The Influence of Preprocessing on Steiner Tree Approximations[⋆]

Stephan Beyer and Markus Chimani

Institute of Computer Science, University of Osnabrück, Germany,
{stephan.beyer,markus.chimani}@uni-osnabrueck.de

Abstract Given an edge-weighted graph G and a node subset R, the Steiner tree problem asks for an R-spanning tree of minimum weight. There are several strong approximation algorithms for this NP-hard problem, but research on their practicality is still in its early stages.
In this study, we investigate how the behavior of approximation algorithms changes when applying preprocessing routines first. In particular, the shrunken instances allow us to consider algorithm parameterizations that have been impractical before, shedding new light on the algorithms' respective drawbacks and benefits.

1 Introduction

Given a connected graph $G = (V, E)$ with edge costs $c\colon E \to \mathbb{R}_{\geq 0}$ and a subset $R \subseteq V$ of required nodes called *terminals*, the *Minimum Steiner Tree Problem in Graphs (STP)* asks for a minimum-cost terminal-spanning subtree $T = (V_T, E_T)$ in G, that is, a tree with $R \subseteq V_T \subseteq V$ and minimum $c(T) := c(E_T) := \sum_{e \in E_T} c(e)$. The STP is long known to be NP-hard and even APX-hard [15].

While 2-approximations can be found easily, breaking this barrier has been a theoretically challenging but fruitful field, resulting in a series of ever-decreasing approximation ratios (see Section 2). The currently strongest known algorithms guarantee a ratio of $1.39+\varepsilon$ for general instances. All these below-2 approximation algorithms share a common trait that makes them usually worrisome in practice: ε depends on a parameter k, assumed to be constant. The algorithm's running time, however, is exponentially dependent on this k, as we have to consider all trees spanning any subset of up to k terminals (see below for details). Hence, to achieve low ε, we have to live with increasingly high running times.

Research on the practicality of below-2 Steiner tree approximations started only very recently. In a series of papers [1,2,4][1], it was shown that the algorithms' dependency on k seems to be an insurmountable stumbling block in practice. Typically, only values of $k = 3$ and probably, for small graphs, $k = 4$ seemed worthwhile. These findings were validated in [5] in the context of a further approximation algorithm not considered in [2].[2]

[⋆] Funded by project CH 897/1-1 of the German Research Foundation (DFG).

[1] Article [2] is an extended version of both [1,4], with several implementation improvements; when referring to these studies in the following, we will only cite [2].

It is certainly beneficial to the field that the STP has garnered enough interest (in particular also from the communities of exact and (meta)heuristic approaches) that SteinLib [11], a wide collection of generated and real-world instances, has been established. We use this well-established testbed as the basis of our study. Some SteinLib instances have special properties (like being Euclidean, planar, or quasi-bipartite), and various special-purpose algorithms have been developed for such settings. We, however, refrain from explicitly exploiting such properties since we are interested in the approximation algorithms for general graphs.

Contribution. Preprocessing means to shrink a given input by simplifying parts that are 'easy' to solve, favorably in theoretically and practically small time. Our considered research questions can be summarized as follows:

- *Interplay between preprocessing and approximations.* Neither of the above mentioned studies on approximation algorithms consider preprocessing. This was in order to concentrate on the algorithmic behavior of the approximations and to rule out any shadowing influences arising from preprocessed data. In this study, we want to analyze the influence of preprocessing on the approximations. Clearly, the approximations become faster when run only on reduced instances. But it is already unclear if preprocessed instances help the algorithms to find significantly better solutions.
- *Approximation behavior for larger k.* As noted above, the approximation algorithms are typically restricted to $k = 3$ when run on the original SteinLib instances. As we will see, the preprocessing routines are strong enough to obtain graphs so small that larger values of k become feasible. So now, for the first time, we can investigate the approximation algorithms' behavior for such values. This is in particular interesting due to a central finding in [2]: There, the oldest below-2 approximations (AC3 and RC3 in the description below) typically outperform the newer, formally stronger approximations—not only w.r.t. running time but also often w.r.t. solution quality. This can be attributed to the fact that, for too small values of k, the actual approximation guarantees of the latter are still worse than AC3's 11/6. Only for larger k, their ratios become (at least formally) strictly better. The question is whether these theoretical results carry over to practical benefits.

