Practical Partitioning-Based Methods for the Steiner Problem

Tobias Polzin[1] and Siavash Vahdati Daneshmand[2]

[1] HaCon Ingenieurgesellschaft mbH, Hannover, Germany
[2] Theoretische Informatik, Universität 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 partitioning-based paradigms rarely leads to empirically successful algorithms. In this paper, we present two approaches to the Steiner problem based on partitioning. The first uses the fixed-parameter tractability of the problem with respect to a certain width parameter closely related to path-width. The second approach is based on vertex separators and is new in the sense that it uses partitioning to design reduction methods. Integrating these methods into our program package for the Steiner problem accelerates the solution process on many groups of instances and leads to a fast solution of some previously unsolved benchmark instances.

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 with many important applications in network design in general and VLSI design in particular [3, 6].

For such ($\mathcal{NP}$-hard) problems, straightforward application of the classical partitioning paradigms rarely leads to empirically successful algorithms. Divide-and-conquer techniques are not generally applicable, because one usually cannot find independent subproblems. Dynamic programming techniques can indeed be applied, but they are usually practical only for a very limited range of instances.

In this paper, we present two practically helpful methods which are based on partitioning. In Sect. 2, we present an algorithm that uses the fixed-parameter tractability of the problem with respect to a certain width parameter closely related to path-width. The running time of the algorithm is linear in the number of vertices when the path-width is constant, and it is practical when the considered graph has a small width. In Sect. 3, we introduce the approach of using partitioning for reducing the size of the instance (i.e., developing partitioning-based reduction methods). We present two new reduction methods based on this approach. Methods like those described in this paper are not conceived as stand-alone solution routines for a wide range of instances; but as subroutines of more complex optimization programs, they are much more broadly applicable.

C. Àlvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 241–252, 2006.
© Springer-Verlag Berlin Heidelberg 2006

An additional feature is that by the cooperation of the methods presented here we can already profit from small width in subgraphs of a given instance; these interactions will be elaborated in Sects. 3 and 4. Finally, some experimental results are presented in Sect. 4.

1.1 Preliminaries

The **Steiner (tree) problem in networks** can be stated as follows: Given a (connected) network $G = (V, E, c)$ (with vertices $V = \{v_1, \ldots, v_n\}$, edges E and edge weights $c_e > 0$ for all $e \in E$) and a set R, $\emptyset \neq R \subseteq V$, of **required vertices** (or **terminals**), find a minimum weight tree in G that spans R (a **Steiner minimal tree**). For more information on this problem, see [6].

We define $r := |R|$. If we want to stress that v_i is a terminal, we will write z_i instead of v_i. A **bottleneck** of a path P is a longest edge in P. The **bottleneck distance** $b(v_i, v_j)$ or b_{ij} between two vertices v_i and v_j in G is the minimum bottleneck length taken over all paths between v_i and v_j in G. An **elementary path** is a path in which only the endpoints may be terminals. Any path between two vertices can be broken at inner terminals into one or more elementary paths. The **Steiner distance along a path** P between v_i and v_j is the length of a longest elementary path in P. The **bottleneck Steiner distance** (sometimes also called "special distance") $s(v_i, v_j)$ or s_{ij} between v_i and v_j in G is the minimum Steiner distance taken over all paths between v_i and v_j in G. A major relevance of bottleneck Steiner distances is that the cost of an optimum Steiner tree in G does not change by deleting edges (v_i, v_j) with $c_{ij} > s_{ij}$ (or, conversely, by inserting edges (v_i, v_j) of length s_{ij}) [4]. For any subset $S \subseteq R$, all b_{ij}, s_{ij} with $v_i, v_j \in S$ can be computed in time $O(|E| + |V| \log |V| + |S|^2)$ [4, 9].

