mpi INFORMATIK

Using (Sub)graphs of Small Width for Solving the Steiner Problem

Tobias Polzin   Siavash Vahdati

MPI–I–2002–1–001   February 2002

FORSCHUNGSBERICHT   RESEARCH REPORT

MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Stuhlsatzenhausweg 85   66123 Saarbrücken   Germany

Authors' Addresses

Tobias Polzin
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: polzin@mpi-sb.mpg.de

Siavash Vahdati Daneshmand
Theoretische Informatik, Universität Mannheim,
68131 Mannheim, Germany
email: vahdati@informatik.uni-mannheim.de

Abstract

For the Steiner tree problem in networks, we present a practical algorithm that uses the fixed-parameter tractability of the problem with respect to a certain width parameter closely related to pathwidth. The running time of the algorithm is linear in the number of vertices when the pathwidth is constant. Combining this algorithm with our previous techniques, we can already profit from small width in subgraphs of an instance. Integrating this algorithm into our program package for the Steiner problem accelerates the solution process on some groups of instances and leads to a fast solution of some previously unsolved benchmark instances.

Keywords

Steiner problem; Pathwidth, Dynamic Programming, Fixed-parameter tractable

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].

In this paper, we present a practical algorithm for solving the Steiner problem in graphs with a small width-parameter (to be defined below). The running time of the algorithm is linear in the number of vertices when the width is constant, thus it belongs to the category of algorithms exploiting the fixed-parameter (FP) tractability of NP-hard problems. But the applicability of this algorithm is much broader in the context of our work. Due to the use of reduction techniques based on partitioning [13] we can already profit from small width in subgraphs of the given instance. These techniques are in turn applicable to a very wide range of instances of the Steiner problem in networks after applying extended reduction methods [10] and instances of geometric Steiner problems after FST-generation [12, 15].

The width-concept in this paper is closely related to pathwidth. (For on overview on subjects concerning pathwidth and the more general notion of treewidth see [2].) There are already FP-polynomial algorithms for the Steiner problem in graphs; specifically, in [8] a linear-time algorithm for graphs with bounded treewidth is described. But this algorithm is more complicated than the one we present here, and its running time grows faster with the (tree-) width (it is given in [8] as $O(nf(d))$ with $f(d) = \Omega(d^{4d})$, where d is the treewidth of the graph); so it seems to be not as practical as our algorithm, and no experimental results are reported in [8]. In a different context (network reliability), a similar approach using pathwidth is described in [9], which is practical for a range of pathwidths similar to the one considered in this paper. We adapted this approach to the Steiner problem, but the empirical results were not as good as with the approach chosen in this paper.

The outline of this paper is as follows: After some definitions in the next subsection, in Section 2 we present the basic idea of the algorithm and a dynamic programming implementation of it. We analyze the running time of the algorithm and discuss some important subroutines. In Section 3 we show an equivalence of the width concept explicitly used in the algorithm with the concept of pathwidth. In Section 4 we present some ideas for making the algorithm faster empirically. Section 5 contains some preliminary experimental results. Finally, in Section 6 we briefly discuss some alternative ways for using the concept of small width in graphs.

1.1 Definitions

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 $c_e > 0$ 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**). If we want to stress that v_i is a terminal, we will write z_i instead of v_i.

