Separator-Based Pruned Dynamic Programming for Steiner Tree

Yoichi Iwata
National Institute of Informatics
yiwata@nii.ac.jp

Takuto Shigemura
The University of Tokyo
sigma@is.s.u-tokyo.ac.jp

Abstract

Steiner tree is a classical NP-hard problem that has been extensively studied both theoretically and empirically. In theory, the fastest approach for inputs with a small number of terminals uses the dynamic programming, but in practice, state-of-the-art solvers are based on the branch-and-cut method. In this paper, we present a novel separator-based pruning technique for speeding up a theoretically fast DP algorithm. Our empirical evaluation shows that our pruned DP algorithm is quite effective against real-world instances admitting small separators, scales to more than a hundred terminals, and is competitive with a branch-and-cut solver.

Introduction

For an undirected graph $G = (V, E)$ and a terminal set $A \subseteq V$, a tree in G is called a *Steiner tree* if it connects all the terminals in A. An input to the Steiner tree problem is an undirected graph $G = (V, E)$ with an edge-weight function $w : E \to \mathbb{R}_{>0}$ and a terminal set $A \subseteq V$, and the task is to find a Steiner tree with the minimum total weight. We use n, m, and k to denote the number of vertices, edges, and terminals, respectively.

The Steiner tree problem has been extensively studied both theoretically and empirically. In theoretical studies, Steiner tree is known as one of the Karp's 21 NP-complete problems and has been studied from various theoretical viewpoints including approximation algorithms and fixed-parameter-tractable algorithms. In empirical studies, Steiner tree is known to have many applications in various fields including the VLSI design of microchips (Held et al. 2011), the design of fiber-optic networks (Leitner et al. 2014), keyword search in relational databases (Ding et al. 2007), and team formulation in social networks (Lappas, Liu, and Terzi 2009)[1]. Due to its practical importance, in recent years, two competitions of Steiner tree solvers, the 11th DIMACS Implementation Challenge (2014) and the 3rd Parameterized Algorithms and Computational Experiments (PACE) Challenge (2018), were held, and development of practically fast Steiner tree solvers has attracted a lot of attention.

In theory, the Steiner tree problem is known to be *fixed parameter tractable (FPT)* parameterized by the number of terminals k; i.e., it can be solved in $f(k)\mathrm{poly}(n)$ time for some computable function f. This means that, although the problem is NP-hard in general, it is easy when the number of terminals is small. The first FPT algorithm was obtained by Dreyfus and Wagner (Dreyfus and Wagner 1971) and has a time complexity $\mathcal{O}(3^k nm)$. This algorithm has been improved in two directions: in terms of the $\mathrm{poly}(n)$ factor, the current fastest FPT algorithm runs in $\mathcal{O}(3^k n + 2^k(m + n \log n))$ time (Erickson, Monma, and Veinott 1987), and in terms of the $f(k)$ factor, the current fastest one runs in $\mathcal{O}((2+\epsilon)^k n^{g(\epsilon)})$ time for any $\epsilon > 0$ for some function g (Fuchs et al. 2007).

All of these FPT algorithms solve the problem as follows. Let T be a minimum Steiner tree for terminals A and pick a terminal $u \in A$. If u has degree one in T, T is the union of the unique incident edge uv and the minimum Steiner tree for terminals $(A \setminus \{u\}) \cup \{v\}$. If u has degree at least two, T is the union of the minimum Steiner tree for $A_1 \cup \{u\}$ and the minimum Steiner tree for $A_2 \cup \{u\}$ for some partition $A = A_1 \cup A_2 \cup \{u\}$. Conversely, any minimum Steiner tree can be obtained by recursively applying these two operations, and therefore, we can solve the problem by computing a minimum Steiner tree for terminals $S \cup \{u\}$ for every subset $S \subseteq A$ and every vertex $u \in V$ in a bottom-up from small to large by using the dynamic programming (DP). This is why the dependence on k is exponential. It has been theoretically proved that avoiding the exponential dependence on k is difficult; under the Set Cover Conjecture, the Steiner tree with k terminals cannot be solved in $(2-\epsilon)^k\mathrm{poly}(n)$ time for any $\epsilon > 0$ (Cygan et al. 2016), and under the Exponential-Time Hypothesis, it cannot be solved in $2^{o(k)}\mathrm{poly}(n)$ time even for planar graphs (Marx, Pilipczuk, and Pilipczuk 2017).

In practice, due to the exponential dependence on k, the FPT algorithms have been used only against inputs with a small number of terminals (usually less than ten). For solving general inputs, the state-of-the-art approach is a com-

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]The latter two applications use the following variant of Steiner tree, called *group Steiner tree*: given a set of groups of vertices instead of the terminals, find a minimum-weight connected tree containing at least one vertex from each group. We can solve this variant by a reduction to the standard Steiner tree (for details, see, e.g., (Zachariasen and Rohe 2003) or (Gamrath et al. 2017)). Actually, the datasets we use in the experiments contain instances obtained by this reduction.

bination of sophisticated preprocessing using various reduction rules (Duin 2000; Polzin 2004; Daneshmand 2004) and the branch-and-cut method using powerful MIP solvers with various Steiner-tree-specific features such as fast cutting plane generations and primal/dual heuristics. In contrast to the FPT algorithms, this approach has no theoretical worst-case analysis better than the trivial $2^n \mathrm{poly}(n)$; however, it is quite powerful in practice. Actually, all the four solvers submitted to the exact SPG (classical Steiner problem in graphs) track of the 11th DIMACS challenge used the branch-and-cut method (Gamrath et al. 2017; Fischetti et al. 2017; Althaus and Blumenstock 2014). Although there have been studies for speeding up the FPT algorithms by using the best-first search (Ding et al. 2007) or the A^* search (Hougardy, Silvanus, and Vygen 2017), they are still limited to instances with several tens of terminals.

One of the goals of the PACE challenge is to investigate the practical applicability of algorithmic ideas from FPT algorithms, and Steiner tree problem was chosen for the 3rd challenge. The challenge consists of one heuristic track and two exact tracks; one is for inputs with a small number of terminals and the other is for inputs with small tree-width. Here, tree-width is a famous sparsity measure of graphs. A variety of real-world graphs have small tree-width (e.g., planar graphs with n vertices have tree-width $O(\sqrt{n})$), and various problems, including Steiner tree, are known to be fixed-parameter tractable parameterized by tree-width (Bodlaender et al. 2015; Fafianie, Bodlaender, and Nederlof 2015).

