i} \right\}$ cases.
Table 1 contains the concrete numbers for small k.

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

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

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

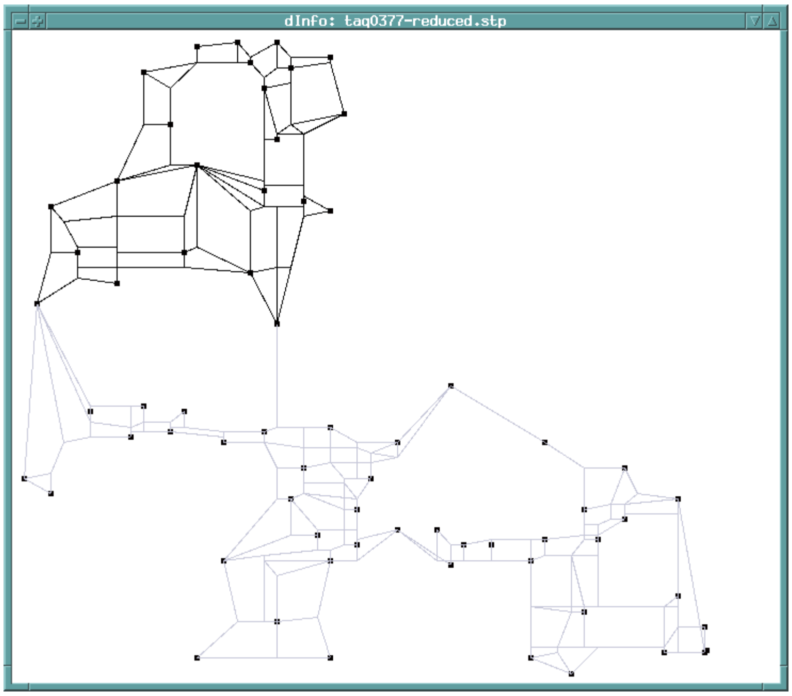

Figure 4: $|S| = 2$

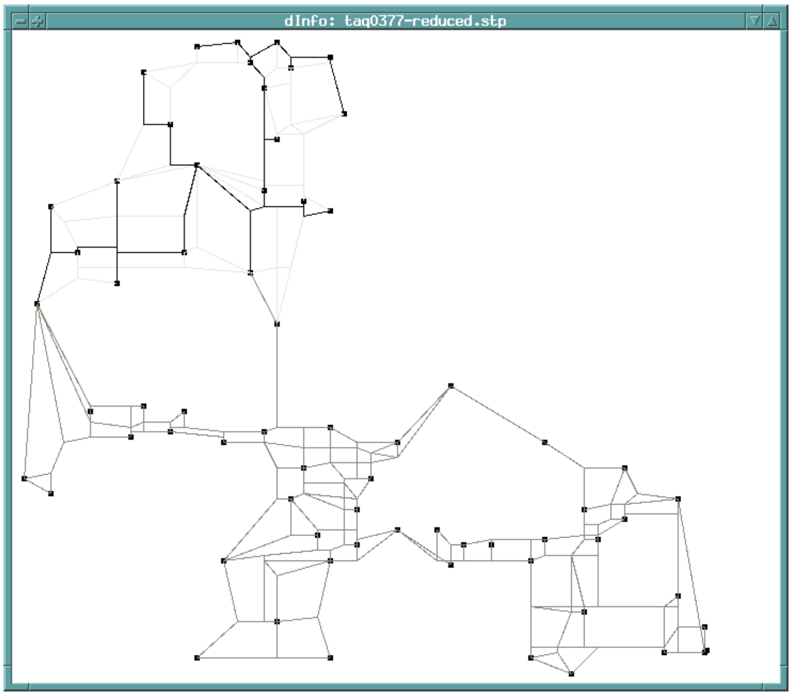

Figure 5: $(|V| = 133,\ |E| = 214,\ |R| = 45)$

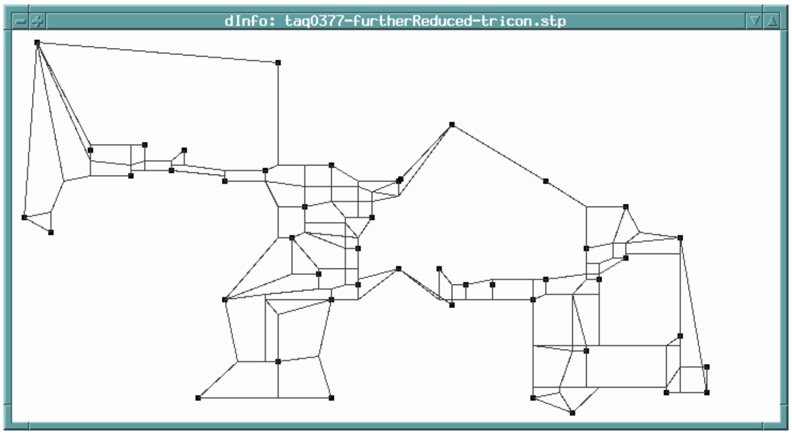

Figure 6: ($|V| = 133,\ |E| = 214,\ |R| = 45$)

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

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

Note also that for the cases in which we assume that z_i and z_j are connected in G_1, we do not merge z_i and z_j while solving the subinstance corresponding to G_2, but connect them with an edge of weight b^1_{ij}. In case this edge is not used in the solution of the subinstance, this can lead to more reductions.
This and similar observations can be used to rule out many cases in advance. Nevertheless, a question arises naturally: Can we find an alternative method that does not need explicit case differentiation? We will introduce such an alternative in the following section.

2.3.2 Local Bounds

The general principle of bound-based reduction methods [10] is to compute an upper bound *upper* and a lower bound under some constraint $lower_{con}$. The constraint cannot be satisfied by any optimal solution if $lower_{con} > upper$. The constraint is usually that the solution must contain some pattern (e.g., a vertex or an edge, or even more complex patterns like paths and trees [9]). But it is usually too costly to recompute a (strong) lower bound from scratch for each constraint. Here one can use an approach based on Linear Programming: Any linear relaxation can provide a dual feasible solution of value *lower* and reduced costs c'. We can use a fast method to compute a constrained lower bound $lower'$ with respect to c'. One easily observes [10] that $lower_{con} := lower + lower'$ is a lower bound for the value of any solution satisfying the constraint.
As an example for such a relaxation, consider the (directed) cut formulation P_C [15]: The Steiner problem is formulated as an integer program by introducing a binary x-variable for