2 Using (Sub-) Graphs of Small Width

In this section, we present a practical algorithm for solving the Steiner problem in graphs with a small width parameter. The width concept used here is closely related to path-width, as we will show in Sect. 2.3. For an overview of subjects concerning path-width and the more general notion of tree-width see [1]. The running time of the algorithm is linear in the number of vertices when the width is constant, thus it belongs to the category of algorithms exploiting the fixed-parameter (FP) tractability of $\mathcal{NP}$-hard problems. There are already FP-polynomial algorithms for the Steiner problem in graphs. Specifically, in [8] a linear-time algorithm for graphs with bounded tree-width is described. But this algorithm is more complicated than the one we present here, and its running time grows faster with the (tree-) width (it is given in [8] as $O(nf(d))$ with $f(d) = \Omega(d^{4d})$, where d is the tree-width of the graph). Therefore, it seems to be not as practical as our algorithm, and no experimental results are reported in [8]. In a different context (network reliability), a similar approach using path-width is described in [11], which is practical for a range of path-widths similar to the one considered here. We also adapted that approach to the Steiner problem, but the experimental results were not as good as with the one presented here.

2.1 The Basic Algorithm

We maintain a set of already visited vertices and a subset of them (the **border**) that are adjacent to some non-visited vertex. In each step, the set of visited vertices is extended by one non-visited vertex adjacent to the border. For all possible partitions in each border, we calculate (the cost of) a forest of minimum cost that contains all visited terminals with the property that each tree in the forest spans just one of the partition sets. We are finished when all vertices have been visited. The observation behind this approach is as follows: For any optimal Steiner tree T, the subgraph of T when restricted to the visited vertices is a forest, which also defines a partition in the border. The plan is to calculate these forests in a bottom-up manner, in each step using the values calculated in the previous step. If the size of all borders can be bounded by a constant, the total time can be bounded by the number of steps times another constant.

For an arbitrary ordering $v_1, \ldots, v_n$ of the vertices and any $s \in \{1, \ldots, n\}$, we define $V_s := \{v_1, \ldots, v_s\}$ and denote with G_s the subgraph of G with vertex set V_s. In the following, we assume an ordering of the vertices with the property that all G_s are connected. (For example, a depth-first-search traversal of G delivers such an ordering.) We denote with B_s the border of V_s, i.e., $B_s := \{v_i \in V_s \mid \exists (v_i, v_j) \in E : v_j \in V \setminus V_s\}$. With L_s we denote the set of vertices that leave the border after step s, i.e., $L_s := (B_{s-1} \cup \{v_s\}) \setminus B_s$. The inclusion of v_s in this definition should cover the case that v_s has no adjacent vertices in $V \setminus V_s$; this simplifies some other definitions. Consider a set Q, $B_s \cap R \subseteq Q \subseteq B_s$, and a partitioning $\mathcal{P} = \{P_1, \ldots, P_t\}$ of Q into non-empty subsets, i.e., $\dot{\bigcup}_{1 \le i \le t} P_i = Q$ and $\emptyset \notin \mathcal{P}$. The number of ways of partitioning a set of $b := |Q|$ elements into t non-empty subsets is $\left\{ {b \atop t} \right\}$, a Stirling number of the second kind, and the total number of partitions is $B(b)$, the b-th Bell number. For a partition $\mathcal{P}$ and a set $L \subseteq V$ we define $\mathcal{P} - L := \{P'_i \mid P_i \in \mathcal{P},\ P'_i = P_i \setminus L\}$. Let $F(s, \mathcal{P})$ be a forest of minimum cost in G_s containing all terminals in V_s and consisting of t (vertex-disjoint) trees $T_1, \ldots, T_t$ such that T_i spans P_i for all $i \in \{1, \ldots, t\}$. With $c(s, \mathcal{P})$ we denote the cost of $F(s, \mathcal{P})$.

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

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

The cost of an optimal Steiner tree in G is: $\min\{c(s, \mathcal{P}) \mid R \subseteq V_s, |\mathcal{P}| = 1\}$. Obviously the forests $F(s, \mathcal{P})$ (and an optimal Steiner tree) can be calculated following the same pattern.