The main contribution of this paper is presenting a novel separator-based pruning technique for speeding up the FPT algorithm for Steiner tree while keeping its theoretical worst-case bound. Our algorithm can implicitly exploit an existence of small separators in real-world instances. We conduct experiments using the benchmark datasets used in the DIMACS and the PACE challenges. The experimental results show that the FPT algorithm with the proposed pruning can solve not only instances with a small number of terminals but also large sparse real-world instances with more than a hundred terminals. The proposed algorithm is not only faster than the existing speeding up method of the FPT algorithm but also competitive with a branch-and-cut solver; there are many instances (especially, small-terminals or sparse instances) for which our algorithm is better but there also exist many instances (especially, large-terminals or dense instances) for which the branch-and-cut solver is better. In the PACE challenge, our solver using the proposed algorithm won the 1st in the track 1 (small number of terminals) and the 2nd in the track 2 (small tree-width). Our implementation is available on GitHub[2].

The idea behind our pruning comes from an existing polynomial-time algorithm for a special case of Steiner tree such that the input graph is planar and all the terminals are on a single face. In this case, we can improve the running time of the FPT algorithm to $\mathcal{O}(k^3 n + k^2 n \log n)$ as follows (Erickson, Monma, and Veinott 1987). We number the terminals from 0 to $k-1$ along the face they locate on (see Figure 1). Let T be a minimum Steiner tree for the terminals A. Because the graph is planar and all the terminals are on a single face, we can see that for any two terminals $s, t \in A$, the path between s and t on T *separates* the terminals $\{s+1, \ldots, t-1\}$ from the others. Let $u \in V(T)$ be a vertex on T. By splitting T at u, we can cut out a minimum Steiner tree for terminals $S \cup \{u\}$ for some subset $S \subseteq A$. By the above property, we can assume that S induces a consecutive interval of A; i.e., we can write $S = \{s, s+1, \ldots, t-1, t\}$ for some integers $s, t \geq 0$ (when $s > t$, we use modulo k). Therefore, in the dynamic programming, it is sufficient to compute a minimum Steiner tree for terminals $S \cup \{u\}$ only for every such consecutive interval $S \subseteq A$. As the number of such intervals is $O(k^2)$, we obtain the above running time.

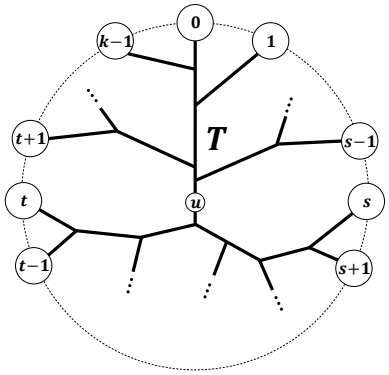

Figure 1: All the terminals $A = \{0, 1, \ldots, k-1\}$ are on the unbounded face. By splitting a Steiner tree T at $u \in V(T)$, we obtain two Steiner trees for terminals $\{s, s+1, \ldots, t\} \cup \{u\}$ and $\{t+1, t+2, \ldots, s-1\} \cup \{u\}$.

We can apply this idea for general inputs as follows. Suppose that we have computed a minimum Steiner tree T for terminals $S \cup \{u\}$ for some subset $S \subseteq A$ and $u \in V$ by the dynamic programming. If the remaining terminals $A \setminus S$ are separated by $V(T) \setminus \{u\}$, any minimum Steiner tree for $(A \setminus S) \cup \{u\}$ must contain some vertex in $V(T) \setminus \{u\}$. Therefore, we immediately know that the current minimum Steiner tree for $S \cup \{u\}$ cannot be extended to a minimum Steiner tree for A, and thus we can discard it from the DP table. In contrast to the previous special case, unfortunately, this condition rarely happens for general inputs. Our key observation is that, even if $V(T) \setminus \{u\}$ itself is not a separator, we can apply the pruning if there exists a separator satisfying some weaker condition. As in the previous special case, this pruning significantly reduces the number of subsets $S \subseteq A$ we need to consider (especially, for sparse graphs admitting small separators) and thus leads to a significant speedup.

Preliminaries

For an integer i, we define $[i]$ as the set $\{1, 2, \ldots, i\}$. For convenience, we assume $A = [k]$. For a subset $S \subseteq V$, we define a *Steiner tree for* S as a connected subgraph $G' = (V', E')$ of the input graph $G = (V, E)$ satisfying $S \subseteq V'$, and we denote the minimum weight of a Steiner tree for S by $\mathrm{opt}(S)$. For a subset $S \subseteq V$, a subset $C \subseteq V$ is called

[2]https://github.com/wata-orz/steiner_tree

Algorithm 1 (Erickson, Monma, and Veinott 1987)

1: $d(S,u) \leftarrow \infty$ for $\forall S \subseteq A$ and $\forall u \in V$.
2: $d(\{a\},a) \leftarrow 0$ for $\forall a \in A$.
3: **for** $S \subseteq A$ in ascending order of $|S|$ **do**
4: Initialize a priority queue Q_d on V.
5: **while** Q_d is not empty **do**
6: Pop u with the smallest $d(S,u)$ value from Q_d.
7: **for** $uv \in E$ **do**
8: $d(S,v) \xleftarrow{\min} d(S,u) + w(uv)$
9: **for** already processed $S' \subseteq A \setminus S$ **do**
10: **for** $u \in V$ **do**
11: $d(S \cup S', u) \xleftarrow{\min} d(S,u) + d(S',u)$
12: **return** $d(A,a)$ for an arbitrary $a \in A$.

an S-*separator* if $S \cap C \neq \emptyset$ or S is not connected in the graph $G[V \setminus C]$; or in other words, every Steiner tree for S contains some vertex in C. For a tree T and two vertices u and v on T, we define $b(T,u,v)$ as the maximum weight of a maximal degree-two path contained in the unique path between u and v in T[3]. Given a tree T and a vertex u, we can compute $b(T,u,v)$ for all $v \in V(T)$ in linear time by a simple DFS from u. For a variable x and a value y, we denote by $x \xleftarrow{\min} y$ an operation updating $x \leftarrow \min(x,y)$.