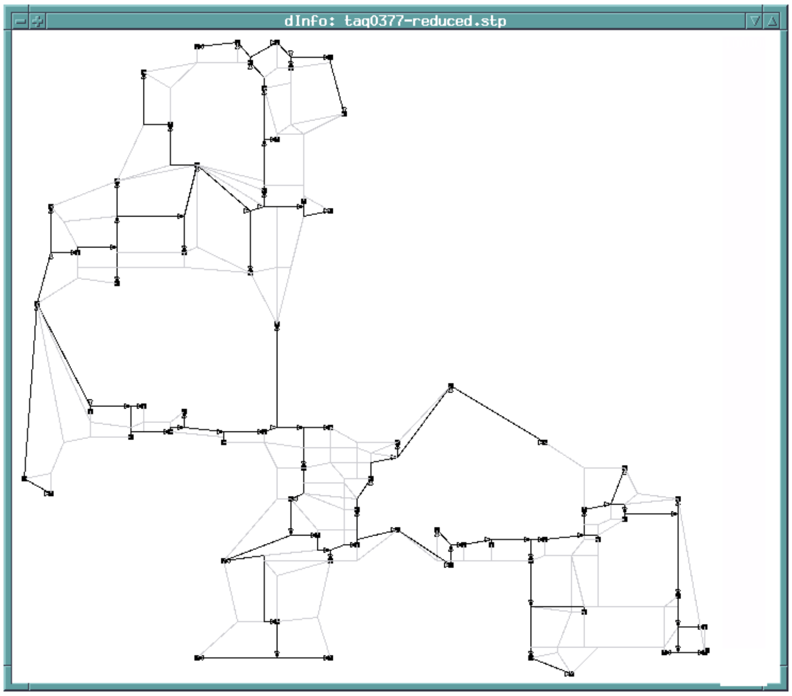

Figure 7: $v(P_C) = 6393$

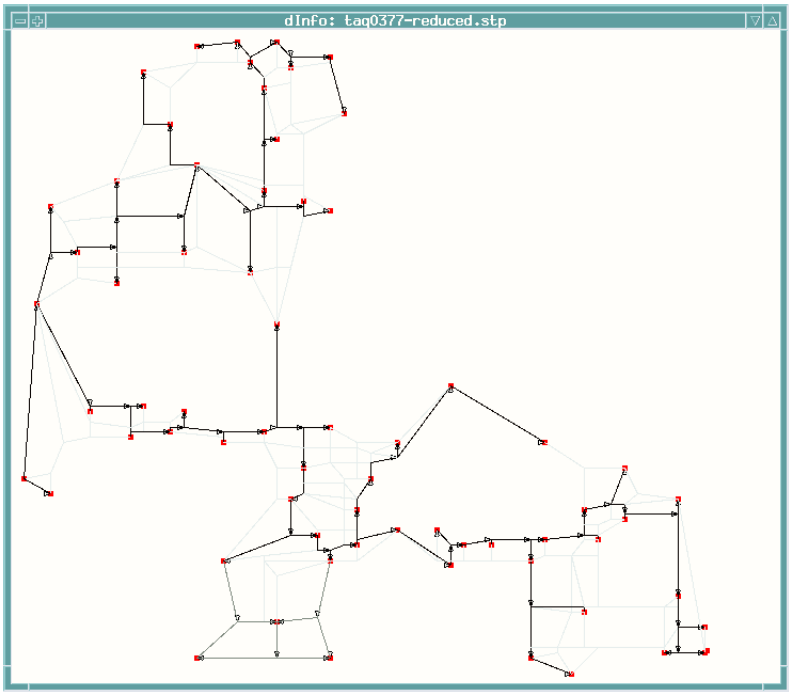

Figure 8: $v(LP_C) = 6392.5$

each arc (in case of undirected graphs, the bidirected counterpart is used, fixing a $z_1 \in R$ as the root). For each cut separating the root from another terminal, there is a constraint requiring that the sum of x-values of the cut arcs is at least 1. In this way, every feasible binary solution represents a feasible solution for the Steiner problem and each minimum solution (with value $v(P_C)$) a Steiner minimal tree (Figure 7). But in the optimal solution of the LP-relaxation LP_C variables can have fractional values. In Figure 8, the grey arcs have value 1/2 (the black arcs have value 1, the other arcs have value 0). One observes that a flow of one unit can still be sent from the root to each terminal, so all cut constraints are satisfied; and if the costs are adverse, an integrality gap can occur. This is in fact the case in this example, where the linear relaxation has optimum solution value $v(LP_C) = 6392.5$. As such gaps accumulate (e.g. in larger instances), the difference between the bounds grows, eventually causing the bound-based reductions to fail.
In the following, we show how to use locally computed bounds for bound-based reduction. This approach has two main advantages: The bounds can be computed faster; and there is less chance of accumulation of errors. The main difficulty is that the bounds must somehow take the dependence on the rest of the graph into account.