We are explicitly *not* arguing that our considered algorithms would be particularly good in practice. In fact, we will see in the last section that state-of-the-art heuristics and exact approaches still outperform all theoretically strong approximations. We are interested in learning more about practical stumbling blocks and possible achievements for the class of strong approximation algorithms, in the hope that such a deeper understanding will at some point help to find practically relevant strong approximation algorithms, at least for certain circumstances.

[2] In [5], an implementation of [3] is considered; in [2], the improved variant [9] is investigated instead, as it gives the same guarantee with a smaller runtime complexity.

Outline. The following two sections summarize the approximation algorithms and preprocessing routines we use for our study, respectively. Section 4 contains the actual experimental evaluation.

Preliminaries. For any graph H, we denote its nodes and edges by V_H and E_H, respectively. When referring to the input graph G, we omit the subscript. Considering edge costs, we use the terms *cost* (expensive, cheap) and *length* (long, short) interchangeably. Let $d(u, v)$ be the *distance* between nodes $u, v \in V$, i. e., the cost of the cheapest path between u, v. Let the *neighborhood* $\mathcal{N}(v)$ of $v \in V$ be the set of nodes adjacent to v, and the *Voronoi region* $\mathcal{V}(r) = \{r\} \cup \{v \in V \setminus R \mid d(r, v) \leq d(s, v)\ \forall s \in R\}$ of a terminal $r \in R$ be the set of nodes that is nearer to r than to any other terminal. If $d(r, v) = d(s, v)$ for $s \neq r$, the node v is arbitrarily assigned to either $\mathcal{V}(r)$ or $\mathcal{V}(s)$. Hence the set of Voronoi regions of each terminal is a partition of V. We define $\mathrm{base}(v) := r$ for all $v \in \mathcal{V}(r)$ and $r \in R$. By $\mathrm{MST}(H)$ we denote a minimum spanning tree in H.

2 Approximation Algorithms and their Engineering

We briefly summarize the algorithms considered for our experimental evaluation. They are identical to the ones considered in [2]. This allows direct comparisons w.r.t. the additional influence of the preprocessing routines. The only additional algorithmic modification for the approximations is the addition of a Dreyfus-Wagner type generation of full components (see below).

Algorithm TM. One of the simplest and most efficient approximation algorithms is the 2-approximation algorithm by Takahashi and Matsuyama [21]. We start with a single terminal node as T_0. In the i-th iteration, we construct T_i from T_{i-1} by adding the shortest path between T_{i-1} and t, where $t \in R \setminus V_{T_{i-1}}$ is the nearest terminal to T_{i-1}. The output of the algorithm is $T_{|R|-1}$. Well-implemented, it has proven to be the most efficient 2-approximation (among the algorithms by Kou et al. [12] and Mehlhorn [14]) in terms of time and solution quality [2, 16].

Full Components. Given a Steiner tree T, we can assume that all its leaves are terminals. We can split all its non-leaf terminals to obtain connected components whose leaves are exactly their terminals. Such components are called *full components* of T. They are *k-restricted* if they contain at most k terminals.

Let $k \geq 3$ be constant and $\mathcal{C}_k$ be the set of all possible k-restricted full components. All below-2 approximation algorithms evaluate the elements of $\mathcal{C}_k$ to find a good subset $\mathcal{S} \subseteq \mathcal{C}_k$, such that the components in $\mathcal{S}$ together yield a feasible Steiner tree. The evaluation method differs per approximation algorithm.

In [2], several methods to construct $\mathcal{C}_k$ are evaluated. All methods are based on shortest path algorithms, and the first beneficial idea is to use a variant of shortest path algorithms that, in case of a tie, prefers a path over terminals. Such paths can be discarded for construction of full components. For the special case of $k = 3$, one can also choose between multiple calls to a single-source shortest path algorithm (SSSP) in overall $\mathcal{O}\big(|R| \cdot |V|^2\big)$ time and a single call to an all-pair shortest path algorithm (APSP) in time $\mathcal{O}\big(|V|^3\big)$. When applied to SteinLib, SSSP should be chosen if and only if the graph's *density* $|E|/\binom{|V|}{2}$ is at most 0.25.

For $k \geq 4$, we have to use a shortest-path matrix between all nodes, so APSP is the only viable choice. However, the success rates of computing $\mathcal{C}_k$ using the enumeration scheme given in [2] are rather bad. In [5], it is proposed to compute $\mathcal{C}_k$ essentially by running the first k iterations of the Dreyfus-Wagner algorithm [7]; we call this method DW. It is based on the simple observation that in order to compute a minimum-cost tree spanning k terminals, the Dreyfus-Wagner algorithm also computes all minimum-cost trees spanning at most k terminals. Hence one call of DW yields all k-restricted full components.

