Solving the Steiner Tree Problem with few Terminals

Johannes K. Fichte
Institute of Artificial Intelligence
TU Dresden
Dresden, Germany
johannes.fichte@tu-dresden.de

Markus Hecher
Databases and Artificial Intelligence Group
TU Wien
Vienna, Austria
hecher@dbai.tuwien.ac.at

André Schidler
Algorithms and Complexity Group
TU Wien
Vienna, Austria
aschidler@ac.tuwien.ac.at

Abstract—The Steiner tree problem is a well-known problem in network design, routing, and VLSI design. Given a graph, edge costs, and a set of dedicated vertices (terminals), the Steiner tree problem asks to output a sub-graph that connects all terminals at minimum cost. A state-of-the-art algorithm to solve the Steiner tree problem by means of dynamic programming is the Dijkstra-Steiner algorithm. The algorithm builds a Steiner tree of the entire instance by systematically searching for smaller instances, based on subsets of the terminals, and combining Steiner trees for these smaller instances. The search heavily relies on a guiding heuristic function in order to prune the search space. However, to ensure correctness, this algorithm allows only for limited heuristic functions, namely, those that satisfy a so-called consistency condition.

In this paper, we enhance the Dijkstra-Steiner algorithm and establish a revisited algorithm, called DS*. The DS* algorithm allows for arbitrary lower bounds as heuristics relaxing the previous condition on the heuristic function. Notably, we can now use linear programming based lower bounds. Further, we capture new requirements for a heuristic function in a condition, which we call admissibility. We show that admissibility is indeed weaker than consistency and establish correctness of the DS* algorithm when using an admissible heuristic function. We implement DS* and combine it with modern preprocessing, resulting in an open-source solver (DS Solve). Finally, we compare its performance on standard benchmarks and observe a competitive behavior.

I. INTRODUCTION

The term Steiner tree problem on graphs encompasses a class of graph problems that all ask to connect specific vertices of a graph, so-called terminals, at minimum cost. Connecting the vertices usually requires to select edges of a given graph that form a tree and the resulting solution is called a Steiner tree. While numerous different definitions exist [1], we consider the minimum Steiner tree problem (STP), where costs are defined by positive integers given at the edges. The total cost is simply the sum of costs of selected edges and one aims to minimize the total cost [2].

STP is long known to be computationally hard, i.e., NP-hard [3]. Despite the longevity and the large body of existing research, solving STP is still an active research topic [4]–[8]. Interest stems from applications in various fields such as the construction of evolutionary trees in phylogeny [9], network design [10], routing problems [11], and VLSI design [12]. Over time, many heuristic and exact solving algorithms and techniques have been developed [2]. One of the longest known algorithms for STP is the Dreyfus-Wagner algorithm [13]. This algorithm implements dynamic programming with runtime bounded exponentially in the number of terminals and polynomially in the number of vertices. In theory, the Dreyfus-Wagner algorithm is fast as long as the number of terminals remains small. However, in practice, the involved constants are far from optimal and the runtime of the algorithm increases too quickly as the algorithm exhaustively enumerates all sub-solutions. This can be avoided and the runtime improved by replacing enumeration with graph search, resulting in the Dijkstra-Steiner algorithm [5]. Its underlying idea is similar to the well-known A* algorithm [14]. One uses the Dijkstra’s algorithm to navigate the search space and guide the search using a heuristic function. In practice, the heuristic function is crucial for the performance of the algorithm. Unfortunately, the heuristic needs to satisfy a strong condition, namely, providing consistent lower bounds, as otherwise correctness cannot be assured. Consistency guarantees that the lower bound is monotonous and the estimate will not decrease in later iterations of the Dijkstra-Steiner algorithm.

Contributions. In this paper, we revisit the Dijkstra-Steiner algorithm and enable heuristic functions for general lower bounds relaxing the previously known consistency condition. We formalize new requirements in a condition called admissibility. We prove that admissibility is weaker than consistency and allows for better heuristic functions. In particular, we can now employ linear programming based approximations as heuristics. We show that our approach still ensures correctness. We implement the new algorithm into a fully fledged solver, DS Solve, for STP. Our solver complements the solving algorithm with modern preprocessing, upper bound heuristics, and local optimization. Finally, we present experimental results where we compare our solver to state-of-the-art STP solvers. Our experiments show that our revisited algorithm with the new heuristic significantly improves the runtime.

Algorithms and latest implementations. The Dreyfus-Wagner algorithm [13] was generalized by Erickson, Momma, and
Veinott [15]. While the algorithm is exponential in the number of terminals with a basis of 3, it was very recently implemented into the solver Prun showing good performance on certain benchmarks [6]. The Dijkstra-Steiner algorithm [5] is implemented in the solvers Jagiellonian [16] and HSV [5]. We also implemented it into our solver to measure the difference with our revisited algorithm and heuristic. While our approach tackles instances with a limited number of terminals, there are also dynamic programming algorithms that employ low treewidth or rank [17], [18]. The solvers wata_sigma [19], Tom [20], and FIT CTU [21] implement such algorithms and showed successful results in the PACE 2018 challenge.

Another approach is branch-and-cut, which is based on linear programming using existing solvers for (mixed) integer linear programs (MILP). Various solvers are available that perform very well on arbitrary STP instances and can mainly be distinguished by the used linear programming model. Two notable solvers are mozaicballs [4] and SCIP-Jack [22].

Challenges. In 2014, the 11th DIMACS implementation challenge was dedicated to different variants of the Steiner tree problem [23]. The 3rd Parameterized Algorithms and Computational Experiments challenge (PACE 2018) addressed the minimum Steiner tree problem and featured three tracks, namely, a) where instances were limited in the number of terminals, b) where instances were limited in the treewidth, and c) that allowed to submit heuristics [24]. Solvers in these competitions mainly used two different techniques: dynamic programming and branch-and-cut.