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

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

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

2 The Algorithm

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

For any ordering $v_1, \ldots, v_n$ of the vertices and any $s \in \{1, \ldots, n\}$, we define $V_s := \{v_1, \ldots, v_s\}$ and denote with G_s the subgraph of G with vertex set V_s. In the following, we assume an ordering of the vertices with the property that all G_s are connected. (For example, a depth-first-search traversal of G delivers such an ordering.)
We denote with B_s the border of V_s, i.e., $B_s := \{v_i \in V_s \mid \exists (v_i, v_j) \in E : v_j \in V \setminus V_s\}$. With L_s we denote the set of vertices that leave the border after step s, i.e., $L_s := (B_{s-1} \cup \{v_s\}) \setminus B_s$. The inclusion of v_s in this definition should cover the case that v_s has no adjacent vertices in $V \setminus V_s$; this simplifies some other definitions.
Consider a set Q, $B_s \cap R \subseteq Q \subseteq B_s$, and a partitioning $\mathcal{P} = \{P_1, \ldots, P_t\}$ of Q into nonempty subsets, i.e., $\dot{\bigcup}_{1 \le i \le t} P_i = Q$ and $\emptyset \notin \mathcal{P}$. For a partition $\mathcal{P}$ and a set $L \subseteq V$ we define $\mathcal{P} - L := \{P'_i \mid P_i \in \mathcal{P},\ P'_i = P_i \setminus L\}$. Let $F(s, \mathcal{P})$ be a forest of minimum cost in G_s containing all terminals in V_s and consisting of t (vertex-disjoint) trees $T_1, \ldots, T_t$ such that T_i spans P_i for all $i \in \{1, \ldots, t\}$. With $c(s, \mathcal{P})$ we denote the cost of $F(s, \mathcal{P})$.
Let $V_0 = B_0 = \emptyset$ and set $c(0, \emptyset) = 0$. The value $c(s, \mathcal{P})$ can be calculated recursively using a case differentiation:

I) $v_s \in Q$:

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

II) $v_s \notin Q$:

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

The cost of an optimal Steiner tree in G is

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

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

2.1 A Dynamic Programming Implementation

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

```
s = 0; r = 0; opt = ∞;          "r : number of visited terminals"
c(s, ∅) = 0;
while s < n :
    s = s + 1; determine v_s, B_s and L_s;
    if v_s ∈ R : r = r + 1;
    forall P with c(s − 1, P) ≠ ∞ :
        oldCost = c(s − 1, P);
        if v_s ∉ R and ∅ ∉ P − L_s :
            c(s, P − L_s) = oldCost;
        Pcandidates = {P_i ∈ P | ∃v_i ∈ P_i : (v_i, v_s) ∈ E};
        forall Pconnect ⊆ Pcandidates :
            connectionCost= Σ_{P_i∈Pconnect} min_{v_i∈P_i,(v_i,v_s)∈E} c(v_i, v_s);
            Pstay = P\Pconnect; Pnew = ({{v_s} ∪ ⋃_{P_i∈Pconnect} P_i} ∪ Pstay) − L_s;
            if ∅ ∉ Pnew and c(s,Pnew) > oldCost + connectionCost :
                c(s,Pnew) = oldCost + connectionCost;
                if r = |R| and |Pnew| = 1 :          "feasible Steiner tree"
                    opt = min(opt, c(s, Pnew));
```

2.1.1 Running time

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

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

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

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

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

2.2 Ordering the Vertices

In Section 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 pathwidth b. The problem of deciding whether the pathwidth of a given graph is at most b, and if so, finding a path-decomposition of width at most b is $\mathcal{NP}$-hard for general b [1], but for constant b, this problem can be solved in linear time [3]. However, the corresponding algorithm is not practical even for $b > 4$ [14], and it seems that no practical exact algorithm is known for more general cases [4]. Furthermore, we have a more special scenario; for example we differentiate between terminals and nonterminals. So for the actual implementation we use a heuristic, which has produced quite satisfactory results for our applications. The heuristic chooses in each step a vertex v_s adjacent to the border using a (ad hoc) priority function of the following parameters:

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

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

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

3 Relation to Pathwidth

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

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

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

Lemma 1 Every path-decomposition $X = (X_1, \ldots, X_r)$ can be transformed to a bijective path-decomposition $(X'_1, \ldots, X'_n)$ of no larger pathwidth.
Proof: We modify the sequence X as follows:
As long as there is some i with $start^{-1}(\{i\}) = \emptyset$, remove X_i from X; adapting the functions $start$ and end.
As long as there are $v \neq v'$ with $start(v) = i = start(v')$, define $X_{j+1} = X_j$ for all $j \geq i$ and remove v' from X_i; adapting the functions $start$ and end.
One easily observes that the finally resulting sequence X' satisfies the properties for a path-decomposition, is bijective, and has no greater pathwidth. $\square$

We call a path-decomposition with functions $start$, end **minimal** if it holds:
$end(v) \geq i \Rightarrow start(v) = i \vee \exists (v, w) \in E : \ start(w) \geq i$. (Note that the other direction is satisfied for every path-decomposition.)