Classical Algorithm

We review a classical $\mathcal{O}(3^k n + 2^k(m + n\log n))$-time DP algorithm (Erickson, Monma, and Veinott 1987) described in Algorithm 1. We first create a two-dimensional table $d : 2^A \times V \to \mathbb{R}_{\geq 0}$, which is initialized as $d(\{a\},a) = 0$ for every terminal $a \in A$ and $d(S,u) = \infty$ for every other $S \subseteq A$ and $u \in V$. We store the minimum weight of a Steiner tree for $S \cup \{u\}$ we have found so far in $d(S,u)$. We compute the table by processing each subset $S \subseteq A$ one by one in ascending order of the size $|S|$. After processing a subset S, we ensure that $d(S,u)$ is equal to $\mathrm{opt}(S \cup \{u\})$ for every $u \in V$. Finally, we can solve the problem by answering $d(A,a)$ for an arbitrary terminal $a \in A$.

We process each subset $S \subseteq A$ as follows. First, we update $d(S,u)$ for every u by using Dijkstra's algorithm as follows; while there are unprocessed vertices, we pop an unprocessed vertex u with the smallest $d(S,u)$ value by using a priority queue; then for each edge $uv \in E$, we update $d(S,v) \xleftarrow{\min} d(S,u) + w(uv)$ (we can obtain a Steiner tree for $S \cup \{v\}$ by inserting the edge uv to a Steiner tree for $S \cup \{u\}$). This takes $\mathcal{O}(m + n\log n)$ time by using the Fibonacci heap. After this update, for each subset $S' \subseteq A \setminus S$ that has been already processed, we update $d(S \cup S', u) \xleftarrow{\min} d(S,u) + d(S',u)$ (we can obtain a Steiner tree for $S \cup S' \cup \{u\}$ by merging two Steiner trees for $S \cup \{u\}$ and $S' \cup \{u\}$). This takes $\mathcal{O}(2^{k-|S|}n)$ time. Thus the total running time is $\sum_{S \subseteq A}(m + n\log n + 2^{k-|S|}n) = \mathcal{O}(3^k n + 2^k(m + n\log n))$.

[3]Suppose that the path between $u = v_0$ and $v = v_\ell$ is $(v_0, v_1, \ldots, v_\ell)$, and among these vertices, $\{v_{i_1}, \ldots, v_{i_p}\}$ have degree at least three in T. Then $(v_0, \ldots, v_{i_1})$, $(v_{i_1}, \ldots, v_{i_2})$, …, and $(v_{i_p}, \ldots, v_\ell)$ are the set of maximal degree-two paths.

Separator-based Pruning

In order to speed up the DP algorithm, instead of computing $d(S,u)$ for every $S \subseteq V$ and $u \in V$, we compute a small portion of them while maintaining correctness of the algorithm. For a subset $S \subseteq V$, we denote by $\mathrm{valid}(S)$ the set of vertices $v \in V$ such that (S,v) is contained in the table d. Because we expect that $\mathrm{valid}(S) = \emptyset$ for a large portion of subsets $S \subseteq A$, instead of using the two-dimensional array, we use a binary search tree to efficiently maintain the table d. The key in each node is a set $S \subseteq A$ with $\mathrm{valid}(S) \neq \emptyset$, and the value is a list of $(u, d(S,u))$ for $u \in \mathrm{valid}(S)$.

For a subset $S \subseteq A$ and a vertex $u \in V$, a Steiner tree T for $S \cup \{u\}$ is called *important* if there exists a Steiner tree T' for $(A \setminus S) \cup \{u\}$ such that $T + T'$ is a minimum Steiner tree for A. We can easily see that the optimal solution can be obtained by computing only $d(S,u)$ such that the minimum Steiner tree for $S \cup \{u\}$ is important; however, it is difficult to test the importance without knowing the optimal solution itself. In our algorithm, we use the following necessary condition of the importance.

Lemma 1. *For a subset $S \subseteq A$ and a vertex $u \in V$, a Steiner tree T for $S \cup \{u\}$ is not important if there exists an $(A \setminus S)$-separator C such that, for every $v \in C$, there exists $S_v \subseteq S$ satisfying the following inequality:*

$$\mathrm{opt}(S_v \cup \{v\}) + \mathrm{opt}((S \setminus S_v) \cup \{u\}) < w(T). \qquad (1)$$

Proof. Suppose that T is important. Then there exists a Steiner tree T' for $(A \setminus S) \cup \{u\}$ such that $T + T'$ is a minimum Steiner tree for A. Because C is an $(A \setminus S)$-separator, T' must contain some vertex $v \in C$. Let T_v be a minimum Steiner tree for $S_v \cup \{v\}$ and let T_u be a minimum Steiner tree for $(S \setminus S_v) \cup \{u\}$. Then $T_v + T_u + T'$ is a Steiner tree for A satisfying $w(T_v + T_u + T') < w(T + T')$, which is a contradiction. □

In our algorithm, after processing a subset $S \subseteq A$, we ensure the following for every $u \in V$: (1) if the minimum Steiner tree for $S \cup \{u\}$ is important, (S,u) is contained in the table d and $d(S,u)$ is equal to $\mathrm{opt}(S \cup \{u\})$ and (2) if it is not important, (S,u) is not contained in d or $d(S,u)$ is at least $\mathrm{opt}(S \cup \{u\})$.

A special case of Lemma 1 is when $S_v = S$ for every $v \in C$. In this case, the inequality (1) can be simplified to $\mathrm{opt}(S \cup \{v\}) < w(T)$. We can exploit this special case as follows. For a value x, we define $C_x := \{v \mid d(S,v) \leq x\}$. Before running the for-loops at lines 9–11, we compute the minimum value x such that C_x forms an $(A \setminus S)$-separator. We then know that $\mathrm{opt}(S \cup \{v\}) \leq d(S,v) \leq x$ holds for every $v \in C_x$. Therefore we can conclude that for every $u \in V$ with $d(S,u) > x$, the corresponding Steiner tree for $S \cup \{u\}$ is not important, and thus we can safely drop (S,u) from the table.

We use the case of $S_v \neq S$ to strengthen the above pruning as follows.