II. PRELIMINARIES

We assume familiarity with standard notions in computational complexity [25] and let \( \mathbb{N} \) be the set of positive integers.

Graphs, Networks, and Steiner Trees. For basic terminology on graphs, we refer to the literature [26]. In particular, we use \( G = (V,E) \) to denote an undirected, connected graph, or graph for short, where \( V \) is a set of vertices and \( E \subseteq \{ \{u,v\} \mid u,v \in V \} \) is a set of edges. Given graphs \( G_1 = (V_1,E_1) \) and \( G_2 = (V_2,E_2) \), \( G_1 \) is a sub-graph of \( G_2 \) if \( V_1 \subseteq V_2 \) and \( E_1 \subseteq E_2 \). Given a graph \( G = (V,E) \) and a vertex \( u \in V \). Then, the set \( \delta_G(u) \) of incident edges is given by \( \delta_G(u) := \{ \{u,v\} \mid \{u,v\} \in E \} \) and \( |\delta_G(u)| \) refers to the degree of \( u \). A tree is a graph that is acyclic and connected, i.e., for every pair of vertices of the graph there is a path between them. Further, we define an undirected network, or network for short, by \( N := (V,E,\sigma) \), where graph \( (V,E) \) is connected and \( \sigma : E \to \mathbb{N} \) is a total mapping, called edge costs, which assigns to each edge \( e \) some integer, called cost of \( e \). Let \( N := (V,E,\sigma) \) be a network and let \( G = (V',E') \) be a sub-graph of \( (V,E) \). We let the costs of \( G \) in \( N \) be \( c_{N}(G) := \sum_{\{u,v\} \in E} \sigma(e) \). Then, we denote by \( d_N(u,v) \) the distance between two vertices \( u \) and \( v \in V \) in \( N \), i.e., \( d_N(u,v) \) is the length of the shortest path from \( u \) to \( v \). Formally, \( d_N(u,v) := \min_{P \in \mathcal{P}} |P| \) where \( \mathcal{P} \) is a connected sub-graph of \( G \) that contains \( u \) and \( v \).

Further, we define the distance network of \( V' \) for \( N \) by \( D_N(V') := (V',\{\{u,v\} \mid u,v \in V'\},\sigma') \), where \( \sigma'(\{u,v\}) := d_N(u,v) \) for \( u,v \in V' \).

Example 1. Figure I (left) illustrates network \( N \), which we use throughout the paper. Circles represent vertices, lines are edges, and edge costs are given by the rectangles. Further, terminals are denoted by lowercase letters. Figure I (middle) provides the distance network \( D_N((a,b,c,d)) \) for \( N \).

An instance \( I \) of the minimum Steiner Tree Problem (STP) is of the form \( I := (N,R) \), where \( N = (V,E,\sigma) \) is a network and \( R \subseteq V \) is a non-empty set of vertices, called terminals. A Steiner tree for \( I \) is a rooted tree \( T := (V',E',r) \), where \( V' \subseteq V, R \subseteq V', E' \subseteq E, \) and \( r \in R \). We call \( r \) the root. \( T(I) \) refers to the set of Steiner trees for \( I \). The tree \( T \) is called an SMT (Steiner minimal tree), if \( c_{N}(T) = \min_{T' \in T(I)} c_{N}(T') \). Let \( \text{csmt}(I) \) be the set of SMTs for \( I \) and let \( \text{csmt}(N) := c_{N}(T) \) for any \( T \in \text{csmt}(I) \), called the csmt (Steiner minimal tree costs). The tree \( T \) is a minimum spanning tree of network \( N \), if \( T \) is an SMT for \((N,V)\).

Example 2. Figure I (right) depicts the SMT for our instance \( I := (N,R) \) from Example 1, where \( N \) is given (left). The set \( R \) of terminals consists of the vertices \( R = \{a,b,c,d\} \) in lower case letters (left).

Dijkstra’s algorithm and the \( \text{A}^* \) algorithm. Given a network \( N = (V,E,\sigma) \). Dijkstra’s algorithm is used to find the shortest path from vertex \( u \) to vertex \( v \) in \( N \). In the following, we briefly mention the ideas of this algorithm. For details, we refer to the literature [27]. Dijkstra’s algorithm maintains a queue \( Q \) and for each vertex \( w \in V \) a distance \( l(w) \) between vertices \( u \) and \( w \). Initially, \( Q = \{u\} \) and the distances \( l(\cdot) \) are assumed to be \( \infty \), except for \( l(u) \), where the distance is known to be 0. In each iteration the algorithm removes the vertex \( v \) from \( Q \) that minimizes \( l(w) \). Then, the algorithm expands \( w \): For each \( (w,x) \in \delta_{(V,E)}(w) \), the value \( l(w) + \sigma((w,x)) \) is computed. Whenever \( l(w) + \sigma((w,x)) < l(x) \), vertex \( x \) is added to \( Q \) and the (smaller) distance \( l(x) \leftarrow l(w) + \sigma((w,x)) \) is kept for \( x \). Whenever a node \( w \) is removed from \( Q \), it holds that \( l(w) = d_N(u,w) \). Therefore, the algorithm stops as soon as \( v \) is removed from \( Q \) and finds the distance in time \( \mathcal{O}(|E| + |V| \cdot \log(|V|)) \) [27].