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

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

Let p_s denote the number of partitions at step s. We have $p_s = \sum_{R\cap B_s \subseteq Q \subseteq B_s} B(|Q|)$, where $B(b)$ is the b-th Bell number; so $p_s = O(2^{b_s} B(b_s))$ with $b_s := |B_s|$. We only maintain one global list of partitions, which is updated after each step, keeping for each valid partition a solution of minimum cost. Because of the loop in Line 11, this list can grow to at most $l_s := 2^{b_s} p_s = O(2^{2b_s} B(b_s))$ partitions. Eliminating the duplicates can be done by sorting the list: Each partition can be represented as a (lexicographically) sorted string (of length at most $2b_s$) of sorted substrings (of length at most b_s) separated by some extra symbol. Using radix sort, all the individual sortings of l_s strings can be done in total time $O(n+l_s b_s)$. Sorting the resulting list of l_s strings takes again time $O(n + l_s b_s)$. We set aside for now a total extra time of $O(|E|)$ for the operations on edges; and assume that an ordering of vertices is given (these points are explained below). The (rest of the) operations in Lines $12 - 17$ can be carried out in time $O(b_s)$. This gives the total running time $O(\sum_{s=1}^{n} b_s 2^{2b_s} B(b_s))$. Note that this bound implicitly contains the extra amortized time $O(|E|)$ by the following observation: After a vertex is visited for the first time, it remains in the border as long as it has some non-visited adjacent vertex; so each edge is accounted for by its first-visited endpoint. Now if we can guarantee an upper bound b for the size of all borders, we have an upper bound of $O(nb2^{2b}B(b))$ for the running time. By upper-bounding $B(b)$ roughly with $(2b)^b$ we get the running time $O(n2^{b \log b + 3b + \log b})$. This means that the algorithm runs in linear time for constant b and, for example, in time $O(n^2)$ for $b = \log n / \log\log n$.

For the actual implementation, some modifications are used. For example, avoiding duplicate partitions is done using hashing techniques, which reduces the amount of necessary memory. Also, some heuristics are used to recognize partitions that cannot lead to an optimal Steiner tree.

2.2 Ordering the Vertices

In Sect. 2.3, we will show that finding an ordering of vertices such that the maximum border size equals b is (up to some easy transformations) equivalent to finding a path-decomposition of path-width b. The problem of deciding whether the path-width of a given graph is at most b, and if so, finding a path-decomposition of width at most b is $\mathcal{NP}$-hard for general b, but for constant b, this problem can be solved in linear time [2]. However, already for $b > 4$ the corresponding algorithm is no longer practical [12], and it seems that no practical exact algorithm is known for more general cases. Furthermore, we have a more specific scenario (for example we differentiate between terminals and non-terminals). So for the actual implementation we use a heuristic, which has produced quite satisfactory results for our applications. The heuristic chooses in each step a vertex v_s adjacent to the border using a (ad hoc) priority function of the following parameters: size of resulting set L_s, number of visited vertices in the adjacency list of v_s, membership of v_s in R, and number of edges connecting V_s and $V \setminus V_s$. We select the starting vertex by trying a small number of terminals and performing a sweep through the graph without actually computing the partitions. In each sweep, we estimate the overall number of resulting partitions by summing up the (ad hoc) values $|B_s|^2|B_s \setminus R|$ in each step. Finally, we select the terminal that yields the smallest estimated number.

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

2.3 Relation to Path-Width

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

A **path-decomposition** of a graph $G = (V, E)$ is a sequence of subsets of vertices $(U_1, U_2, \ldots, U_p)$, such that

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

The **path-width** of a path-decomposition $(U_1, U_2, \ldots, U_p)$ is $\max\{|U_i| \mid 1 \leq i \leq p\} - 1$. The **path-width** of a graph G is the minimum path-width over all possible path-decompositions of G. Note that the 3rd condition in the definition of path-decomposition can be rewritten as follows: There are functions *start*, *end*: $|V| \to \{1, \ldots, p\}$ with $v \in U_j \Leftrightarrow \mathit{start}(v) \leq j \leq \mathit{end}(v)$. We call a path-decomposition with functions *start*, *end* **bijective** if the mapping *start* is a bijection; and **minimal** if it holds: $\mathit{end}(v) \geq i \Rightarrow \mathit{start}(v) = i \vee \exists (v, w) \in E : \mathit{start}(w) \geq i$. We state the following two lemmas (for the proofs, see [15]).

Lemma 1. *Every path-decomposition can be transformed to a minimal and bijective path-decomposition of no larger path-width.*

Lemma 2. *Let* $(U_1, \ldots, U_n)$ *be a minimal, bijective path-decomposition of* G *with the functions start and end. Assume that the vertices are ordered according to their start values, i.e.,* $start(v) = s \Leftrightarrow v \in V_s \setminus V_{s-1}$. *For each* $s \in \{1, \ldots, n\}$ *it holds:* $U_s = \{v_s\} \cup B_{s-1}$.