Lemma 2. *For a subset $S \subseteq A$ and a vertex $u \in V$, a Steiner tree T for $S \cup \{u\}$ is not important if there exists an $(A \setminus S)$-separator C such that, for every $v \in C$, at least one of the following two conditions is satisfied.*

1. $\mathrm{opt}(S \cup \{v\}) < w(T)$, *or*
2. *there exists a vertex $s \in V(T)$ such that the distance between s and v is less than $b(T, u, s)$.*

Proof. We prove the lemma by applying Lemma 1. Let v be a vertex in C. If v satisfies the first condition, the inequality (1) holds for $S_v = S$. If v satisfies the second condition, let P be a maximum-weight degree-two path contained in the path between u and s on T (so we have $w(P) = b(T, u, s)$). By deleting P from T, we obtain a Steiner tree for $S' \cup \{s\}$ and a Steiner tree for $(S \setminus S') \cup \{u\}$ for some $S' \subseteq S$ whose total weight is $w(T) - w(P)$. Therefore, we have $\mathrm{opt}(S' \cup \{s\}) + \mathrm{opt}((S \setminus S') \cup \{u\}) \leq w(T) - w(P)$. Because we can construct a Steiner tree for $S' \cup \{v\}$ by inserting the shortest-path between s and v to a Steiner tree for $S' \cup \{s\}$, we have $\mathrm{opt}(S' \cup \{v\}) < \mathrm{opt}(S' \cup \{s\}) + w(P)$. We now have

$$\begin{aligned}
&\mathrm{opt}(S' \cup \{v\}) + \mathrm{opt}((S \setminus S') \cup \{u\}) \\
&< \mathrm{opt}(S' \cup \{s\}) + w(P) + \mathrm{opt}((S \setminus S') \cup \{u\}) \\
&\leq (w(T) - w(P)) + w(P) \\
&\leq w(T).
\end{aligned}$$

Therefore, the inequality (1) holds for $S_v = S'$. □

We now describe the entire algorithm (see Algorithm 2). We iterate only over subsets $S \subseteq A$ such that $\mathrm{valid}(S)$ is non-empty. For a vertex $u \in \mathrm{valid}(S)$, we denote by T_u the corresponding Steiner tree for $S \cup \{u\}$ of weight $d(S, u)$.

Before running Dijkstra's algorithm, we first apply the following update (lines 4–9). For each vertex $u \in \mathrm{valid}(S)$, we construct the corresponding Steiner tree T_u. Because the Steiner tree T_u is also a Steiner tree for $S \cup \{v\}$ for every $v \in V(T_u)$, we update $d(S, v) \xleftarrow{\min} d(S, u)$ for every $v \in V(T_u)$. Note that, even if T_u is important for $S \cup \{u\}$, it may not be important for $S \cup \{v\}$, and therefore the table d may not contain (S, v). We mark every such v as 'dummy' so that we can identify the corresponding Steiner tree as unimportant. Although any Steiner tree obtained by extending an unimportant Steiner tree is also unimportant, this update leads to smaller $d(S, v)$ values for unimportant Steiner trees, and therefore it is helpful for pruning.

We then update the table d by running Dijkstra's algorithm (lines 10–16). This part is almost the same as the classical algorithm without pruning. The only difference is that we propagate the 'dummy' mark.

Next, we compute a set N of vertices as follows (lines 17–27). Let p be a table initialized as follows: if v is contained in every T_u, we set $p(v) \leftarrow -\min_{u \in \mathrm{valid}(S)} b(T_u, u, v)$, and otherwise, we set $p(v) \leftarrow 0$. We update p by running Dijkstra's algorithm with the initial distance p and then set $N := \{v \mid p(v) < 0\}$. After this update, $p(v)$ is the minimum of zero and $\mathrm{dist}(s, v) - \min_{u \in \mathrm{valid}(S)} b(T_u, u, s)$ over all s contained in every T_u. Therefore, $p(v) < 0$ implies that, for every $u \in \mathrm{valid}(S)$, there exists $s \in V(T_u)$ such that $\mathrm{dist}(s, v) < b(T_u, u, s)$. Thus, the second condition of Lemma 2 is satisfied for every $v \in N$.

Algorithm 2 Separator-based Pruned DP Algorithm

```
1:  d({a}, a) ← 0 for ∀a ∈ A.
2:  for S ⊆ A with valid(S) ≠ ∅ in ascending order do
3:      dummy(u) ← false for ∀u ∈ V.
4:      for u ∈ valid(S) do
5:          T_u ← the Steiner tree for S ∪ {u}.
6:          for v ∈ V(T_u) do
7:              if d(S, v) > d(S, u) then
8:                  d(S, v) ← d(S, u)
9:                  dummy(v) ← true
10:     Initialize a priority queue Q_d on V.
11:     while Q_d is not empty do
12:         Pop u with the smallest d(S, u) value from Q.
13:         for uv ∈ E do
14:             if d(S, v) > d(S, u) + w(uv) then
15:                 d(S, v) ← d(S, u) + w(uv)
16:                 dummy(v) ← dummy(u)
17:     for v ∈ V do
18:         if v is contained in every T_u then
19:             p(v) ← −min_{u∈valid(S)} b(T_u, u, v)
20:         else
21:             p(v) ← 0
22:     Initialize a priority queue Q_p on V.
23:     while Q_p is not empty do
24:         Pop u with the smallest p(u) value from Q_p.
25:         for uv ∈ E do
26:             p(v) ←min p(u) + w(uv)
27:     N ← {v | p(v) < 0}.
28:     x ← minimum x s.t. (C_x ∪ N) is an A \ S-separator.
29:     for u ∈ V with d(S, u) > x or dummy(u) do
30:         Drop (S, u) from d.
31:     for already processed S' ⊆ A \ S do
32:         for u ∈ valid(S) ∩ valid(S') do
33:             d(S ∪ S', u) ←min d(S, u) + d(S', u)
34: return d(A, a) for an arbitrary a ∈ A.
```

We now do pruning (lines 28–30). We compute a minimum value x such that $C_x \cup N$ forms an $(A \setminus S)$-separator (or zero if N itself is an $(A \setminus S)$-separator), where $C_x := \{v \mid d(S, v) \leq x\}$. We can efficiently compute such x as follows; starting from an empty set R, we insert a vertex $v \in V \setminus N$ to R one by one in non-increasing order of $d(S, v)$; when all the vertices in $A \setminus S$ get connected in $G[R]$, we set $x := d(S, v)$ for the last inserted vertex v. Let u be a vertex with $d(S, u) > x$. For each vertex $v \in C_x$, the first condition of Lemma 2 is satisfied, and for each vertex $v \in N$, the second condition of Lemma 2 is satisfied. Therefore, we can safely drop every (S, u) with $d(S, u) > x$ from the table d by applying Lemma 2. We can also drop (S, u) for every $u \in V$ with the 'dummy' mark because we know that the corresponding Steiner tree is unimportant.