The \( \text{A}^* \) algorithm is an extension of Dijkstra’s algorithm that uses a heuristic function \( h : V \to \mathbb{N} \) to speed up the search. Assuming \( h(x) \) is an estimate of the distance between \( u \) and \( x \), in each iteration of \( \text{A}^* \), instead of removing vertex \( w \) from \( Q \), where \( l(w) \) is minimal as in Dijkstra’s algorithm, the \( \text{A}^* \) algorithm removes \( w \), where \( l(w) + h(w) \) is minimal. We call \( h \)

- admissible, if \( h(w) \leq d_N(w,v) \) for all \( w \in V \), and
- consistent, if \( h(w) \leq \sigma((w,y)) + h(y) \) for all \( \{w,y\} \in E \).

Intuitively, admissibility of \( h \) implies that \( h \) does not “over-approximate”, i.e., it provides a lower bound, and consistency of \( h \) additionally establishes a form of triangle inequality for \( h \).

1In this paper, we use \( \infty \) as an abbreviation for \( \Sigma_{e \in E} \sigma(e) \).
While $A^*$ is correct if $h$ is admissible, polynomial runtime can only be guaranteed if the heuristic is consistent [14].

III. SOLVING THE STEINER TREE PROBLEM

In this section, we describe our advancement to the Dijkstra-Steiner (DS) algorithm. DS combines ideas from the $A^*$ and Dreyfus-Wagner algorithms. We first discuss the Dreyfus-Wagner algorithm, followed by DS. Finally, we lift DS to more general heuristic functions, which provide a lower bound on the costs, and show correctness. For this section, we assume an STP instance $I = (N, R)$ with network $N = (V, E, \sigma)$ and set $R$ of terminals, where $r \in R$ is an arbitrary terminal. The vertex $r$ is used as the root of the resulting SMT.

A. The Dreyfus-Wagner (DW) algorithm

The algorithm is motivated by the fact that any SMT for instance $I$ is guaranteed to consist of so-called sub-SMTs [13]. Given a vertex $u \in V$ and a set $I \subseteq R$, we define a sub-SMT $S$ for $(u, I)$ as an SMT for instance $(N, I \cup \{u\})$, i.e., $S \in \text{smt}(\{(N, I \cup \{u\})\})$. Further, we denote by $I^*(u, I)$ the sub-SMT costs, $I^*(u, I) = \text{csmt}(N, I \cup \{u\})$. The sub-SMTs of an SMT are vertex-disjoint apart from their corresponding roots. In other words, any SMT consists of sub-SMTs that are “joined” using their root vertices, which are referred to by join vertices. Then, intuitively, it suffices [13] to incrementally compute sub-SMTs for parts of the instance and join them accordingly. Thereby, we create larger sub-SMTs, until finally ending at an SMT for $(N, R)$.

Example 3. Recall instance $I = (N, R)$, where $R = \{a, b, c, d\}$ of Example 2. In Figure 1 (right), vertex $F$ is a join vertex for the Steiner tree and vertex $F$ joins the sub-SMTs for instances $(N, \{a, b, F\})$, $(N, \{c, F\})$, and $(N, \{d, F\})$.

Listing 1 shows DW. The sub-SMT costs are computed for $(u, I)$ with increasing cardinalities of sets $I$. Sub-SMTs for $(u, \{v\})$ (singleton of terminals) are computed using the distance between $u$ and $v$, cf., Line 2. The remaining sub-SMTs are computed in two steps. First, in step (i), a tentative cost value $l(u, I)$ is computed for every vertex $u \in V$ and each set $I$ of cardinality $l$, by combining costs $l^*(u, J)$ of sub-SMTs for $J \subseteq I$ accordingly, cf., Line 5. Intuitively, this corresponds to joining two sub-SMTs at root $u$. Then, in Step (ii), cf., Line 6, sub-SMT costs $l^*(u, I)$ are computed for each vertex $u \in V$, by propagating costs $l(v, I)$ to all vertices $v \in V$. Intuitively, Line 6 corresponds to connecting a vertex $u$ to the sub-SMT for $(v, I)$ by a path between $u$ and $v$. After the algorithm terminates, $l^*(r, R \setminus \{r\}) = \text{csmt}(N, R)$. The SMT can then be found by retracing the steps of the algorithm.

Proposition 4 ([5], [13]). Given an instance $I = (N, R)$ of STP, where $N = (V, E, \sigma)$. The DW algorithm runs in time $O(3^{|R|})$ and space $O(2^{|R|} |E|)$.

B. The Dijkstra-Steiner (DS) algorithm

In theory, the runtime of DW seems suitable for instances with a small number of terminals. In practice this algorithm consumes too much time and memory, even for about a few dozen of terminals. However, one can still lower the runtime by changing how the whole search space is explored. To this end, we first define for given instance $I = (N, R)$ of STP the Steiner search network $\mathcal{N}(I) := (V', E', \sigma')$, where the vertices are a set of pairs among $V \times 2^R$, i.e., $V' := \{(u, I), u \in V, \emptyset \subseteq I \subseteq R \setminus \{r\}\}$. Then, there is an edge $e = \{(u, I), (v, J)\}$ in $E'$ between any two distinct vertices $(u, I)$ and $(v, J)$, if we have either $(1) \{u, v\} \in \delta(V, E)(u)$ and $I = J$; or $(2) u = v$ and $\emptyset \subseteq J \subseteq I$. The cost for each edge $e \in E'$ is given as $\sigma'(e) := w$, where $w = \sigma(\{u, v\})$ in Case (1), and $w = l^*(v, I \setminus J)$ in Case (2). Since $\mathcal{N}(I)$ is a network, we can apply Dijkstra’s algorithm to $\mathcal{N}(I)$. Since we cannot construct $\mathcal{N}(I)$ as $l^*$ is not known in advance, the algorithm runs on a partial network that is dynamically amended. During expansion of a vertex (in Dijkstra’s algorithm), we consider only those neighboring tuples that are either adjacent in the network according to Case (1) or where $(u, I \setminus J)$ has been expanded before for Case (2). The improved algorithm is called the Dijkstra-Steiner (DS) algorithm [5]. DS has a worst-case runtime that is similar to DW. But DS does not expand vertices $(u, I)$, where costs $l^*(u, I) > \text{csmt}(N, R)$, which can cut down the runtime considerably in practice.
### Listing 2: The DS' algorithm extending DS [5]