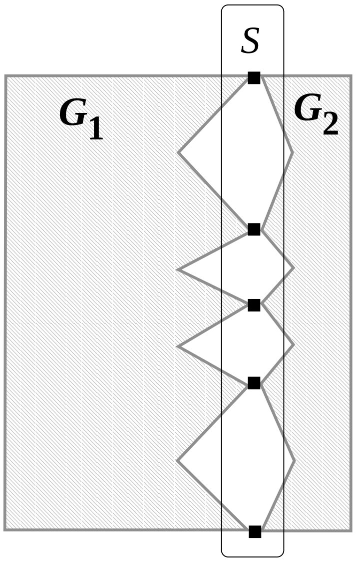

Figure 9: A terminal separator S and corresponding subgraphs G_1 and G_2

Let S be a terminal separator in G and G_1 and G_2 the corresponding subgraphs (Figure 9). The bounds will be computed locally in supplemented versions of G_2. Let C be a clique over S. We denote with (C, b) the weighted version of C with weights equal to bottleneck distances in G_1; similarly for (C, s) with weights equal to Steiner bottleneck distances in G. Let G_2' and G_2'' be the instances of the Steiner problem created by supplementing G_2 with (C, s) respectively (C, b). We compute a lower bound $lower_{con}(G_2'')$ for any Steiner tree satisfying a given constraint in G_2'' and an upper bound $upper(G_2')$ corresponding to an (unrestricted) Steiner tree in G_2'. The test condition is: $upper(G_2') < lower_{con}(G_2'')$.

Lemma 1 The test condition is valid, i.e., no Steiner minimal tree in G satisfies the constraint if $upper(G_2') < lower_{con}(G_2'')$.

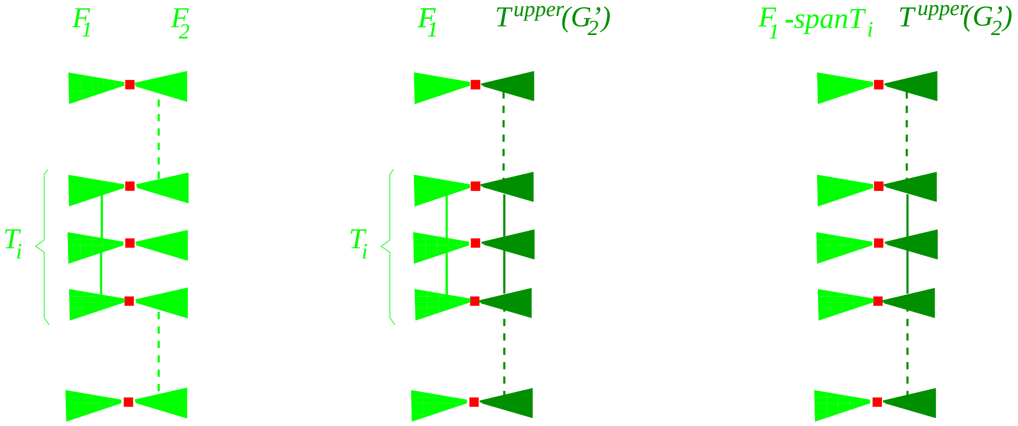

Figure 10: Construction of $T^{upper}(G')$ from $T^{opt}_{con}(G)$

Proof: Consider $T^{opt}_{con}(G)$, an (unknown) optimum Steiner tree satisfying the constraint. The subtrees of this tree restricted to subgraphs G_1 and G_2 build two forests F_1 (with connected components T_i) and F_2 (Figure 10, left). Removing F_2 and reconnecting F_1 with $T^{upper}(G_2')$ we get a feasible solution again, which is not necessarily a tree (Figure 10, middle). Let S_i be the subset of S in T_i. Consider two terminals of S_i: Removing a bottleneck on the corresponding fundamental path disconnects T_i into two connected components. Repeating this step until all terminals in S_i are disconnected in T_i, we have removed $|S_i| - 1$ bottlenecks, which together build a spanning tree $spanT_i$ for S_i (Figure 10, right). Repeating this for all T_i, we get again a feasible Steiner tree $T^{upper}(G')$ for the graph G' which is created by adding the edges of (C, s) to G. Note that the optimum solution values in G' and G are the same (see 1.1). Let $upper(G')$ be the weight of $T^{upper}(G')$. By construction of $T^{upper}(G')$, we have:

$$upper(G') = opt_{con}(G) + upper(G_2') - c(F_2) - \sum_i c(spanT_i)$$

The edge weights of the trees $spanT_i$ correspond to bottlenecks in F_1, so by definition they can not be smaller than the corresponding bottleneck distances in G_1. By construction of G_2'', all these edges (with the latter weights) are available in G_2''. Since the trees $spanT_i$ reconnect the forest F_2, together with F_2 they build a feasible solution for G_2'', which even satisfies the constraint (because F_2 did), so it has at least the cost $opt_{con}(G_2'')$. This means:

$$\begin{aligned} upper(G') &\leq opt_{con}(G) + upper(G_2') - opt_{con}(G_2'') \\ &< opt_{con}(G) + lower_{con}(G_2'') - opt_{con}(G_2'') && \text{(because of the test condition)} \\ &\leq opt_{con}(G), \end{aligned}$$