Finally, for each already processed subset $S' \subseteq A \setminus S$, we update d by merging a Steiner tree for $S \cup \{u\}$ and a

Steiner tree for $S' \cup \{u\}$ into a Steiner tree for $S \cup S' \cup \{u\}$ (lines 31–33). This part is almost the same as the classical algorithm without pruning. The only difference is how to enumerate all such S'. Because we only need to enumerate subsets $S' \subseteq A \setminus S$ with $\mathrm{valid}(S) \cap \mathrm{valid}(S') \neq \emptyset$, we use a sophisticated data structure presented in the next section.

Further Speed-up Techniques

We present two techniques for further speeding up the pruned DP algorithm.

Data Structure

We propose a binary tree data structure to maintain a set of already processed subsets $S' \subseteq A$ with $\mathrm{valid}(S') \neq \emptyset$ so that, given a subset $S \subseteq A$, we can efficiently enumerate all the subsets $S' \subseteq A \setminus S$ with $\mathrm{valid}(S) \cap \mathrm{valid}(S') \neq \emptyset$. For a node i, let L_i denote the set of leaves of the subtree rooted at i. For two subsets $S, S' \subseteq A$, we define $p(S, S')$ as the minimum integer p such that $S \cap [p]$ and $S' \cap [p]$ differ.

Each leaf t of the tree contains a set $S_t \subseteq A$. Each internal node i of the tree contains an integer k_i and two sets $I_i \subseteq A$ and $U_i \subseteq V$. Initially, the data structure consists of a single leaf t with $S_t = \emptyset$. We maintain the data structure so that the following holds for every internal node i.

1. $I_i = \cap_{t \in L_i} S_t$.
2. $U_i = \cup_{t \in L_i} \mathrm{valid}(S_t)$.
3. For all $t \in L_i$, $S_t \cap [k_i - 1]$ is the same.
4. For the left child l, $k_i \notin S_t$ for all $t \in L_l$, and for the right child r, $k_i \in S_t$ for all $t \in L_r$.

We insert a subset $S \subseteq A$ into the data structure by recursively applying the following procedure starting from the root. If the current node l is a leaf, we create a new internal node i with $I_i := S_l \cap S$, $U_i := \mathrm{valid}(S_l) \cup \mathrm{valid}(S)$, and $k_i := p(S_l, S)$; create a new leaf r with $S_r := S$; and then replace l with the node i having two children l and r. If the current node i is an internal node and $p(I_i, S) < k_i$, we create a new internal node j with $I_j := I_i \cap S$, $U_j := U_i \cup \mathrm{valid}(S)$, and $k_j := p(I_i, S)$; create a new leaf r with $S_r := S$; and then replace i with the node j having two children i and r. If the current node i is an internal node and $p(I_i, S) \geq k_i$, we update $I_i \leftarrow I_i \cap S$ and $U_i \leftarrow U_i \cup \mathrm{valid}(S)$; and then proceed to the left child if $k_i \notin S$ or to the right child if $k_i \in S$. The worst-case running time of the insertion is $\mathcal{O}(kn)$.

We enumerate subsets $S' \subseteq A \setminus S$ with $\mathrm{valid}(S) \cap \mathrm{valid}(S') \neq \emptyset$ by recursively applying the following procedure starting from the root. If the current node t is a leaf, we check the condition for S_t and add it to the candidates. If the current node i is an internal node with $I_i \cap S \neq \emptyset$ or $U_i \cap \mathrm{valid}(S) = \emptyset$, we immediately know that L_i contains no subsets $S' \subseteq A \setminus S$ with $\mathrm{valid}(S') \cap \mathrm{valid}(S) \neq \emptyset$. Therefore, we do not process its children. If the current node i is an internal node with $I_i \cap S = \emptyset$ and $U_i \cap \mathrm{valid}(S) \neq \emptyset$, we recursively process the two children of i. The worst-case running time of the enumeration is $\mathcal{O}(\min(2^{k-|S|}, |\mathcal{L}|)n)$, where $\mathcal{L}$ is the set of subsets inserted into the data structure.

Note that our data structure is completely useless for the unpruned version because $\mathrm{valid}(S) = V$ holds for all $S \subseteq A$.

Meet in the Middle

We use the following folklore lemma about the existence of a balanced separation of a tree (for the proof, see, e.g., (Fomin et al. 2018)) to show that we can obtain a minimum Steiner tree by merging three Steiner trees for small subsets.

Lemma 3. *Let T be a tree and $\mu : V(T) \to \mathbb{R}_{\geq 0}$ be a non-negative vertex weight function. Then there exists a vertex $u \in V(T)$ such that $\mu(V(C)) \leq \mu(V(T))/2$ for every connected component C of $T - u$.*

Corollary 1. *Let T be a minimum Steiner tree for A. When $|A| \geq 3$, there exists a vertex $u \in V(T)$ and a partition $A = S_1 \cup S_2 \cup S_3$ with $1 \leq |S_1| \leq |S_2| \leq |S_3| \leq |A|/2$ such that T is a union of Steiner trees for $S_1 \cup \{u\}$, $S_2 \cup \{u\}$, and $S_3 \cup \{u\}$.*

Proof. By applying Lemma 3 against $\mu : V \to \{0, 1\}$ such that $\mu(u) = 1 \iff u \in A$, we obtain a vertex $u' \in V(T)$ such that each connected component of $T - u'$ contains at most $|A|/2$ terminals. Let u be a vertex of degree at least three or contained in A that is nearest to u' on T. Then, we obtain a partition $A = S_1 \cup \ldots \cup S_d$ with $d \geq 3$ and $1 \leq |S_i| \leq |A|/2$ for each S_i such that T is a union of Steiner trees for $S_1 \cup \{u\}, \ldots$, and $S_d \cup \{u\}$. While $d \geq 4$, we pick two smallest S_i and S_j and then replace them with the union $S_i \cup S_j$. Note that because $d \geq 4$, $|S_i| + |S_j| \leq |A|/2$ holds. Finally, when d becomes three, we obtain the desired partition. □

Using this corollary, we can speed up the algorithm as follows. In the for-loop at line 2, we iterate only over subsets $S \subseteq A$ of size at most $|A|/2$. In the for-loop at line 31, we iterate only over subsets $S' \subseteq A \setminus S$ of size at most $|A| - 2|S|$. Finally, we return the minimum of $d(S, u) + d(A \setminus S, u)$ over all the processed S and $u \in \mathrm{valid}(S)$.