Algorithms ACk *and* RCk. Zelikovsky [22] was the first one to exploit the idea of k-restricted full components for a below-2 approximation algorithm. He summarized the ideas as the *greedy contraction framework* [25]: we first compute a 2-approximation T_0. Given T_{i-1}, we seek the full component $C \in \mathcal{C}_k$ that promises the maximum improvement over T_{i-1} according to some given *win function.* T_i is constructed by contracting C in T_{i-1} and removing the most expensive edge in each arising cycle, that is, $T_i = \mathrm{MST}(T_{i-1}/C)$. This is repeated until no promising component is found. The contracted components and the remaining edges in the last constructed tree together form the approximative solution to the original Steiner tree problem.

In this setting, ACk is the algorithm that uses an *absolute* win function that describes the actual cost reduction of T_{i-1} when including a component C. It provides an approximation ratio of $11/6 \approx 1.83$ [22–24] for $k \geq 3$. RCk uses a *relative* win function [25] that relates the saved cost by contracting C in T_{i-1} to the cost of C. For $k = 3$, it only achieves ratio 1.97; however, the ratio decreases for higher k, down to 1.69 for $k \to \infty$. We say the approximation ratio is $1.69 + \varepsilon$.

Generating $\mathcal{C}_3$ can be accelerated by using an on-demand generation [23] for AC3, or a Voronoi-based approach [24] for RC3. The latter is also valid for $k \geq 4$, but the general enumeration approach (even without DW) is favorable [2].

We also use the beneficial implementation details mentioned in [2]: a strategy that allows to discard non-promising components early (and hence reduces the number of full components) and a data structure for lowest common ancestor queries on static auxiliary trees to efficiently obtain values for the win functions.

Algorithms LCk. Robins and Zelikovsky [20] achieved a ratio $1.55 + \varepsilon$ by using *loss-contractions* (instead of simple contractions) of C in the greedy contraction framework. The *loss* of C is a sub-forest in C such that each nonterminal is connected to exactly one terminal. A loss-contraction performs a contraction of the loss edges only (instead of all edges). The used win function represents the concept of a relative win function transfered to loss-contractions.

For the generation of full components, we have to use the general enumeration approach for every $k \geq 3$. It is beneficial to compute the loss on components containing edges representing shortest paths instead of the original edges [2].

Algorithms LPk. The algorithm by Goemans et al. [9] is the most recent Steiner tree approximation and achieves a ratio of $1.39 + \varepsilon$, similar to the algorithm by Byrka et al. [3]. Both algorithms are based on linear programming techniques. The corresponding LP formulations have exponentially many constraints but can be solved in polynomial time using a separation oracle that involves a linear number of maximum flow computations. However, the approach by Goemans et al. is favorable since the resulting LP formulation has less variables (by a factor proportional to k) and the LP relaxation is solved only once.

Each variable in the formulation represents a full component in $\mathcal{C}_k$. The algorithm first solves the LP and then applies a sophisticated randomized rounding scheme to obtain a provably good integral solution. There, the idea is to iteratively contract a (fractionally chosen) component, and make the solution feasible for the contracted problem again (which involves removal and splitting of other components). In order to minimize the cost of the resulting feasible solution, we estimate $w(C)$, the maximum cost of the edges that are to be removed in order to re-establish feasibility after contracting C. Any component C with $c(C) \leq w(C)$ can be chosen. (It is guaranteed that there always exists such a component.) We note that finding such a maximum-cost set of edges to be removed reduces to a minimum-cost flow computation in an auxiliary network.

For the generation of $\mathcal{C}_k$, the general enumeration algorithm has to be used. It is beneficial to augment the maximum-flow-based separation oracle with further preconditional tests; to choose 'leaf components' before actually invoking the approximation algorithm on the fractional solution; and to use edges representing shortest paths (instead of original edges) in the auxiliary network [2].

Exact Algorithms BC *and* BCS. We compare the approximations to two exact algorithms. Algorithm BC is a branch-and-cut approach based on the bidirected cut LP formulation [13]; it is our own straight-forward, simple, and short implementation using a standard branch-and-cut framework (ABACUS [10]), and therefore arguably easier to implement than, e.g., the sophisticated strong approximation algorithms. Furthermore, we consider the highly tuned, more sophisticated version of BC that has been presented in [8]. It has been one of the winners of the *11th DIMACS Implementation Challenge* on Steiner tree problems [6], and we denote it by BCS.