Lemma 2 Every (bijective) path-decomposition X can be transformed to a minimal (bijective) path-decomposition of no larger pathwidth.

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

Lemma 3 Let $(X_1, \ldots, X_n)$ be a minimal, bijective path-decomposition of G with the functions *start* and *end*. Assume that the vertices are ordered according to their *start* values, i.e., $start(v) = s \Leftrightarrow v \in V_s \setminus V_{s-1}$. For each $s \in \{1, \ldots, n\}$ it holds: $X_s = \{v_s\} \cup B_{s-1}$.
Proof:

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

□

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

4 Reducing the Number of Partitions

The straightforward implementation of the algorithm as described in Section 2.1 computes many forests that are not part of any (or some) minimum Steiner tree. It could be advantageous to recognize such forests as soon as possible, so that their extensions are not generated in the first place. Below we describe two ideas that can be used in heuristics for recognition of such forests (partitions).

We say that a partition $\mathcal{P} = \{P_1, \ldots, P_r\}$ is **dominated** by another partition $\mathcal{Q} = \{Q_1, \ldots, Q_t\}$ at stage s iff

1. $c(s, \mathcal{Q}) \leq c(s, \mathcal{P})$,
2. $\forall P_i \in \mathcal{P} \ \exists Q_j \in \mathcal{Q}: \ P_i \subseteq Q_j$,
3. $\forall Q_j \in \mathcal{Q} \ \exists P_i \in \mathcal{P}: P_i \subseteq Q_j$.

If at any stage s, a partition $\mathcal{P}$ is dominated by another partition $\mathcal{Q}$, then the (cost of the) forest $F(s, \mathcal{P})$ need not be computed, as the following argument shows. Assume that $F(s, \mathcal{P})$ is the subgraph of some Steiner tree T when restricted to the set of vertices V_s.

Because of 2. and 3., T restricted to $V \setminus V_s$ completes the forest $F(s, \mathcal{Q})$ to a connected graph H that spans all terminals in $V \setminus V_s$. Since $F(s, \mathcal{Q})$ spans all terminals in V_s, H spans all terminals in V. Finally, because of 1., the cost of H is not larger than the cost of T. So we can do without $F(s, \mathcal{P})$ once we have $F(s, \mathcal{Q})$.

Another approach uses the idea of bound-based reductions. The idea is to show that a lower bound for any Steiner tree containing a certain forest exceeds a (heuristically computed) upper bound. For fast computation of lower bounds for individual forests, one can use a method based on reduced costs (for details see [10, 11]). More precisely, we use a dual feasible solution of value *lower* for the directed cut relaxation of the problem and corresponding reduced costs c'. In the directed version $\vec{T}$ of any minimum Steiner tree, each of the trees in the current forest must be contained in $\vec{T}$ as a directed subtree rooted at one of its nodes in the current border. Besides, there must be a path from the root z_1 of $\vec{T}$ to the root of each of these trees. Let d' denote the distance function in G with respect to the reduced cots c'. Let $\vec{T}_i^v$ be the rooted version of a tree T_i corresponding to a partition set P_i when rooted at the node $v \in P_i$. For any choice T_j in the current forest $F = \{T_1, \ldots, T_l\}$, one easily observes that the following value $lower_j$ is a lower bound for the cost of any Steiner tree containing F: $lower_j := \min\{d'(z_1, v) + c'(\vec{T}_j^v) \mid v \in P_j\} + \sum_{i,1 \leq i \neq j \leq l} \min\{c'(\vec{T}_i^v) \mid v \in P_i\} + lower$. One can additionally choose j such that $lower_j$ is maximized. If such a lower bound exceeds a known upper bound, the forest F (and all its extensions) can be excluded from further consideration.