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

Studying the test condition, one detects a weakness of the used lower bound relative to the upper bound: bottleneck distances used in the lower bound correspond to single edges, whereas the Steiner bottleneck distances used in the upper bound correspond to whole paths and can be much larger. The attempt to use some paths in F_1 instead of bottlenecks fails because the tree $T_{con}^{opt}(G)$ and consequently the forest F_1 are unknown. But going through the proof, one observes that we can use the fact that the tree $T^{upper}(G_2')$ is available: Instead of removing a single edge in F_1, we can remove a (longest) key path on the corresponding fundamental path in $T^{upper}(G_2')$. This leads to the following improvement of the test: While choosing the edge weights for constructing G_2'', we can use the length of a key path in $T^{upper}(G_2')$ whenever it is larger than the corresponding bottleneck distance in G_1. However, care must be taken to keep the used key paths disjoint.

An application example for this test is given in Figure 11, where a separator of size 4 and the corresponding component G_2 are shown (black edges). Figure 12 shows the result of application of the presented reduction method (black edges contracted, grey edges remaining); in Figure 13 the reduced instance is redrawn.
This is also an example of how reduction methods based on partitioning can reduce the errors in an LP-relaxation: As shown in Figure 14, the relaxation LP_C has now an integer optimal solution. It is perhaps interesting that this improvement is mainly achieved by applying the same relaxation locally.

3 Experimental Results

In [10], we reported empirical times for exact solution of benchmark instances, which were by far the best results at the time of their submission (April 1998). Since then, we have vastly improved the times, especially for VLSI and geometric instances. The improvements are due to several techniques, described in a series of papers [9, 12]. Here we study the impact of partitioning. We compare the results of our program (as reported for example in [11]), which uses partitioning by default, with the results of exactly the same program with only the partitioning methods described in this paper switched off. As test data, we use two groups of instances from the library SteinLib [7]. The first group, the ALUE-instances, are VLSI-instances and include some of the largest such instances in SteinLib. The second group, ES1000, are "geometric graph" instances. These graphs are produced by applying FST generation (Section 2.1) to some randomly generated point sets.

All tests were performed on a PC with an AMD Athlon 1.1 GHz processor and 768 MB of main memory, using the operating system Linux 2.2.13. We used the GNU egcs 1.1.2 compiler and CPLEX 7.0 as LP-solver.

In Tables 1 and 2, we compare the running times for the exact solution of the test instances with and without the partitioning techniques described in this paper. We also give the number of nodes in the branch-and-bound tree. Studying these tables, one observes:

- The reduction methods based on partitioning help generally, with the benefit becoming more apparent on the more time-consuming instances.
- The gains are smaller (less than 30%) for the ALUE-instances. This is due to the fact that for these instances, most of the time is spent in the initial phases of the reduction process (see Section 2.1); and by the time the partitioning methods are called for the first time, the instances are usually reduced to such an extent that any improvement