It follows that every path-decomposition of G can be transformed to a path-decomposition $U = (U_1, \ldots, U_n)$ of no larger path-width such that for an ordering of vertices according to the *start* function of U it holds: $U_s = \{v_s\} \cup B_{s-1}$. On the other hand, it is easy to verify that each ordering of vertices and the corresponding sequence of borders $(B_1, \ldots, B_n)$ deliver a (minimal, bijective) path-decomposition U by setting $U_s = \{v_s\} \cup B_{s-1}$. In each case, we have: $\max\{|U_s| \mid 1 \leq s \leq n\} - 1 = \max\{|B_{s-1}| \mid 1 \leq s \leq n\}$.

3 Partitioning as a Reduction Technique

In this section, we present our approach of using partitioning to design reduction methods, i.e., methods to reduce the size of a given instance without destroying an optimal solution. This approach turns out to be quite effective in the context of the Steiner problem, and it can also be useful for other problems. Furthermore, it offers a straightforward path for a distributed implementation.

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

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. This is particularly the case for geometric instances after a geometric preprocessing (FST generation phase, see [16]), but also for general instances after applying powerful reduction techniques (see [10]). In both cases, the resulting intermediate instances frequently have many small terminal separators. For geometric instances, the existence of small vertex separators was already observed in [13]; however, in that work a standard dynamic programming approach was suggested for exploiting this observation, which is not nearly as practical as the approach chosen here. The difference between the two approaches will be elaborated in Sect. 3.2.

3.1 Finding Terminal Separators

It is well known that the problem of finding vertex separators (or the vertex connectivity problem) can be solved by network flow techniques in the so-called split graph [5]. This graph 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 for a graph with n vertices and m edges in time $O(\min\{kmn, (k^3 + n)m\})$ [5] (this bound comes from a combination of augmenting path and preflow-push methods).

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

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

3.2 Reduction by Case Differentiation

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

The case $|S| = 1$ corresponds to articulation points (and biconnected components). It is known [6] that the subinstances corresponding to the biconnected components can be solved independently.

The case $|S| = 2$ corresponds to separation pairs (and triconnected components). Note that the two subinstances (corresponding to two subgraphs G_1 and G_2) are no longer independent. Now, for any Steiner minimal tree T, two cases are possible:

1. 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.
2. The terminals in S are connected by T inside G_1. Now there are two subtrees of T inside G_2, and we do not know in advance how the terminals of G_2 are divided between them. But one can observe that the problem can be solved by merging the terminals in S and solving the resulting subinstance.

Since we do not know T in advance, for a direct solution we must also consider both cases for the complement G_1. But if G_2 is relatively small, the solution of

the complementary subinstance can be almost as time-consuming as the solution of the original instance, meaning that not much is gained (or time may even be lost, because now we have to solve it twice). A classical approach would search for components of almost equal size, but we choose a different approach. The main 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. After these modifications, we have a smaller instance with the same optimum solution value, and we can proceed with this reduced instance.

For the general case ($|S| = k$), the basic approach is the same as for the case $|S| = 2$; but a larger number of cases must be considered now. Remember from Sect. 2.1 that the number of cases is $B(k)$, the k-th Bell number. So this method can be used profitably only for small k (usually for $k \leq 4$).

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 is the following:

Lemma 3. *Let z_i and z_j be two terminals in the separator S and let b^1_{ij} and s^2_{ij} be the bottleneck distance in G_1 and bottleneck Steiner distance in G_2 between z_i and z_j, respectively. Then the cases in which z_i and z_j are connected in G_1 can be discarded if $b^1_{ij} \geq s^2_{ij}$.*

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

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.

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

3.3 Reduction by Local Bounds

The general principle of bound-based reduction methods is to compute an upper bound *upper* and a lower bound under some constraint $lower_{constrained}$. The constraint cannot be satisfied by any optimal solution if $lower_{constrained} > upper$. The constraint is usually that the solution must contain some pattern (e.g., an edge or more complex patterns like trees, see [10]). Once it is established that the test condition (the inequality above) is valid, the corresponding pattern (e.g., the edge) can be excluded, yielding a smaller (reduced) instance with the same optimal solution. 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 $\tilde{c}$. We can use a fast method to compute a constrained lower bound with respect to $\tilde{c}$. The sum of the two bounds is a lower bound for the value of any solution satisfying the constraint. For details, see [4, 10].