5 Empirical Behaviour

In this section, we illustrate the empirical impact of the presented algorithm by giving an example of its application. The goal here is not to perform an experimental study, but to show how this algorithm is used in our program package for dealing with the Steiner problem in networks. Note that an approach like the one in this paper cannot be expected to be usable as a stand-alone solution method for a wide range of instances; however, it can be helpful as a subroutine in some cases. Here we demonstrate this on an instance where such a subroutine has played a decisive role. All results were produced on a PC with an AMD Athlon XP 1800+ (1.53 GHz) processor and 1 GB of main memory.

In Figure 1, the instance fl1400fst from the benchmark library SteinLib [7] is visualized. This instance has been generated by applying an FST-generation program [15] to a point set originating from some application. Our program package solved this instance for the first time, without branching and in 118 seconds. This is done in many stages, applying different subroutines. In Figure 2, we see the modified instance after applying a series of reduction methods, as described in [10, 11]. Then the program uses some partitioning techniques, described in [13], to focus on different components. In Figure 3, we have zoomed into the upper right component of Figure 2. For each such component, a heuristic checks if the algorithm in this paper should be tried. This is the case for the illustrated component, and it turns out that the maximum border size b is 9 for the corresponding subgraph. The pure dynamic programming implementation (without reductions) generates a total of

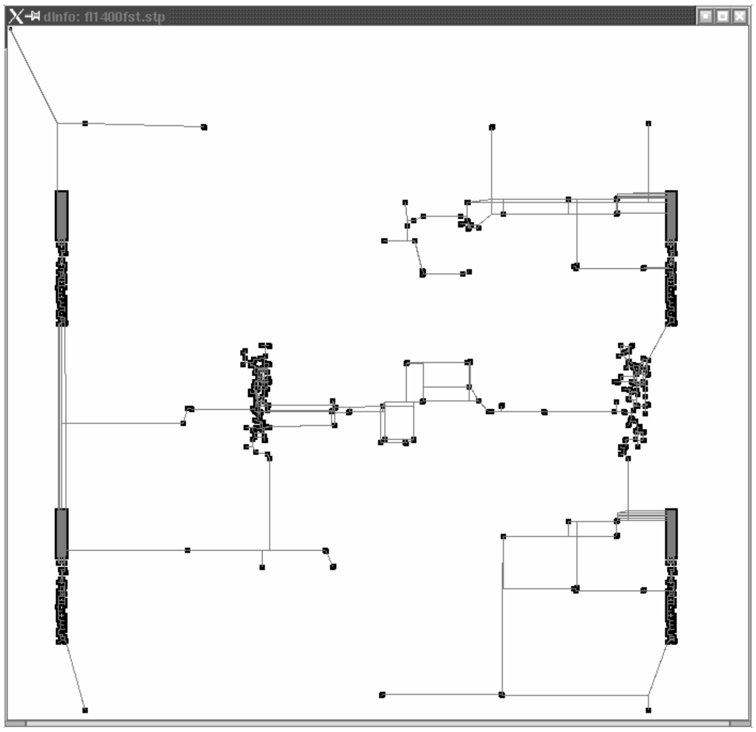

Figure 1: the instance fl1400fst ($|V| = 2694,\ |E| = 4546,\ |R| = 1400$)