Lemma 4. *The above speedup is correct.*

Proof. Let S_1, S_2, S_3 be the subsets in Corollary 1. Because the size of each subset is at most $|A|/2$, the algorithm correctly computes a minimum Steiner tree for $S_i \cup \{u\}$ for every $i \in [3]$. Because $\mathrm{opt}(S_1 \cup S_2 \cup \{u\}) = \mathrm{opt}(S_1 \cup \{u\}) + \mathrm{opt}(S_2 \cup \{u\})$ and we have $|S_1| \leq |A| - 2|S_2|$, it also correctly computes a minimum Steiner tree for $S_1 \cup S_2 \cup \{u\}$. Therefore, $d(S_3, u) + d(A \setminus S_3, u) = d(S_3, u) + d(S_1 \cup S_2, u)$ gives the minimum Steiner tree. □

Experimental Evaluation

We conducted experiments on a Linux server with Intel Xeon E5-2670 (2.6 GHz) which produced a score of 399 for the DIMACS benchmark. For each test, we use a single thread and limit the maximum run time by 30 minutes and the maximum memory by 6GB. In the experiments, we use the following benchmark datasets used in the DIMACS challenge and the PACE challenge.

SteinLib A collection of crafted/real-world testsets (Koch, Martin, and Voß 2000). We classify testsets into the following 7 categories.

Random (B, C, D, E, MC, I, P4Z, P6Z): random graphs.

Artificial (SP, PUC): artificial instances.

Euclidian (X, P4E, P6E): Euclidian graphs.

CrossGrid (1R, 2R): 2D/3D cross-grid graphs.

VLSI (ALUE, ALUT, DIW, DMXA, GAP, MSM, TAQ, LIN): grid graphs with holes coming from VLSI applications.

Rectilinear (ESFST, TSPFST): rectilinear graphs with L1 distances preprocessed with GeoSteiner (Warme, Winter, and Zachariasen 2000).

Group (WRP): instances obtained by a reduction from the group Steiner tree problem coming from industrial wire routing problems (Zachariasen and Rohe 2003).

Vienna Real-world instances from telecommunication networks (Leitner et al. 2014).

Copenhagen Obstacle-avoiding rectilinear Steiner tree instances preprocessed with ObSteiner (Huang and Young 2013).

PACE Public instances of the two exact tracks of the PACE challenge, each of which consists of 100 instances from SteinLib and Vienna.

Impact of Pruning

We first evaluate the impact of the pruning by comparing the performance with the unpruned version EMV ((Erickson, Monma, and Veinott 1987)) using the PACE dataset. Note that EMV is not a state-of-the-art method but a baseline algorithm. Because the performance depends on the type of instances, we will give a detailed comparison with two modern solvers later in this section. For evaluating the effect of the two speed-up techniques, we implemented three variants of the pruned DP algorithm: PRUNED(DS,MM) uses both of the two speed-up techniques, PRUNED(DS) uses the data structure but does not use the meet-in-the-middle technique, and PRUNED($\emptyset$) uses neither techniques. PRUNED($\emptyset$) keeps the processed subsets in a list and enumerates all the subsets $S' \subseteq A \setminus S$ with $\mathrm{valid}(S) \cap \mathrm{valid}(S') \neq \emptyset$ by naively testing all the subsets in the list. All the solvers were implemented using the Rust programming language.

The cactus plot in Figure 2 shows the running time of each solver. We can see that PRUNED($\emptyset$) is already quite faster than EMV. The data structure improves the performance against difficult instances, and the meet-in-the-middle leads to a uniform speedup. In the subsequent experiments, we will use PRUNED to refer to the solver PRUNED(DS,MM).

Figure 3 illustrates the effect of the number of terminals k. A point at coordinates (k, t) means that an instance with k terminals was solved in t seconds. We can see that the running time of EMV exponentially depends on k but the running time of PRUNED does not. While there are unsolved instances with only 25 terminals, there also exist solved instances with more than a thousand terminals. This is not so surprising when recalling that the idea of the proposed pruning came from the polynomial-time algorithm for special planar instances for which the number of subsets of terminals we need to consider is not $\exp(k)$ but $O(k^2)$. Hence the effectiveness of the pruning strongly depends on structures

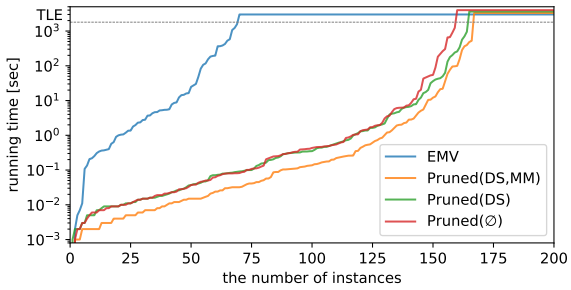

Figure 2: A cactus plot showing the running times of EMV and the three variants of PRUNED.

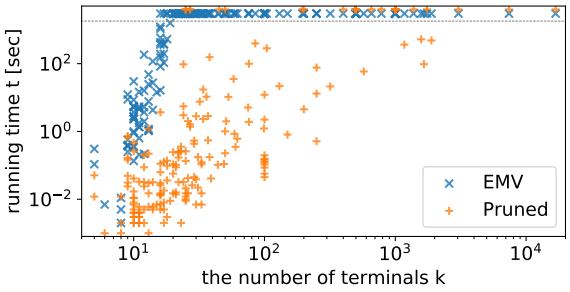

Figure 3: Effect of the number of terminals k

of graphs (e.g., we can expect that the pruning will be effective for sparse graphs admitting many small separators, but it will not be effective for dense or random graphs).