In the following, we show how to develop a reduction test condition based on local bounds, the application is analog to the usage of globally computed bounds for reductions, see [10]. This approach has two main advantages: The bounds can be computed faster; and there is less chance that the deviations between the original instance and its linear relaxation can accumulate (and thus deteriorate the computed bounds and methods using them). The main difficulty is that the bounds must somehow take the dependence on the rest of the graph into account.

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

Lemma 4. *The test condition is valid, i.e., no Steiner minimal tree in G satisfies the constraint if $upper(G_2') < lower_{constrained}(G_2'')$.*

Proof. Consider $T^{opt}_{con}(G)$, an optimum Steiner tree of cost $opt_{con}(G)$ satisfying the constraint. The subtrees of this tree restricted to subgraphs G_1 and G_2 build two forests F_1 (with connected components T_i) and F_2 (Fig. 1, 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 (Fig. 1, 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 (Fig. 1, 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.

Remember from Sect. 1.1 that the optimum solution value does not change by inserting any edges (v_i, v_j) of length s_{ij} into G, so the optimum solution values in G' and G are the same. Let $upper(G')$ be the weight of $T^{upper}(G')$. By construction of $T^{upper}(G')$, we have: $upper(G') = opt_{con}(G) + upper(G_2') - c(F_2) - \sum_i c(spanT_i)$. The edge weights of the trees $spanT_i$ correspond to bottlenecks in F_1, so by definition they cannot be smaller than the corresponding bottleneck distances in G_1. By construction of G_2'', all these edges (with the latter weights) are available in G_2''. Since the trees $spanT_i$ reconnect the forest F_2, together with F_2 they build a feasible solution for G_2'', which even satisfies the constraint (because F_2 did), so it has at least the cost $opt_{con}(G_2'')$. This means:

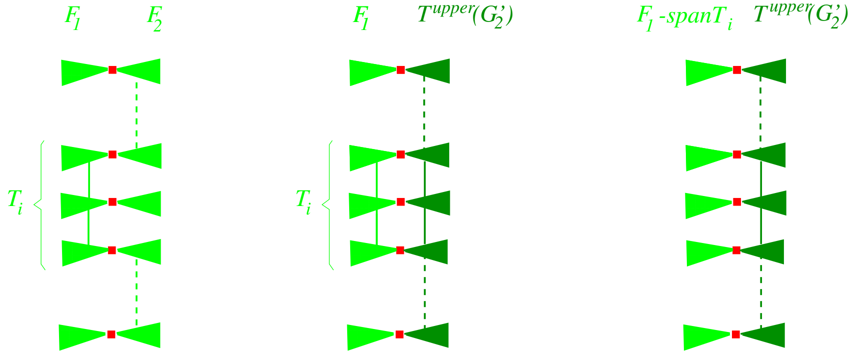

Fig. 1. Construction of $T^{upper}(G')$ from $T^{opt}_{con}(G)$

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

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

4 Experimental Results

In this section, we study the empirical impact of the presented methods. Methods like those in this paper cannot be expected to be usable as stand-alone solution methods for a wide range of instances; however, they are very helpful as subroutines in many cases. By integrating these methods in our program package for the Steiner problem [15], we could already solve several previously unsolved benchmark instances from SteinLib [7], which otherwise could not be tackled in reasonable times. For the experiments in this paper, we concentrate on instances from a current real-world application (the LOFAR radio telescope project, which is described below). An additional advantage of these instances (beside their interesting practical background) is that we are already able to solve all of them without the techniques described in this paper, so we can present concrete running times for different solution methods, which also demonstrate the improvements gained by the techniques presented here.