In: An STP instance $I = (N, R)$, where $N = (V, E, \sigma)$, root $r \in R$, and a Steiner heuristic $h^*$

Out: Pair $(T, c_{ST}(T))$, where $T$ is an SMT for $I$

1. $R' := R \setminus \{r\}$
2. $l((u, I) \leftarrow \infty$ for all $(u, I) \in V \times 2^{R'}$
3. $l((u, \{\}) \leftarrow 0$ for all $u \in R'$
4. $l(u, \{\}) \leftarrow 0$ for all $u \in V$
5. $b(u, I) \leftarrow \emptyset$ for all $(u, I) \in V \times 2^{R'}$
6. $Q \leftarrow \{(u, \{\}) \mid u \in R'\}$
7. $P \leftarrow \emptyset$
8. while $(r, R') \notin P$

9. $(u, I) \leftarrow \min_{(u, J) \in Q} l((u, I) + h^*(u, R \setminus I)$
10. $Q \leftarrow Q \setminus \{(u, I)\}$
11. $P \leftarrow P \cup \{(u, I)\}$
12. foreach $\emptyset \subset J \subseteq R' \setminus I$ with $(u, J) \in P$

13. if $l((u, I) + l((u, J) \leftarrow l((u, I \cup J)$
14. $l((u, I \cup J) \leftarrow l((u, I) + l((u, J)$
15. $b(u, I \cup J) \leftarrow \{(u, I), (u, J)\}$
16. if not prune combine$(u, I, J)$ then

17. $Q \leftarrow Q \setminus \{(u, I)\}$
18. $(u, v) \leftarrow \delta(V,E)(u)$

19. if $l((u, I) + \sigma(I, v) \leftarrow l((u, I)$
20. $l((u, I) \leftarrow l((u, I) + \sigma(I, v)$
21. $b(v, I) \leftarrow \{(u, I)\}$
22. if not prune$(u, I)$ then $Q \leftarrow Q \cup \{(v, I)\}$

23. return $(\bigcup_{u \in V} E, r, l(r, R'))$, where

$E' = \text{retraceSMT}(u, r)$

---

**Example 5.** Consider again our instance $I = (N, R)$, where $N = (V, E, \sigma)$ from Example 2, and let $r := c$ be the root node. In the following, we assume a perfect heuristic $h^*$, where $h^*(u, I) = \text{csmt}(N, I \cup \{u\})$ for any vertex $(u, I)$ of $\mathcal{N}(I)$. Then, algorithm DS' requires 10 iterations. The necessary operations can be tracked using the SMT in Figure 1 (right). First, the initial tuples $(a, \{\}), (b, \{\})$ and $(d, \{\})$, all with costs 0, are expanded, cf. Line 10 of Listing 2. Next, the tuples $(b, \{a\})$ of cost 1 and $(I', \{d\})$ of cost 3 are expanded. Then, tuple $(b, \{a\})$ of cost 1 gets expanded. After expanding $(F, \{d\})$ of cost 10 and $(F, \{a, b, d\})$ of cost 22 yielding $\text{csmt}(N, \{a, b, c, d\}) = 22$. This run creates 20 tuples in $Q$. Without the heuristic, i.e., assuming $h^*(u, I) = 0$ for any vertex $(u, I) \in \mathcal{N}(I)$, every tuple $(v, I)$ with costs smaller than 22 has to be expanded. This takes about 55 iterations and 90 entries in Q. Note that this is still considerably less than the absolute worst case of about 160 iterations as in Listing 1 (DW).

### C. Extended Dijkstra-Steiner DS* for admissibility

In this section, we discuss our changes to DS. Since $\mathcal{N}(I)$ is a dynamic graph structure and DS contains adaptions to the $A^*$ algorithm, correctness is not immediately obvious [5]. We therefore also discuss correctness of our new algorithm DS'.

We establish the following definition to lift the existing correctness result of DS for any Steiner heuristic function that provides a cost lower bound.

**Definition 6.** Given an instance $I = (N, R)$ of STP, where $N = (V, E, \sigma)$ and $r \in R$, and a Steiner heuristic function $h^* : V \times 2^R \rightarrow \mathbb{N}$. Then, we say $h^*$ is

- admissible, if $h^*(u, I) \leq l^*(u, I)$ for every $u \in V$ and $\{r\} \subseteq I \subseteq R$; and
- consistent, cf. [5], if $h^*(u, I) \leq h^*(v, I' \cup \{u\}) + l^*(v, (I' \cup \{u\}) \setminus \{r\})$ for every $u, v \in V$ and $\{r\} \subseteq I' \subseteq I \subseteq R$.

Intuitively, with this definition we lift the concept of admissibility of heuristic functions $h$ to Steiner heuristic functions $h^*$ and establish the perspective of admissibility and consistency in the context of Steiner heuristic functions. Similar to heuristic
functions for $A^*$, an admissible Steiner heuristic function $h^*$ does not “over-approximate”, i.e., it provides a lower bound, which is a generalization of the stricter consistency notion [5] that was used before. Note, that a vertex of the Steiner search network intuitively refers to several sub-graphs of potential Steiner trees and not to a simple path as used in $A^*$. As a result, a Steiner heuristic function goes beyond plain heuristic functions on top of the Steiner network. Next, we show that consistency still implies admissibility.