Comparison with the A^* Search

We compare the performance of PRUNED with an existing solver HSV (Hougardy, Silvanus, and Vygen 2017) using the experimental results reported in their paper. Their experiments were conducted on a computer with Intel Xeon W5590 (3.33 GHz) which produced a score of 391 for the DIMACS benchmark and their solver was implemented using C++ language. HSV combines the EMV algorithm and the A^* search as follows: instead of filling the DP table $d(S, u)$ from small $|S|$ to large $|S|$, it computes some lower bound $\mu(A \setminus S, u)$ of $\mathrm{opt}(A \setminus S, u)$ and processes the one with the smallest $d(S, u) + \mu(A \setminus S, u)$ value using a priority queue. It also uses a pruning based on the lower bound μ. Note that it seems difficult to combine the A^* search with our separator-based pruning because, in order to efficiently obtain the separator used in our pruning, we need to compute the values of $d(S, u)$ for all $u \in V$ at once. Because HSV did not use any preprocessing, we run PRUNED without preprocessing for a fair comparison.

Table 1 shows the comparison of the performance of HSV and PRUNED against datasets SteinLib, Vienna, and Copenhagen. From each dataset, we use only instances with less than 64 terminals (because HSV is limited to such instances). The column '#' shows the number of such instances in each dataset. The column 'solved' shows the number of instances solved by each solver within the time/memory limit, and the column 'better' shows the num-

Table 1: Comparison of HSV and PRUNED

		HSV		PRUNED	
Dataset	#	solved	better	solved	better
Random	418	262	0	**281**	**35**
Artificial	21	11	0	11	0
Euclidian	26	26	0	26	0
CrossGrid	45	33	0	**42**	**14**
VLSI	128	128	1	128	0
Rectilinear	93	93	0	93	4
Group	89	16	0	**87**	**82**
Vienna	6	3	0	**6**	**3**
Copenhagen	10	10	0	10	0

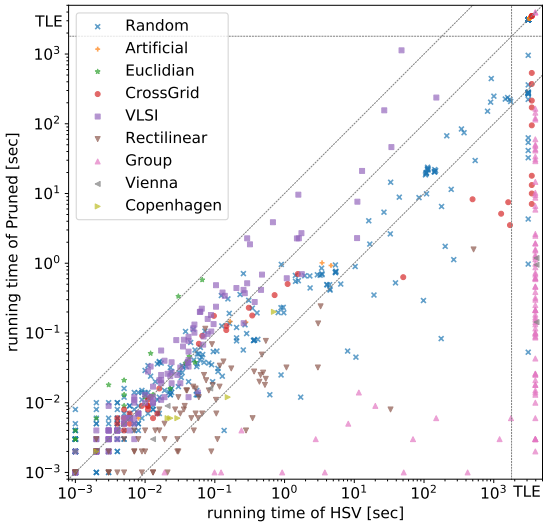

Figure 4: Running time comparison of HSV and PRUNED

Table 2: Comparison of SCIP-JACK and PRUNED

			SCIP-JACK		PRUNED	
Dataset	#	k/n	solved	better	solved	better
Random	133	.256	**96**	**91**	16	0
Artificial	53	.239	8	4	6	6
VLSI	40	.046	20	0	**32**	**27**
Rectilinear	143	.425	**138**	**41**	113	1
Group	85	.086	65	4	**79**	**58**
Vienna	210	.145	**129**	**97**	42	7
Copenhagen	12	.140	8	2	7	3

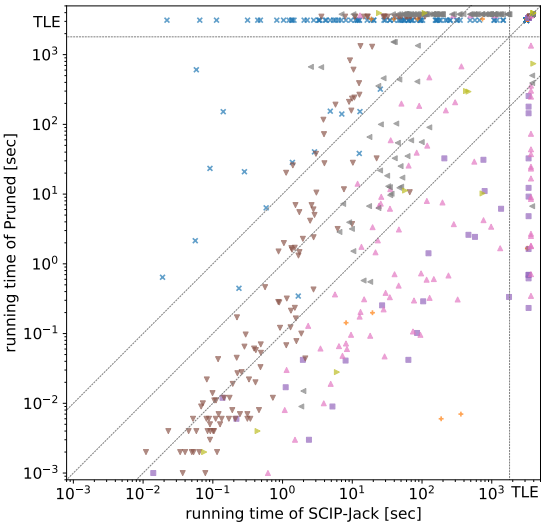

Figure 5: Running time comparison of SCIP-JACK and PRUNED. The legend is the same as in Figure 4.

ber of instances for which the solver performed significantly better than the other; we say that the performance of a solver s_1 is significantly better than a solver s_2 for an instance i if either (1) s_1 solved i but s_2 did not or (2) both of them solved i but the run time of s_2 is greater than the run time of s_1 times ten plus one second. Figure 4 illustrates the running time comparison against each instance. Each point corresponds to a single instance and its coordinates (x, y) means that HSV solved the instance in x seconds and PRUNED solved the instance in y seconds.

We can see that PRUNED outperforms HSV for datasets Random, CrossGrid, Group, and Vienna. For Random, the pruning is not so effective but A^* search is not even more effective. For the other three datasets, the pruning works effectively because they are near-planar.

Comparison with the Branch-and-Cut Method

We compare the performance of PRUNED with an open-source branch-and-cut solver SCIP-JACK (Gamrath et al. 2017). We use the latest version of SCIP-JACK which has been submitted to the PACE challenge and won the 3rd in the track 1 and the 1st in the track 2. It internally uses an MIP solver SCIP (Gleixner et al. 2018) version 5.0.1 and an LP solver SoPlex (Gleixner, Steffy, and Wolter 2012; 2015) version 3.1.1. In addition to the branch-and-cut method, SCIP-JACK uses a variety of preprocessing techniques to reduce the graph size (Rehfeldt 2015). In the following comparison, PRUNED also uses the same preprocessing as SCIP-JACK and the run time for the preprocessing is excluded for both solvers so that the difference between the pruned DP algorithm and the branch-and-cut method becomes clear.

Table 2 and Figure 5 show the comparison of the performance of SCIP-JACK and PRUNED against datasets SteinLib, Vienna, and Copenhagen. We omit all the too-easy instances for which the preprocessing alone solved the problem. The column '#' shows the number of the remaining instances in each dataset, and the column 'k/n' shows the average of k/n in the dataset. Because the theoretical worst-case running time of SCIP-JACK and PRUNED exponentially depends on n and k, respectively, we can expect that a dataset with a smaller k/n value is more advantageous to PRUNED. Note that the table does not contain Euclidian and CrossGrid because the preprocessing solved all the instances in these datasets.