3 Preprocessing Techniques

We use reduction tests to preprocess STP instances. These tests check for conditions that imply the inclusion or exclusion of nodes and edges in a Steiner minimum tree (SMT)[3]. For our experimental evaluation, we only consider efficient tests, i. e., with running time bounded by $\mathcal{O}(|V|^2)$. Hence, when applied iteratively, the total running time of the preprocessing process is $\mathcal{O}(|E| \cdot |V|^2)$.

[3] This word order and abbreviation is historically common, to avoid conflicts with the established abbreviation 'MST' for minimum *spanning* tree.

We give a brief overview over the used reduction tests. A more detailed decription (including proofs) is given, e.g., by Polzin and Vahdati-Daneshmand [18].

Trivial reductions. Test D_1 deals with nodes of degree 1. Let $v \in V$ with $\deg(v) = 1$ and $e = \{v, w\}$. If $v \in R$ and $|R| \geq 2$, edge e must be in every SMT. We can hence contract e and let the resulting node be a terminal. If $v \in V \setminus R$, remove v (and e) since it is not part of any SMT. Test NTD_2 considers nonterminals of degree 2 and replaces its two incident edges by a single edge. Test S removes self-loops and parallel edges (keeping the edge of minimum cost). Another trivial test is to remove connected components not containing any terminals (NTC). We summarize the former tests as trivial test set T.

Consider the *distance network* on the terminals R, i. e., the complete graph on R with edge costs resembling the length of the shortest paths in G between the respective terminals. Here and in the remainder of this paper, let M denote an MST in this distance network. It can be computed in time $\mathcal{O}(|E| + |V| \log |V|)$ using Voronoi regions [14]. The simple test CTD (cost vs. terminal distance) is to remove every edge e with cost larger than the shortest path between two terminals, that is, if $c(e) > \max_{f \in E_M} \{c(f)\}$.

Classical inclusion tests using Voronoi regions. Let $s \in R$. Test NV (nearest vertex) is as follows: Let $\{s, u\}$ and $\{s, w\}$ be the shortest and second shortest edge incident to s, respectively. The edge $\{s, u\}$ belongs to at least one SMT if

$$c(\{s, w\}) \geq \begin{cases} \mathrm{rdist}_u(s) & \text{if } u \in \mathcal{V}(s), \\ c(\{s, u\}) + d(u, \mathrm{base}(u)) & \text{otherwise,} \end{cases}$$

where $\mathrm{rdist}_u(s)$ is the shortest distance between s and any $t \in R$, $t \neq s$, over $\{s, u\}$.

Test SL (short links) is as follows: Let $e_1 = \{v_1, w_1\}$ and $e_2 = \{v_2, w_2\}$ with $v_i \in \mathcal{V}(s), w_i \notin \mathcal{V}(s), i = 1, 2$, be the shortest and second shortest edge, respectively, that leaves $\mathcal{V}(s)$. Then e_1 belongs to at least one SMT if $c(e_2) \geq d(s, v_1) + c(e_1) + d(w_1, \mathrm{base}(w_1))$.

Tests based on the Steiner bottleneck distance. A path P in G can be decomposed into a sequence of *full paths*, i. e., full components with exactly two terminals each. Let $L(P)$ be the longest full path of P. The *bottleneck Steiner distance* $b(u, v)$ is the minimum $c(L(P))$ over all paths P between u and v. We use an upper bound on $b(u, v)$ as described in [18]: if u and v are terminals, we query the largest cost on the unique u-v-path in M (which is a lowest common ancestor query in a weight-tree); if u and/or v is not a terminal, we restrict ourselves to paths over their respective κ nearest terminals, for some constant κ.

Test PT (paths with many terminals) removes every edge $\{u, v\}$ with $c(\{u, v\}) > b(u, v)$. Here, PT should be followed by NTC to remove nonterminal components.