**Proposition 7.** Given an instance $(N, R)$ of STP, where $r \in R$ and a consistent Steiner heuristic function $h^*$. Then, $h^*$ is admissible.

**Proof (Sketch).** We show that $h^*(u, I) \leq l^*(u, I)$ for every $u \in V, \{r\} \subseteq I \subseteq R$, by induction on the cardinality of $I$. The base case of $|I| = |\{r\}| = 1$ is similar to finding the shortest path in $N$ from $u$ to $r$, as used in $A^*$. In particular, in this case consistency and admissibility of Steiner heuristic functions are a special case of consistency and admissibility of heuristic functions, respectively. Therefore, the result follows by the same argument [28], as in the original work for $A^*$. In the induction step we show that $h^*(u, I) \leq l^*(u, I)$ for every $u \in V, \{r\} \subseteq I \subseteq R$, assuming that $h^*(v, J) \leq l^*(v, J)$ for every $v \in V, \{r\} \subseteq J \subseteq I$. Thereby, we use a result [13], which shows that for each vertex $u \in V$, either one of the cases holds:

1. $l^*(u, I) = l^*(u, J) + l^*(u, I \setminus J)$, with $J \subseteq I$: By consistency of $h^*$ and by the induction hypothesis, we have $h^*(u, I) \leq h^*(u, J) + l^*(u, (I \setminus J) \cup \{u\}) \leq l^*(u, J) + l^*(u, I \setminus J) = l^*(u, I)$.
2. $l^*(u, I) = l^*(v, I) + d_N(u, v)$, for $u \neq v$, and for $v$ we have Case 1. We conclude by consistency of $h^*$ and by Case 1: $h^*(u, I) = h^*(v, I) + l^*(v, (I \setminus J) \cup \{u\}) \leq l^*(v, I) + d_N(u, v) = l^*(u, I)$.

However, consistency is indeed strictly stronger, i.e., one can easily construct examples of non-consistent, admissible Steiner heuristic functions.

**Example 8.** Consider network $N$ given in Figure 1 (left). Let $r := c$ and $h^*$ be an admissible Steiner heuristic function. Take $h^*(d, \{c\}) := 16$ and $h^*(F, \{c\}) := 1$. Observe that $h^*$ is admissible, but not consistent, as $h^*(d, \{d\}) = h^*(F, \{c\}) + l^*(F, \{d\})$, i.e., $16 > 1 + 10$.

So far, correctness of DS is known for consistent Steiner heuristic functions [5]. In this case, consistency guarantees optimality of expanded tuples, i.e., for every expanded tuple $(u, I)$ in $P$, cf., Listing 2, we have $l((u, I) = l^*(u, I)$. On the contrary, in case of mere admissibility, which, intuitively, ensures only a cost lower bound, the algorithm might find a lower value for $l((u, I)$ even after expansion of $(u, I)$, i.e. it only holds that $l((u, I) \geq l^*(u, I)$. Recall Listing 2, which indeed presents the DS algorithm. Actually, only Lines 18 and 13 of Listing 2 differ slightly from the original DS algorithm [5]. During expansion, DS explicitly considers already expanded tuples in $P$ since the tentative cost value $l((v, I)$ for an expanded tuple $(v, I) \in P$ is not guaranteed to be optimal.

Although the changes are small, the effect is significant: new Steiner heuristic functions can be used with DS and since the proof of correctness for DS strongly depends on tentative costs of expanded tuples being optimal, we have to use a different approach for showing correctness of $DS^*$. 

**Theorem 9 (main result).** For an instance $(N, R)$ of STP, where $N = (V, E, \sigma)$ and a root $r \in R$, the $DS^*$ Algorithm given in Listing 2, terminates if $h^*$ is admissible. Further, after termination, $l((r, R \setminus \{r\})$ is the cost of an SMT for the instance $(N, R)$, i.e., $l((r, R \setminus \{r\}) = \text{csmt}(N, R)$.

**Proof (Idea).** Since $h^*$ is admissible, we have $h^*(r, R \setminus \{r\}) = 0$. By construction of $DS^*$ and admissibility of $h^*$, every tuple $(u, I)$ such that $l((u, I) + h^*(u, R \setminus I) < l((r, R \setminus \{r\})$ is expanded. As a result, after termination, every tuple $(u, I)$ that “contributes” to $\text{csmt}(N, R)$, is expanded beforehand. •

While consistent Steiner heuristic functions $h^*$, one obtains for DS a similar runtime result as for DW, cf., Proposition 4, this is not guaranteed for admissible functions $h^*$ and $DS^*$. Intuitively, as described above, for non-consistent Steiner heuristic functions $h^*$, paths to arbitrary tuples $(v, I)$ are not necessarily cost optimal.

**Theorem 10.** Given an instance $\mathcal{I} = (N, R)$ of STP, where $N = (V, E, \sigma)$, any root $r \in R$, and an admissible Steiner heuristic function $h^*$. Then, the $DS^*$ algorithm runs in time $O(|R| \cdot (|V| + 2|R| + |V| + |R|)$.