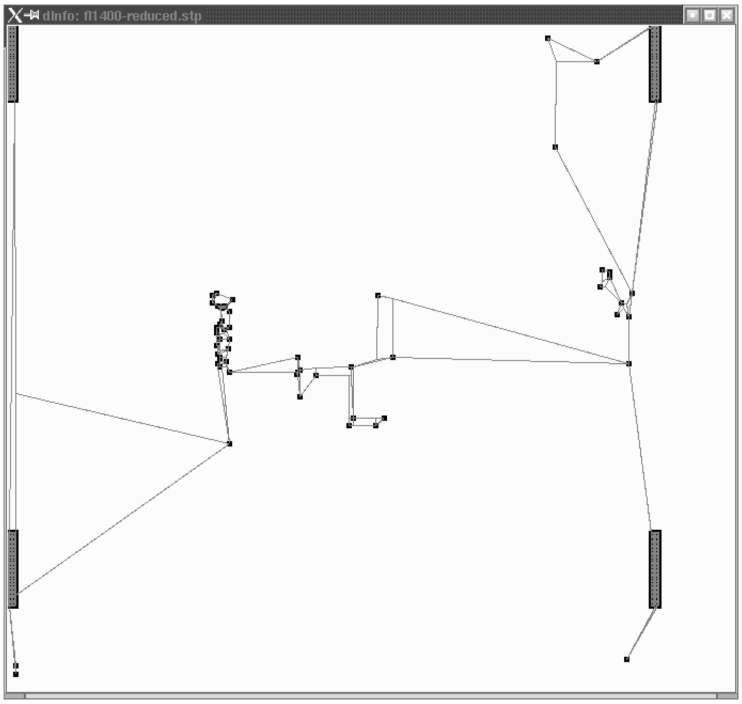

Figure 2: fl1400fst, reduced ($|V| = 1871,\ |E| = 3474,\ |R| = 937$)

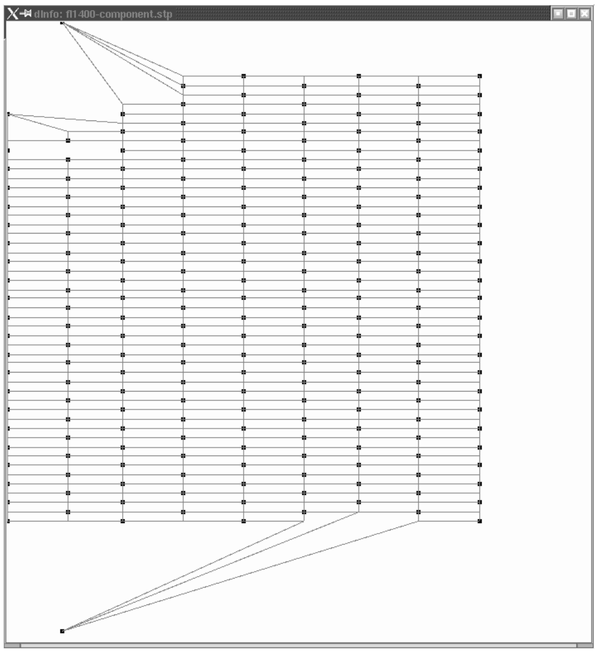

Figure 3: fl1400fst, a component ($|V| = 427,\ |E| = 796,\ |R| = 215$)

1970173 partitions; it solves the subinstance in 27 seconds. Without using this algorithm, the program based on the (classical) directed cut relaxation went into extensive branching, generating thousands of nodes in the branch-and-bound tree after one day with no termination in sight. The same situation happens when trying to solve (the directed cut/flow formulation of) this subinstance using the MIP optimizer of CPLEX (version 7.0).
Experimenting with the heuristics for reducing the number of partitions (Section 4), it turns out that their impact is not decisive, in the sense that the tractable width cannot be increased by these heuristics significantly. The dominance rule typically eliminates ca. 30% of partitions, but an efficient implementation is not easy, so it is not clear if it could pay off. The bound-based reductions suffer from the fact that for those instances where this algorithm is the approach of choice, the bounds tend to be rather weak; so not enough forests can be ruled out in this way to achieve a major overall improvement.

6 Concluding Remarks

A possible path for future work is the extension of the methods used in this paper to the more general notion of treewidth (or branchwidth). But we found this approach only modestly motivated: First, the resulting algorithm would most probably be more complicated (and less practical) than the method presented here. Second, in our concrete applications of the presented technique (usually some kind of grids) the treewidth (branchwidth) turned out to be equal to (or at least not substantially smaller than) the pathwidth, so no gain was to be expected from this alternative.

References

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

[2] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[3] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[4] H. L. Bodlaender. Personal communication, 2002.

[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] 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] A. Pönitz and P. Tittmann. Computing network reliability in graphs of restricted pathwidth. Technical report, Hochschule Mittweida, 2001.

[10] 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.

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

[12] 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.

[13] T. Polzin and S. Vahdati Daneshmand. Partitioning techniques for the Steiner problem. Research Report MPI-I-2001-1-006, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, 2001.

[14] H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[15] 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.