Let $d \geq 3$ be constant. Test NTD_d (nonterminals of degree d) checks for every $v \in V \setminus R$ with $\deg(v) \leq d$ whether it satisfies

$$\sum_{w \in N} c(\{v, w\}) \geq c(\mathrm{MST}((N, N \times N, b))) \qquad \forall N \subseteq \mathcal{N}(v), |N| \geq 3.$$

Here $(N, N \times N, b)$ is the complete graph over N where the costs are given by (upper bounds on) the bottleneck Steiner distances. If the above universally quantified property holds, v has $\deg_T(v) \leq 2$ in at least one SMT T. Consequently, v can be removed, and edges $\{u, w\}$ with cost $c(\{u, v\}) + c(\{v, w\})$ are introduced for (all) $u, w \in \mathcal{N}(v)$.

After a pilot study, we chose $\kappa = 5$ and $d = 10$ as the best practical values.

Bound-based tests using Voronoi regions. Bound-based tests compute a lower bound for the SMT under the assumption that a nonterminal or an edge is included in the SMT. If that lower bound exceeds an upper bound U, the assumption must be false and the nonterminal or edge can be removed.

We can use Voronoi regions to compute lower bounds. Let $\text{exit}(s), s \in R$, be the shortest distance to any $v \in V \setminus \mathcal{V}(s)$. Let X be the sum of all these values except the largest two, and let $d_i(v)$ be the distance to the i-th nearest terminal of $v \in V$. Test *LBE* (lower bound on edges) removes $e = \{u, v\}$ if $c(e) + d_1(u) + d_1(v) + X > U$. Let $G' := (R, E', c')$ with

$$E' := \{\{s,t\} \mid \{u,v\} \in E, u \in \mathcal{V}(s), v \in \mathcal{V}(t)\} \text{ and}$$
$$c'(\{s,t\}) := \min\{c(e) + \min\{d_1(u), d_1(v)\} \mid e = \{u,v\} \in E, u \in \mathcal{V}(s), v \in \mathcal{V}(t)\}.$$

Test *LBN* (lower bound on nodes) removes $v \in V \setminus R$ if $d_1(v) + d_2(v) + \min\{X, c'(\text{MST}(G')) - \max_{e \in E'} c'(e)\} > U$.

4 Experimental Evaluation

In the following experimental evaluation, we use an Intel Xeon E5-2430 v2, 2.50 GHz running Debian 8. The binaries are compiled in 64bit with g++ 4.9.0 and `-O3` optimization flag. All algorithms are implemented as part of the free C++ Open Graph Drawing Framework (OGDF), the used LP solver is CPLEX 12.6. We apply our algorithms on all 1200 connected instances from SteinLib.

As in [2, 8], we say that an algorithm *fails* for a specific instance if it exceeds one hour of computation time or needs more than 16 GB of memory; otherwise it *succeeds*. The *success rate* is the percentage of instances that succeed.

By χ we denote the ratio of edges remaining after the preprocessing.

We evaluate the solution quality of a solved instance by computing a *gap* as $\frac{c(T)}{c(T^*)} - 1$, the relative discrepancy between the cost of the found tree T and the cost of the optimal Steiner tree T^*, usually given in thousandths (‰). When no optimal solution values are known, we use the currently best known upper bounds from the 11th DIMACS Challenge [6].

Preliminaries. Before we go into the depth of our main research questions, we first have two discuss two relevant preliminary issues.

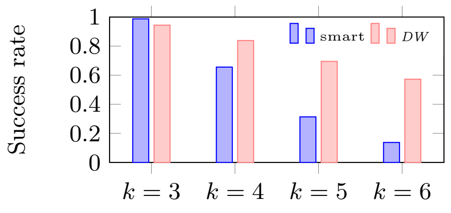


Figure 1. Success rates of component enumeration methods.

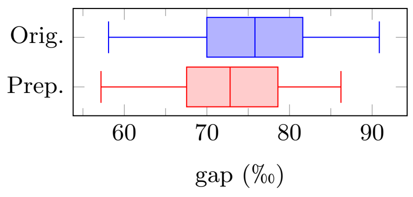


Figure 2. Average variation in solution gaps due to different shufflings.

Component generation via DW. Fig. 1 illustrates the success rates of different component enumeration methods: the 'smart' one from [2], and DW. The results are clear and consistent with theory: While the former is better for $k = 3$, the latter outperforms it for $k \geq 4$. Hence, for $k \geq 4$ we will from now on use DW. For $k = 3$, we can still use the the best applicable approaches [2]: the direct approach for AC3, the Voronoi-based approach for RC3, the 'smart' approach for LC3 and LP3.