**Proof (Sketch).** Each vertex of the form $(u, I)$ in the Steiner network $\mathcal{N}(\mathcal{I})$ has by construction at most $|V| + 2|R| - |I|$ neighboring tuples in the network. Observe that an SMT of $\mathcal{I}$ consists of at most $|V|$ many vertices and has at most $|R|$ join vertices. Then, the length of a path from any tuple $(u, I)$ to $(r, R \setminus \{r\})$ in $\mathcal{N}$ that is discovered in $DS^*$ is at most $|V| + |R|$, since a longer path would exceed $\text{csmt}(N, R)$, causing algorithm $DS^*$ to not expand the corresponding tuple. We therefore reach the goal $(r, R \setminus \{r\})$ after expanding at most $(|V| + 2|R| - |I|) |V| + |R|$ vertices. In the worst case this is done for each terminal, except the root $r$. Hence, the bound holds. •

In theory, the worst-case runtime may worsen significantly in case of mere admissibility, similar to the situation of exponential blow up for the $A^*$ algorithm [29]. However, in practice the ability to use non-consistent, but admissible heuristics often provides tighter bounds leading to shorter runtimes. The same is true for the $A^*$ algorithm, where at first nobody could imagine merely admissible heuristic functions to outperform consistent ones [29], [30]. For $DS^*$ as well as $A^*$ it turns out that ILP based heuristic functions are both not consistent and often well-suited. In the next section, we discuss the Steiner heuristic functions used in our solver $DS^* Solve$. 

1Proofs of theorems marked with “*” can be found in a self-archived version on arxiv.
IV. IMPLEMENTATION OF DS*: DS* Solve

We created the solver DS* Solve, a fully functional STP solver written in C++. DS* Solve consists of two modules: the solving module, using DS* as presented in Listing 2, and the preprocessing module, trying to reduce the complexity of the instance, before solving is started. In the following, we discuss implementation details regarding both these modules. Within this section, we again assume a given instance \((N, R)\), where \(N = (V, E, \sigma)\) and a root \(r \in V\).

A. Solving Module

DS* is implemented as previously described with the same pruning technique as originally suggested for DS [5]. The main improvement is the choice of Steiner heuristic functions, which is crucial for good results, as our experimental results show. We use two different Steiner heuristic functions and will discuss them subsequently: dual ascent and 1-tree. Given a lower bound \(lb(N, I)\) of \(c_{smt}(N, I)\) for any \(I \subseteq R\), we define the corresponding Steiner heuristic function \(h_t^*(u, I) := lb(N, I \cup \{u\})\) for any \(u \in V, I \subseteq R\).

Dual Ascent [32, Section 3]. This admissible, but not consistent, Steiner heuristic function can only be used with DS* and not DS. The idea is to use a feasible solution for the dual of the ILP formulation as a lower bound. The algorithm starts with edge costs \(\sigma' = \sigma\) and \(|R \setminus \{r\}|\) sub-graphs of \((V, E)\), where each sub-graph \(C_t\) consists of exactly one non-root terminal \(t \in R \setminus \{r\}\). In each iteration one sub-graph \(C_t\) is selected and extended. First, the set \(E_t\) is computed: \(E_t\) consists of all edges \(e \in E\) incident to a node in \(C_t\), but not in \(C_t\). Next, the lowest cost \(e^* = \min_{e \in E_t} \sigma'(e)\) is determined. Then, all edges \(e \in E_t\) with \(\sigma'(e) = e^*\) are added to \(C_t\) and the costs of all edges in \(E_t\) are reduced by \(e^*\) in \(\sigma'\). Eventually, the sub-graphs will form one connected sub-graph \(C_t\), containing all terminals including the root. Therefore, \(C_t\) is a Steiner tree for the instance. The sum of all \(e^*\) when adding an edge is a lower bound for \(c_{smt}(N, R)\). Whenever we select a sub-graph \(C_t\), we choose the sub-graph that has the minimal number of incident edges [8], [33]. The algorithm runs in \(O(|E| \cdot \min\{|V| \cdot |R|, |E|\})\) [34].

1-Tree. Alternatively, DS* Solve uses the 1-Tree lower bound, whose corresponding Steiner heuristic function is consistent. This method is used in the original DS implementation [5].

Proposition 11 (1-tree lower bound). [5, Lemma 8] Given an STP instance \((N, R)\). Let \(r \in R\) be any terminal and \(R^r = R \setminus \{r\}\). Further, let \(c\) be the cost of any minimum spanning tree of \(D_N(R^r)\). Then,
\[
\frac{1}{2}(c + \min_{u, v \in R^r, u \neq v} \sigma(|R^r| = 1) d_N(r, u) + d_N(r, v))
\]
is a lower bound of any SMT of \((N, R)\).

The distance network can be constructed in time \(O(|V| \log(|V|) + |E|)\) and the minimum spanning tree for this network in time \(O(|R|^2)\). The lower bound can then be obtained in time \(O(|R|)\) [5].

DS* Solve only uses one of the corresponding Steiner tree heuristic functions per instance. Oftentimes, dual ascent computes tighter bounds and DS* therefore requires fewer iterations to succeed. Unfortunately, the runtime for dual ascent also increases faster with increasing graph size than the runtime of 1-tree. Hence, we use dual ascent for graphs of up to 10,000 edges and use the 1-tree heuristic function for larger graphs.

The root terminal is not chosen arbitrarily. DS* Solve tries different terminals as the root with dual ascent. The terminal giving the highest lower bound is then used as the root. This strategy delivered superior results, compared to selecting the last terminal as used originally [5].

B. Preprocessing Module

Preprocessing for STP identifies and removes vertices and edges that are not required for an SMT. These methods usually run in time \(O(|V| \log(|V|))\) and can quickly reduce the complexity of an instance. Many instances would be too hard for the solving module alone, without preprocessing.

We use the standard methods described in literature [2], [35]. Preprocessing is applied until no more reductions are possible. The order of preprocessing operations is chosen in a way to maximize reuse of calculated information, e.g. distances in the network. One noteworthy preprocessing method is based on the previously discussed dual ascent lower bound.