The LOFAR (LOw Frequency ARray) project is concerned with the construction of the largest radio telescope of the world, which is currently being built by ASTRON in the Netherlands. It consists of many sensor stations that have to be located along five spiral arms, with each arm stretching over hundreds of kilometers. The distance between adjacent sensor stations along each arm should increase in a logarithmic progression. The LOFAR sensor stations must be placed while avoiding obstacles where stations cannot be placed geographically (e.g., the North Sea and population centers). The sensor stations should be

connected by (expensive) optical fibers in order to send the data collected by the sensors to a central computer for processing and analysis. Since cheaper existing optical fibers with unused capacity can be purchased from cable providers, they should be utilized to set up the optical network in the most economical way. Note that the cost function is quite general, so these instances are not (pure) geometric instances. A research branch in the LOFAR project is working on a simulation-optimization framework [14], where different topological designs and cost functions are developed. The Steiner problem is used to model the routing part of the problem, in order to find a low-cost cabling for each of the scenarios considered. The large amounts of money involved justify the wish for optimal (or at least provably near-optimal) solutions. On the other hand, changes in the scenario give rise to a large number (hundreds) of new candidate instances, so excessively long runs for single instances are not tolerable.

The latest collection of instances we received from the LOFAR project [14] consists of more than one hundred instances, divided in 13 groups (with different settings of parameters). All these instances have 887 vertices, thereof 101 terminals, and 163205 edges; but with various cost functions. For the experiments here, we (randomly) chose one instance from each group (the results inside each group were similar). We compare three (exact) solution methods:

(I) As a basis for comparisons, we use a somehow standard branch-and-cut approach based on the classical (directed) cut formulation of the Steiner problem, using a cut generating routine as described in [9]. However, in contrast to [9], here we use no reductions at all. Since the program in [9] heavily exploits the reduction-based methods (e.g. for computing sharp upper bounds), here as a substitute we utilize the MIP optimizer of CPLEX 8.0 after solving the LP-relaxation to get a provably optimal integer solution (the additional times for the MIP optimizer were relatively marginal for the considered instances).

(II) In the second set of experiments, we use exactly the same cut-based algorithm as under (I), but this time after performing our strong reduction techniques from [10] as preprocessing.

(III) Finally, in the third set of experiments we use exactly the same reduction techniques as under (II), but additionally we use the partitioning-based techniques described in this paper. The cut-based routine is dropped, so no LP-solver is used at all.

The results are summarized in Table 1 (all computations were performed on a machine with an INTEL Pentium-4 3.4 GHz processor). We observe:

- The reduction techniques can heavily accelerate the solution process, a fact that is meanwhile well established. For the considered set of instances, our previous reduction techniques already improve the solution times of the branch-and-cut algorithm by more than one order of magnitude.
- The partitioning-based techniques presented in this paper improve the (exact) solution times by one additional order of magnitude, thereby eliminating the need for an LP-solver like CPLEX altogether for all considered instances.

Table 1. Summarized solution times for the LOFAR instances using different methods

Solution Method	(I) (B&C)	(II) (Preprocessing + B&C)	(III) (Preprocessing + Partitioning)
Average Solution Time (seconds)	396	11	1.2

References

1. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.
2. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
3. X. Cheng and D.-Z. Du, editors. *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*. Kluwer Academic Publishers, Dordrecht, 2001.
4. C. W. Duin. Preprocessing the Steiner problem in graphs. In D. Du, J. Smith, and J. Rubinstein, editors, *Advances in Steiner Trees*, pages 173–233. Kluwer, 2000.
5. M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.
6. F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1992.
7. T. Koch and A. Martin. SteinLib. `http://elib.zib.de/steinlib`, 2001.
8. E. Korach and N. Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded tree-width. Technical Report 632, Technicon - Israel Institute of Technology, Computer Science Department, Haifa, Israel, 1990.
9. T. Polzin and S. Vahdati Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112:263–300, 2001.
10. T. Polzin and S. Vahdati Daneshmand. Extending reduction techniques for the Steiner tree problem. In R. Möhring and R. Raman, editors, *ESA 2002*, volume 2461 of *Lecture Notes in Computer Science*, pages 795–807, 2002. Springer.
11. A. Pönitz and P. Tittmann. Computing network reliability in graphs of restricted pathwidth. Technical report, Hochschule Mittweida, 2001.
12. H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, 1998.
13. J. S. Salowe and D. M. Warme. Thirty-five point rectilinear Steiner minimal trees in a day. *Networks*, 25:69–87, 1995.
14. L. P. Schakel. Personal communication, 2005. Faculty of Economics, University of Groningen, and `http://www.lofar.org/`.
15. S. Vahdati Daneshmand. *Algorithmic Approaches to the Steiner Problem in Networks*. PhD thesis, University of Mannheim, 2004. `http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2004/176`.
16. 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, 2000.