Preprocessing Sequence. The described reduction tests can be applied in various permutations and repetition schemes. The application of one reduction may help or hinder another reduction to be applicable. Since the used preprocessing is always faster than the subsequent algorithms, we focus on the reduction quality.

Using only *T* already gives an average $\chi = 90.6\,\%$. We considered all $7! = 5040$ possible permutations of the preprocessing steps that start with *T*. For each sequence, we iterate until no further reduction is achieved. The sequence ⟨*T*, *CTD*, *LBE*, *PT*, *NTD*, *SL*, *LBN*, *NV*⟩—using *T* in between whenever necessary—gives the best average χ of 75.49 %. However, all sequences resulted in comparable reduction ratios and running times; the worst permutation gives average $\chi = 75.57\,\%$. The percentage of instances solved purely via preprocessing is 5.25 %, and 9.75 % of the instances are not susceptible to any of our considered reductions.

The average runtime of our preprocessing is negligible for the subsequent strong approximation algorithms (0.28 seconds on average). The only notable outliers are `alue7080` taking 15 seconds and `es10000fst01` taking 35 seconds.

Effect of Preprocessing on Solution Quality. In this subsection we temporarily disregard instances that are not susceptible to preprocessing at all. Our first attempt to compare the solution quality with and without preprocessing was surprisingly hazy. While the gaps slightly decreased *on average*, the situation was muddled on the instance level: for many instances the gap indeed shrunk, but it also increased for many. The reason seems to be primarily that all algorithms have to rely on some tie-breaking when choosing between multiple equally long shortest paths, multiple components with the same win-value, etc. This tie-breaking is implicitly done based on the internal order of the nodes and edges in the graph data structure. To investigate the interplay between these orders

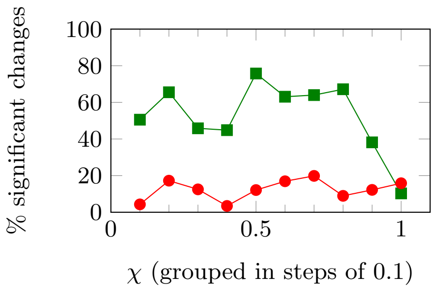


Figure 3. Preprocessing: percentage of solution quality changes that are significant. Green squares are improvements, red circles are degradations.

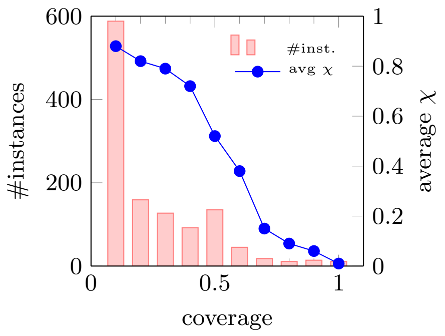


Figure 4. Average χ (ratio of edges remaining after preproc.) over all graphs.

and the preprocessing, we *shuffle* all instances 50 times, i. e., we have 50 versions of the *same* instances differing only in internal storage order, and report on TM without and with preprocessing. Our results for the other algorithms, albeit only on a random sample of the instances, are analogous.

Probably despite one's first intuition, for any given instance, the so-generated 50 solutions with (or without) preprocessing do *not* resemble a normal distribution; in fact they are very far away from doing so (see Appendix A for illustrative examples). The central limit theorem does not apply since choices of edges (and hence solution costs) are not independent. In other words, we cannot apply the standard (parametric) significance tests in our statistical evaluation.

Fig. 2 gives a comparison of the average distributions for the gaps obtained by our original and preprocessed instances. We observe that the median gap improves, and the whole range as well as the quartile ranges shrink and move to the left. This indicates a favorable (though statistically insignificant) effect of preprocessing on the gaps in general. Interestingly, for VLSI and Euclidean instances we even see a slight increase of the gap, see Appendix B.

Our first hypothesis is that the (arbitrary) order of the elements in the graph data structure has a larger influence on the solution quality than the preprocessing. Consider some instance. We are interested in the probability p that its solution will improve by preprocessing. If $p = 0.5$, any improvements are only due to chance based on different shufflings; if $p = 1$, preprocessing will always improve the solution independent of the shuffling. We compute these probabilities based on our 50 shufflings, and obtain an average of only $p = 0.58$ over all 1083 instances, i.e., we accept the hypothesis.