Dual ascent calculates a lower bound for \(c_{smt}(N, R)\) and can be extended to provide a lower bound on the costs of any SMT containing a specific vertex or edge. Given an upper bound for \(c_{smt}(N, R)\) we can remove any vertices and edges, where this lower bound exceeds the upper bound. We use different methods to obtain good upper bounds and thereby maximize the number of removals.

Upper bounds are obtained using the repeated shortest path heuristic (RSPH). RSPH is a well-established STP heuristic [36], [37], that can compute Steiner trees in time \(O(|R| \cdot |V|^2)\). We can often find a tighter upper bound by running RSPH using only one sub-graph that contains all terminals. In addition to using the full instance, we run RSPH also using the following sub-graphs:

1) The sub-graph \(C_t\) after a dual ascent run [33].
2) A preprocessed instance after guessing an upper bound [33].
3) A sub-graph obtained from combining several Steiner trees [33], [38].

Given a Steiner tree, we can also find a Steiner tree of lower costs by applying local search. This is done by systematically replacing sub-graphs of the tree in an effort to find a tree of smaller size. These sub-graphs may be single vertices or whole paths [39].

V. EXPERIMENTAL WORK

We conducted a series of experiments using standard benchmark instances for STP. Instances and results\(^3\), including raw data, are publicly available. Our experimental work aims for

\(^3\)We refer the following online resource [40].
a comparison between different heuristics in order to understand whether our proposed extensions are valuable in practice. Further, we compare the effectiveness of our prototypical solver with other established implementations.

**Benchmark Instances.** We considered a selection of overall 1,623 instances, which originate from the 11th DIMACS Challenge [23] and PACE 2018 Challenge [24]. We group them as done in the literature [6].

1. SteinLib, which contain generated graphs with random costs (random), artificial instances (artificial), instances with euclidean weights (euclidean), cross-grid graphs (crossgrid), grid graphs with holes (vlsi), randomly generated rectilinear instances (rectilinear), wire routing problem based instances (group);
2. Cph14 (simplified obstacle-avoiding rectilinear instances);
3. Vienna (real world instances from telecommunication networks);
4. PACE2018 (PACE 2018 challenge instances).

**Measure, Setup, and Resource Enforcement.** Our results were gathered on a cluster running Ubuntu 18.04.3 LTS (kernel 4.15.0-101-generic) and GCC 7.5.0. Each node is equipped with two Intel Xeon E5-2640v4 CPUs and 160GB RAM. We limited the solvers to 1800 seconds wall clock time and 8GB of RAM per instance. We used reprobench\(^6\) to setup the benchmarks and to enforce the resource limits.

**Benchmarked Solvers.** We tested three configurations of our solver: DS\(^\ast\), which implements our enhanced algorithm without preprocessing, DS\(^\ast\) Solve, which in addition includes preprocessing, and -da, which on uses the 1-tree heuristic function. We test the solver HSV\(^\ast\), which is known as a successful implementation of DS algorithm [5]. We include the best PACE 2018 solvers Pruned\(^7\) (2700bdc/Rust1.36.0) [6] and SCIP-Jack\(^8\) (6.0.2/SoPlex4.0.2).

**Experiment 1.** In order to benchmark the effectiveness of our enhanced algorithm and a comparison to plain DS, we take DS\(^\ast\) and the solver HSV\(^\ast\) into account. Since one might argue that also implementation specific tricks and algorithm engineering might have a strong influence on the number of solved instances, we also tested our solver in the configuration -da, which disables the dual ascent heuristic and assembles a solver that is close to the underlying algorithm of HSV. Since HSV can handle only instances of less than 64 terminals, we restrict the instances accordingly.

**Result:** Table I the results on the number of solved instances. DS\(^\ast\) without dual ascent performs worse than HSV. DS\(^\ast\) performs better on all sets. If we subgroup SteinerLib, one can observe that it particularly helps to solve randomly generated graphs (random) and wire routing problem based instances (group).

**Discussion:** Besides implementation details, DS\(^\ast\)-da performs worse than HSV. We suspect that the main reason is the use of the Steiner heuristic functions in HSV, which is slightly more sophisticated than the 1-tree heuristic function [5]. The results show that using merely admissible Steiner heuristic functions can significantly improve the number of solved instances.

**Experiment 2.** In order to contrast our algorithm and implementation with approaches and their implementations, we take various state-of-the-art solvers in the field into account.

**Results:** In Table II, we report the number of solved instances for the considered solvers. N indicates the overall number of instances.

![Table I: Number of solved instances with less than 64 terminals for DS-based solvers. N\(^\ddagger\) lists the results for DS\(^\ast\) without dual ascent heuristic. N indicates the total number of instances.](image-url)

| Set        | N   | DS\(^\ast\) | DS\(^\ast\)-da | HSV\(^\ast\) |
|------------|-----|-------------|----------------|-------------|
| SteinLib   | 820 | 691         | 522            | 557         |
| Cph14      | 10  | 10          | 10             | 10          |
| Vienna     | 6   | 4           | 3              | 3           |
| PACE2018   | 195 | 185         | 157            | 157         |
| \(\Sigma\) | 1031| 890         | 692            | 727         |

**Table II: Number of solved instances for the considered solvers. N\(^\ddagger\) indicates the overall number of instances.**

| Set        | N\(^\ddagger\) | DS\(^\ast\) Solve | DS\(^\ast\)-Solve-da | Pruned | SCIP |
|------------|----------------|-------------------|----------------------|--------|------|
| SteinLib   | 1186           | 928               | 796                  | 874    | 1055 |
| Cph14      | 21             | 15                | 12                   | 16     | 17   |
| Vienna     | 216            | 11                | 5                    | 26     | 120  |
| PACE2018   | 200            | 189               | 178                  | 189    | 178  |
| \(\Sigma\) | 1623           | 1143              | 991                  | 1105   | 1370 |

\(^6\)https://github.com/rkkautsar/reprobench

\(^7\)https://github.com/wata-orz/steiner_tree

\(^8\)https://scip.zib.de/
Figure 2: Runtime of DS*: Solve vs SCIP. The x-axis labels the number of terminals of the instance and y-axis captures the runtime in seconds.