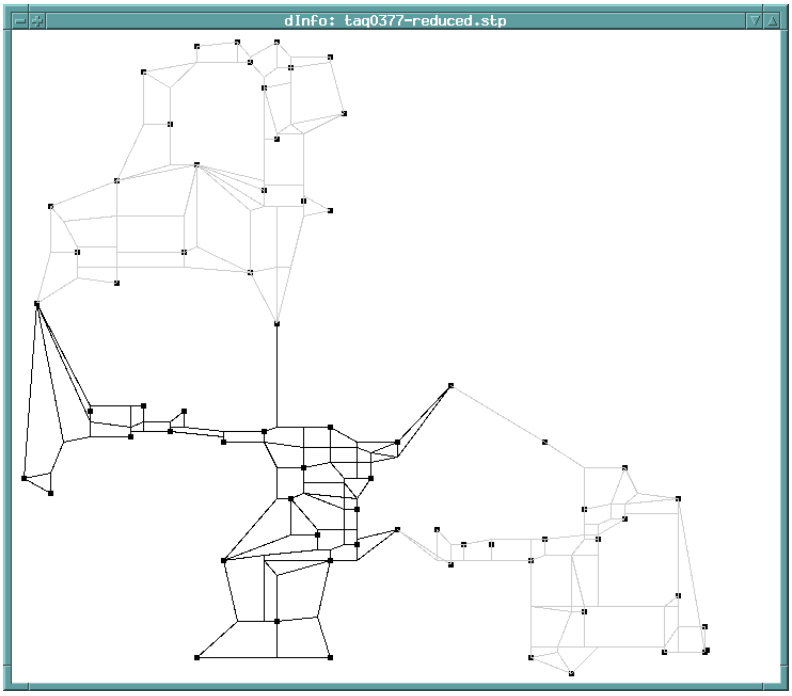

Figure 11: $|S| = 4$

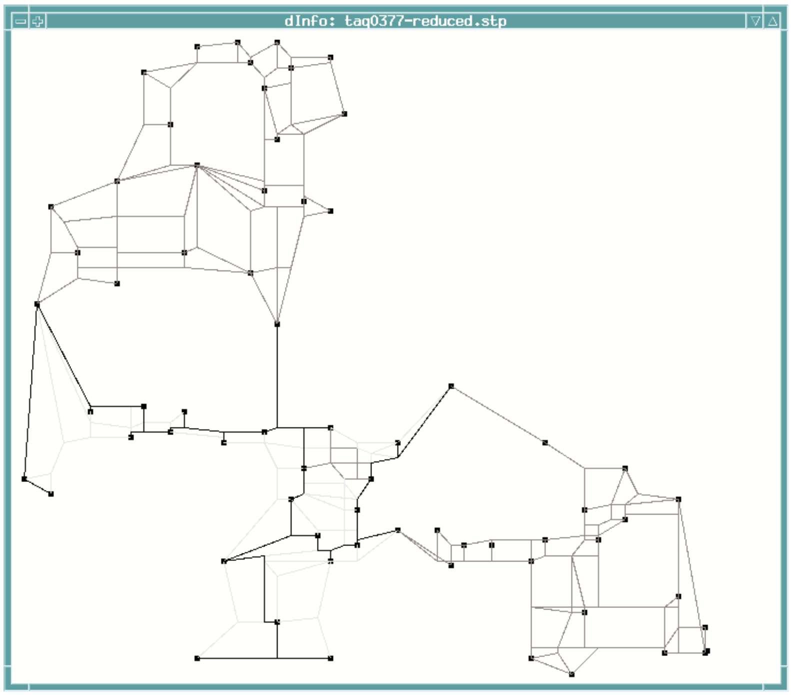

Figure 12: $(|V| = 121,\ |E| = 200,\ |R| = 41)$

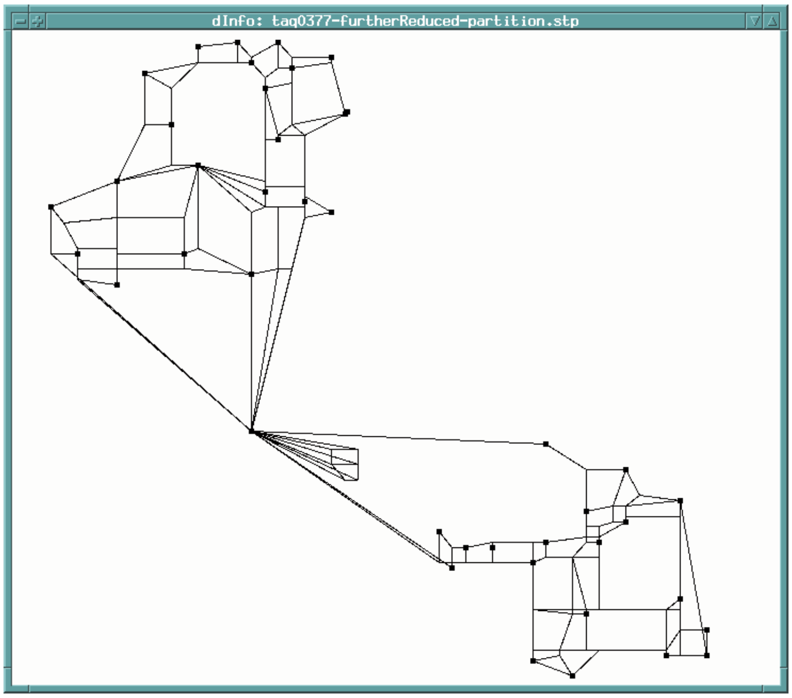

Figure 13: $(|V| = 121,\ |E| = 200,\ |R| = 41)$

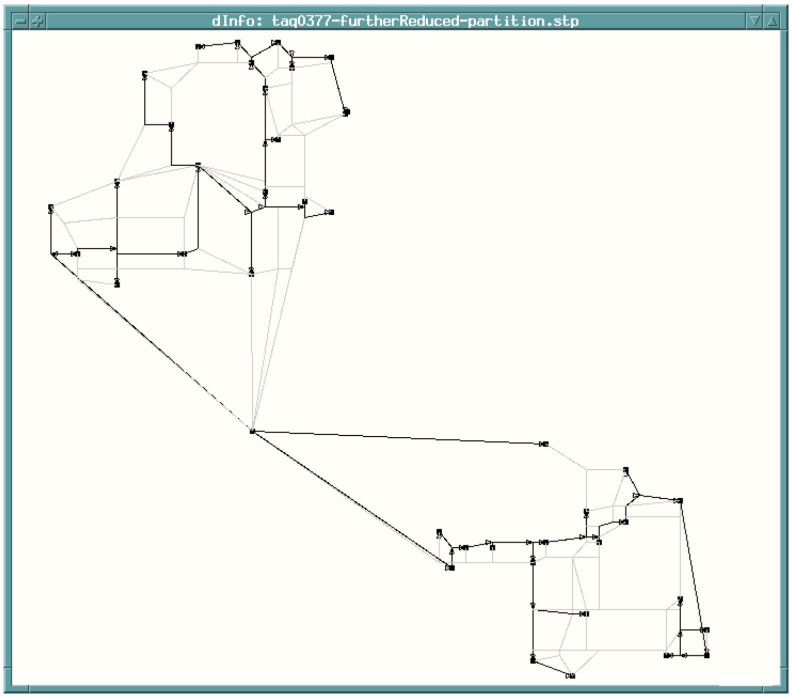

Figure 14: $v(LP_C) = 6393$

in the following stages can only have a minor overall effect. But this effect gets more significant in larger instances.

- The gains are more significant for the ES1000-instances (an overall gain of more than a factor of 3). This is due to the fact that the initial reduction stages have a smaller impact on these instances, so the effect of the partitioning methods is more apparent. The differences are even more significant in larger instances (not included); for example, our program solves the instance es10000fst from SteinLib (the largest instance of this type ever solved) in less than 20 minutes, whereas it needs more than two hours with partitioning switched off.

- Branching has not been necessary for any of the considered instances, with or without partitioning. This is partly due to the usage of our new, enhanced relaxations [12]. Without these enhancements (e.g., using LP_C in its original form), the running times would be much larger in many cases, and the impact of partitioning even more significant.