Our second hypothesis is that, despite this overshadowing effect, preprocessing does significantly improve (or at least does not significantly decrease) solutions. Due to the lack of a normal distribution, we resort to the Wilcoxon-Mann-Whitney test, and choose a significance level of 5 % (cf. Fig. 3). We observe a significant

Table 1. Results of different algorithms. Per algorithm, we provide the success rates in percent and the average times in *sec* among all succeeded instances, as well as the percentage of optimally solved instances among all instances. PUW is discussed in the paper's conclusion.

	Succ.	avg gap‰		opt%		avg time		⋂VLSI		⋂Complete	
Alg	%		⋂		⋂		⋂	gap‰	time	gap‰	time
TM	100.0	74.46	65.93	10.7	16.5	0.26	0.11	26.62	0.06	262.23	0.77
AC3	99.8	35.88	30.84	21.2	31.8	2.44	0.12	5.48	0.07	87.24	0.82
RC3	99.8	35.56	30.36	20.8	31.5	3.64	0.12	5.60	0.06	88.64	0.82
LC3	99.2	41.06	36.57	18.1	27.1	20.62	0.14	22.11	0.07	87.87	1.00
LP3	92.8	36.38	34.36	21.5	32.1	29.22	0.18	6.39	0.08	99.73	1.21
AC4	90.5	23.17	20.63	25.8	38.1	66.35	3.63	3.31	19.53	53.72	2.43
RC4	90.5	22.46	19.46	25.2	36.8	66.86	3.61	3.28	19.57	54.07	2.34
LC4	90.5	29.15	27.40	20.2	30.3	67.78	3.64	19.42	19.37	53.69	2.89
LP4	86.1	21.98	21.32	33.1	46.6	67.90	3.71	1.60	19.42	67.10	2.63
AC5	79.5	17.28	15.29	26.9	40.6	170.85	16.14	4.09	60.21	36.72	11.86
RC5	79.5	16.60	14.62	27.0	40.3	168.60	16.37	2.86	60.74	36.79	11.82
LC5	79.0	25.50	24.24	20.9	31.1	158.86	17.35	18.86	60.75	36.70	12.62
LP5	74.9	16.43	15.49	38.1	54.4	104.55	17.04	1.03	61.55	46.26	13.75
AC6	67.4	13.50	12.11	27.7	43.5	197.90	154.59	4.12	339.96	23.77	93.84
RC6	67.5	13.19	11.76	27.2	42.7	197.36	156.66	2.55	346.44	24.09	92.57
LC6	67.5	23.66	22.23	21.0	33.1	199.67	155.93	18.92	343.81	24.70	96.86
LP6	64.8	13.63	12.95	39.8	62.1	172.32	159.63	0.40	347.23	35.86	99.02
BCS	88.2	0.00	0.00	88.2	100.0	94.78	44.40	0.00	92.33	0.00	94.25
PUW$_0$	100.0	16.04	15.04	23.3	35.3	(0.06)	(0.03)	6.71	0.01	9.53	0.25
PUW$_2$	100.0	6.27	4.61	41.3	59.7	(0.22)	(0.09)	1.64	0.03	5.01	0.62
PUW$_8$	100.0	0.54	0.07	84.7	98.0	(16.08)	(5.57)	0.02	1.73	0.01	38.87

improvement for 35.5 % (55.7 %, 60.5 %) of the instances with $\chi < 1$ ($\chi \leq 0.9$, $\chi \leq 0.8$), and significant degradation only for 14.1 % (12.7 %, 12.8 %, resp.).

Overall, we have seen that the (rather unpredicable) effect of different storage orders dominates the effect of preprocessing on the solution quality. Preprocessing achieves (slight) improvements on average, but the evidence is not strong enough to conclude that preprocessing improves the gaps significantly. We can, however, accept the hypothesis that, overall, preprocessing does not significantly degrade the expected solution quality. Hence, preprocessing can and should be applied.

Higher *k*. We have run experiments on all SteinLib instances using all algorithms described in Section 2 with $k \leq 6$ and preprocessing enabled.

Table 1 summarizes the results. For a simpler comparison between different algorithms and values for k, we also report the respective numbers on the instance subset that was solved by *every* considered algorithm in the table. This subset ('⋂') contains only 63.25 % of the SteinLib, with average $\chi = 70.22$ %. Be aware that it contains, essentially, instances that are relatively 'simple' for the approximation algorithms.

The overall success rate of the exact BCS is 88.2 % and an average successful run takes approximately 95 seconds. Note that the much simpler BC, obtains an overall success rate of 79.1 % with 90 seconds running time on average in the successful cases.