VI. CONCLUSION AND FUTURE WORK

In this paper, we established the concept of admissible Steiner heuristic functions for the Steiner tree problem. We focused on instances with few terminals, and lifted the so-called Dijkstra-Steiner (DS) algorithm from consistent heuristic functions to admissibility, resulting in the DS* algorithm. Intuitively, admissibility of heuristic functions only requires a weak condition for lower bounds. More precisely, an admissible heuristic function is guaranteed to never over-approximate the actual costs, and is indeed strictly weaker than consistency. Admissible heuristic functions enable lower bound computation based on LP techniques, as for example the dual ascent method. Our solver DS* Solve combines the usage of admissible heuristic functions during solving, efficient preprocessing techniques, and methods for obtaining strong upper bounds.

An interesting question for future work is the integration of the dual ascent technique into other approaches and solvers.

REFERENCES

[1] M. Hauptmann and M. Karpinski, “A compendium on Steiner tree problems,” http://theory.cs.uni-bonn.de/info5/steinerkompendium/, 2015.
[2] F. K. Hwang, D. S. Richards, and P. Winter, The Steiner Tree Problem, ser. Annals of Discrete Mathematics. Elsevier, 1992, vol. 53.
[3] R. M. Karp, “Reducibility among combinatorial problems,” in Complexity of computer computations. Plenum, 1972, pp. 85–103.
[4] M. Fischetti, M. Leitner, I. Ljubić, M. Luipersbeck, M. Monaci, M. Resch, D. Salvagnin, and M. Sinnl, “Thinning out Steiner trees: A node-based model for uniform edge costs,” Mathematical Programming Computation, vol. 9, no. 2, pp. 203–229, 2017.
[5] S. Hougardy, J. Silvanus, and J. Vygen, “Dijkstra meets Steiner: A fast exact goal-oriented Steiner tree algorithm,” Mathematical Programming Computation, vol. 9, no. 2, pp. 135–202, 2016.
[6] Y. Iwata and T. Shigemura, “wata* Sigma: a Steiner tree solver,” http://dimacs11.zib.de/wata_orl/steiner_tree_solver/.
[7] T. van der Zanden, “SteinerTreeTW,” https://github.com/TomvdZanden/SteinerTreeTW, 2018.
[8] T. Pajor, E. Uchoa, and R. F. Werneck, “A robust and scalable Steiner tree solver for pace 2018,” https://github.com/PMitura/pace2018, 2018.
[9] G. Gamrath, T. Koch, J. Maher, D. Rehfeldt, and Y. Shinano, “SCIP-Jack—a solver for STP and variants with parallelization extensions,” Mathematical Programming Computation, vol. 9, no. 2, pp. 231–296, 2017.
[10] D. S. Johnson, T. Koch, R. F. Werneck, and M. Zachariasen, “11th dimacs implementation challenge in collaboration with icerm: Steiner tree problems,” http://dimacs11.zib.de/, 2014.
[11] É. Bonnet and F. Sikora, “The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration,” in IPEC’18, LIPIcs, vol. 115. Dagstuhl Publishing, 2019, pp. 26:1–26:15.
[12] C. H. Papadimitriou, Computational Complexity. Addison-Wesley, 1994.
[13] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality of a∗,” J. ACM, vol. 32, no. 3, pp. 505–S, Jul. 1985.
[14] Z. Zhang, N. R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner, “A* Search with Inconsistent Heuristics,” in IJCAI 2009. 2009, pp. 634–639.
[15] A. Martelli, “On the Complexity of Admissible Search Algorithms,” Artificial Intelligence, vol. 8, pp. 1–13, 1977.
[16] A. Schidler, M. Hecher, and J. K. Fichte, “DS Star – a Steiner tree solver,” Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.1209429
[17] R. T. Wong, “A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph,” Mathematical Programming, vol. 28, no. 3, pp. 271–287, 1984.
[18] T. Polzin, “Algorithms for the Steiner Tree Problem in Networks,” PhD Thesis, Universität des Saarlandes, 2003.
[19] C. Wu, “Steiner’s Problem in Graphs,” PhD Thesis, University of Amsterdam, 1993.
[20] C. W. Duin and A. Volgenant, “Reduction Tests for the Steiner Problem in Graphs,” Networks, vol. 19, no. 5, pp. 549–567, 1989.
[21] H. Takahashi and A. Matsuyama, “An approximate solution for the steiner problem in graphs,” Mathematica Japonica, vol. 24, no. 6, pp. 573-577, 1980.
[22] M. P. de Aragão and R. F. Werneck, “On the Implementation of MST-Based Heuristics for the Steiner Problem in Graphs,” in Algorithm Engineering and Experiments. Springer, 2002, pp. 1–15.
[23] C. Ribeiro, E. Uchoa, and R. F. Werneck, “A Hybrid Grasp with Perturbations and Adaptive Path-Relinking for the Steiner Problem in Graphs,” 2000.
[24] E. Uchoa and R. F. Werneck, “Fast Local Search for the Steiner Problem in Graphs,” ACM Journal of Experimental Algorithms, vol. 17, no. 2, pp. 2.2.1–2.2.22, 2012.
[25] J. K. Fichte, M. Hecher, and A. Schidler, “Raw data on experiments for DS Star,” Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.1209429