As expected, the performance of PRUNED against Ran-

dom is bad because random graphs admit no small separators. PRUNED outperforms SCIP-JACK against VLSI and Group both coming from industrial applications. This is due to the following two reasons: (1) the number of terminals is relatively small for these applications (the number of terminals of an instance obtained by the reduction from the Group Steiner tree is the number of groups) and (2) because the graphs are grid with holes, they admit many small separators. On the other hand, SCIP-JACK outperforms PRUNED against geometric datasets Rectilinear and Vienna.

Acknowledgments

Yoichi Iwata is supported by JSPS KAKENHI Grant Number JP17K12643.

References

Althaus, E., and Blumenstock, M. 2014. Algorithms for the maximum weight connected subgraph and prize-collecting steiner tree problems. *11th DIMACS Implementation Challenge Workshop (Technical Report)*.

Bodlaender, H. L.; Cygan, M.; Kratsch, S.; and Nederlof, J. 2015. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.* 243:86–111.

Cygan, M.; Dell, H.; Lokshtanov, D.; Marx, D.; Nederlof, J.; Okamoto, Y.; Paturi, R.; Saurabh, S.; and Wahlström, M. 2016. On problems as hard as CNF-SAT. *ACM Trans. Algorithms* 12(3):41:1–41:24.

Daneshmand, S. V. 2004. *Algorithmic approaches to the Steiner problem in networks*. Ph.D. Dissertation, Universität Mannheim.

Ding, B.; Yu, J. X.; Wang, S.; Qin, L.; Zhang, X.; and Lin, X. 2007. Finding top-k min-cost connected trees in databases. In *ICDE*, 836–845.

Dreyfus, S. E., and Wagner, R. A. 1971. The steiner problem in graphs. *Networks* 1(3):195–207.

Duin, C. 2000. Preprocessing the steiner problem in graphs. In *Advances in Steiner Trees*. Springer US. 175–233.

Erickson, R. E.; Monma, C. L.; and Veinott, A. F. 1987. Send-and-split method for minimum-concave-cost network flows. *Math. Oper. Res.* 12(4):634–664.

Fafianie, S.; Bodlaender, H. L.; and Nederlof, J. 2015. Speeding up dynamic programming with representative sets: An experimental evaluation of algorithms for steiner tree on tree decompositions. *Algorithmica* 71(3):636–660.

Fischetti, M.; Leitner, M.; Ljubic, I.; Luipersbeck, M.; Monaci, M.; Resch, M.; Salvagnin, D.; and Sinnl, M. 2017. Thinning out steiner trees: a node-based model for uniform edge costs. *Math. Program. Comput.* 9(2):203–229.

Fomin, F. V.; Lokshtanov, D.; Saurabh, S.; Pilipczuk, M.; and Wrochna, M. 2018. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Trans. Algorithms* 14(3):34:1–34:45.

Fuchs, B.; Kern, W.; Mölle, D.; Richter, S.; Rossmanith, P.; and Wang, X. 2007. Dynamic programming for minimum steiner trees. *Theory Comput. Syst.* 41(3):493–500.

Gamrath, G.; Koch, T.; Maher, S. J.; Rehfeldt, D.; and Shinano, Y. 2017. SCIP-Jack - a solver for STP and variants with parallelization extensions. *Math. Program. Comput.* 9(2):231–296.

Gleixner, A.; Bastubbe, M.; Eifler, L.; Gally, T.; Gamrath, G.; Gottwald, R. L.; Hendel, G.; Hojny, C.; Koch, T.; Lübbecke, M. E.; Maher, S. J.; Miltenberger, M.; Müller, B.; Pfetsch, M. E.; Puchert, C.; Rehfeldt, D.; Schlösser, F.; Schubert, C.; Serrano, F.; Shinano, Y.; Viernickel, J. M.; Walter, M.; Wegscheider, F.; Witt, J. T.; and Witzig, J. 2018. The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin.

Gleixner, A. M.; Steffy, D. E.; and Wolter, K. 2012. Improving the accuracy of linear programming solvers with iterative refinement. In *ISSAC*, 187–194.

Gleixner, A. M.; Steffy, D. E.; and Wolter, K. 2015. Iterative refinement for linear programming. Technical Report 15-15, ZIB, Takustr. 7, 14195 Berlin.

Held, S.; Korte, B.; Rautenbach, D.; and Vygen, J. 2011. Combinatorial optimization in VLSI design. In *Combinatorial Optimization - Methods and Applications*. IOS Press. 33–96.

Hougardy, S.; Silvanus, J.; and Vygen, J. 2017. Dijkstra meets steiner: a fast exact goal-oriented steiner tree algorithm. *Math. Program. Comput.* 9(2):135–202.

Huang, T., and Young, E. F. Y. 2013. Obsteiner: An exact algorithm for the construction of rectilinear steiner minimum trees in the presence of complex rectilinear obstacles. *IEEE Trans. on CAD of Integrated Circuits and Systems* 32(6):882–893.

Koch, T.; Martin, A.; and Voß, S. 2000. SteinLib: An updated library on steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin.

Lappas, T.; Liu, K.; and Terzi, E. 2009. Finding a team of experts in social networks. In *KDD*, 467–476.

Leitner, M.; Ljubic, I.; Luipersbeck, M.; Prossegger, M.; and Resch, M. 2014. New real-world instances for the steiner tree problem in graphs. *Technical Report, ISOR, Uni Wien*.

Marx, D.; Pilipczuk, M.; and Pilipczuk, M. 2017. On subexponential parameterized algorithms for steiner tree and directed subset TSP on planar graphs. *CoRR* abs/1707.02190.

Polzin, T. 2004. *Algorithms for the Steiner problem in networks*. Ph.D. Dissertation, Saarland University.

Rehfeldt, D. 2015. A generic approach to solving the steiner tree problem and variants. Master's thesis, Technische Universität Berlin.

Warme, D.; Winter, P.; and Zachariasen, M. 2000. Exact algorithms for plane Steiner tree problems: A computational study. In Du, D.-Z.; Smith, J.; and Rubinstein, J., eds., *Advances in Steiner Trees*. Kluwer. 81–116.

Zachariasen, M., and Rohe, A. 2003. Rectilinear group steiner trees and applications in VLSI design. *Math. Program.* 94(2-3):407–433.