instance	size			optimum	w/o partitioning		w partitioning	
	$\|R\|$	$\|V\|$	$\|E\|$		nodes	time (s)	nodes	time (s)
alue2087	34	1244	1971	1049	1	0.06	1	0.06
alue2105	34	1220	1858	1032	1	0.05	1	0.05
alue3146	64	3626	5869	2240	1	0.31	1	0.30
alue5067	68	3524	5560	2586	1	0.70	1	0.69
alue5345	68	5179	8165	3507	1	3.74	1	3.57
alue5623	68	4472	6938	3413	1	1.76	1	1.66
alue5901	68	11543	18429	3912	1	2.68	1	2.60
alue6179	67	3372	5213	2452	1	0.61	1	0.62
alue6457	68	3932	6137	3057	1	0.81	1	0.84
alue6735	68	4119	6696	2696	1	0.68	1	0.69
alue6951	67	2818	4419	2386	1	0.59	1	0.56
alue7065	544	34046	54841	23881	1	101.00	1	95.40
alue7066	16	6405	10454	2256	1	7.23	1	7.38
alue7080	2344	34479	55494	62449	1	105.00	1	81.80
alue7229	34	940	1474	824	1	0.03	1	0.03

Table 1: Exact solution of ALUE-instances with and without partitioning

4 Concluding Remarks

In this paper, we have presented a new approach that uses a partitioning scheme through vertex separators as the basis for new reduction methods. We introduced two such methods and looked at their impact empirically.
Of course, there are also other approaches for exploiting the idea of partitioning. For example, we also use an approach based on Voronoi regions in graphs to accelerate the computation

instance	size			optimum	w/o partitioning		w partitioning	
	$\|R\|$	$\|V\|$	$\|E\|$		nodes	time (s)	nodes	time (s)
es1000fst01	1000	2865	4267	230535806	1	87.60	1	20.00
es1000fst02	1000	2629	3793	227886471	1	17.20	1	10.20
es1000fst03	1000	2762	4047	227807756	1	16.40	1	15.30
es1000fst04	1000	2778	4083	230200846	1	19.30	1	14.70
es1000fst05	1000	2676	3894	228330602	1	34.30	1	11.40
es1000fst06	1000	2815	4162	231028456	1	199.00	1	28.20
es1000fst07	1000	2604	3756	230945623	1	12.00	1	5.41
es1000fst08	1000	2834	4207	230639115	1	42.80	1	19.50
es1000fst09	1000	2846	4187	227745838	1	34.40	1	17.10
es1000fst10	1000	2546	3620	229267101	1	34.30	1	6.15
es1000fst11	1000	2763	4038	231605619	1	22.60	1	13.10
es1000fst12	1000	2984	4484	230904712	1	31.40	1	22.00
es1000fst13	1000	2532	3615	228031092	1	10.70	1	5.40
es1000fst14	1000	2840	4200	234318491	1	91.40	1	19.40
es1000fst15	1000	2733	3997	229965775	1	25.80	1	10.50

Table 2: Exact solution of ES1000FST-instances with and without partitioning

of lower bounds (mainly by generating constraints in subgraphs) and to strengthen the upper bounds (mainly by improving them locally in subgraphs). But since these methods are conceptually different, we have not described them here.

References

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

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

[3] M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.

[4] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

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

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

[7] T. Koch and A. Martin. SteinLib. `http://elib.zib.de/steinlib/steinlib.php`, 2001.

[8] N. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[9] T. Polzin and S. Vahdati Daneshmand. Extending reduction techniques for the Steiner tree problem: A combination of alternative- and bound-based approaches. Research Report MPI-I-2001-1-007, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, 2001.

[10] T. Polzin and S. Vahdati Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112:263–300, 2001.

[11] T. Polzin and S. Vahdati Daneshmand. On Steiner trees and minimum spanning trees in hypergraphs. Research Report MPI-I-2001-1-005, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, 2001.

[12] T. Polzin and S. Vahdati Daneshmand. Using tighter relaxations for the Steiner problem. MPI Saarbrücken, Universität Mannheim, 2001.

[13] D. M. Warme. *Spanning Trees in Hypergraphs with Applications to Steiner Trees.* PhD thesis, University of Virginia, 1998.

[14] D. M. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D-Z. Du, J. M. Smith, and J. H. Rubinstein, editors, *Advances in Steiner Trees*, pages 81–116. Kluwer Academic Publishers, 2000.

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