Almost all algorithms (except for LP5) could be considered practical for up to $k = 5$ where they achieve a success rate comparable to BC, trading solution quality for speed. For $k = 6$, however, the success rates drop harshly below BC's. Considering BCS, the approximation algorithms' success rates are only comparable up to $k = 4$. For any given k, success rates and times are comparable among all algorithms. In other words, the time to generate the set of all k-restricted full components dominates the running time.

We observe that higher k yield smaller gaps (as suggested in theory for RC, LC and LP) in practice. There are interesting things that can be observed:

- RC is the best, and LC is, by far, the worst choice among all strong approximation algorithms. This is remarkable since, according to theoretical bounds, RCk is the worst algorithm for $k \leq 18$, and LCk is the best for $6 \leq k \leq 14$.
- LP has the highest chances to find the optimum. Despite this, it has a worse average gap and a much worse success rate than RC, i. e., if it fails to find the optimum, its solution is either quite weak or non-existent at all.
- AC's gaps decrease with increasing k and are comparable to those of RC and LP. However, for AC we only know the approximation ratio 11/6 for $k \geq 3$. According to theoretical bounds, it should be the best one for $k \leq 5$.

Dependency on Instance Classes. Most of our observations on general algorithmic behavior is surprisingly consistent over the different instance groups of SteinLib. There are a couple of notable exceptions and observations, however.

When an instance has high *coverage* $|R|/|V|$, the preprocessing is typically much stronger, cf. Fig. 4. This consequently leads to better average gaps and running times. Generally, the instance class and the obtained χ are very strongly correlated. While SteinLib's rectilinear instances have high coverage, the VLSI and wire-routing instances, for example, have low.

Most interestingly, the density of an instance has a big influence on the algorithms' behaviors. On average, the LC algorithms give gaps 1.5–2 times larger than those of the other strong approximations. For sparse instances ($|E| \leq 2 \cdot |V|$) this worsens to factors 2–3. On sparse instances and for larger k, LP's gaps become at least as strong as RC's. VLSI instances are particularly extreme: LP's gaps are the by far strongest, while LC's gaps are over 10-fold. For complete instances, however, the picture changes drastically: we observe that LC becomes comparable to RC and AC, while LP's gaps deteriorate. Table 1 shows the average gaps and running times of VLSI ($\chi = 89.81\,\%$) and complete ($\chi = 74.80\,\%$) instances, restricted to the '$\bigcap$'-instances.

5 Conclusion

We showed that the efficient Steiner tree preprocessings do not degrade but also not significantly improve the observable solution quality. Using preprocessing—and the efficient Dreyfus-Wagner type enumeration due to [5]—allowed us to raise the bar of considered values for k. We may now say that $k \leq 5$ is applicable in practice. The probably biggest surprises for the larger values of k are that (a) the (comparably old) $\mathsf{RC}k$ continues to perform best in practice (this was previously only established for the special case $k = 3$); (b) $\mathsf{AC}k$'s solution quality improves in lockstep with $\mathsf{RC}k$ and $\mathsf{LP}k$, despite the fact that its proven approximation ratio does not change for larger k; and (c) for identical k, $\mathsf{LC}k$ gives consistently clearly weaker results than any of the other approximations.

Although not the focus of this research, it remains to briefly discuss the overall practicality of the strong approximations. Therefore, we can compare to the currently practically strongest algorithms as presented in the recent DIMACS challenge [6][4]. In Table 1 we also report on the strong and fast heuristic PUW_i [17], where the number of iterations is 2^i. We see that the heuristic clearly dominates all approximations by orders of magnitudes. For exact approaches, we have considered a simple branch-and-cut approach BC, as well as its more sophisticated variant BCS from [8]. We may also consider the exact results from [19]. For them, we only have detailed data over a subset of 434 SteinLib instances that have been considered particularly interesting. Over those, BCS achieves a success rate of 75.1 % and average running time of 122 seconds, while [19] achieve 89.4 % and 9.5 seconds, respectively. We can hence deduce that the current theoretically strongest approximation algorithms are neither competitive to state-of-the-art heuristics nor to exact approaches.

Acknowledgements. We thank Mihai Popa for implementations of reduction tests, and Google for funding him through the Google Summer of Code 2014 program. We also thank the authors of [8] for making their exact solver available, and Renato Werneck for detailed logs of their experiments in [17].

References

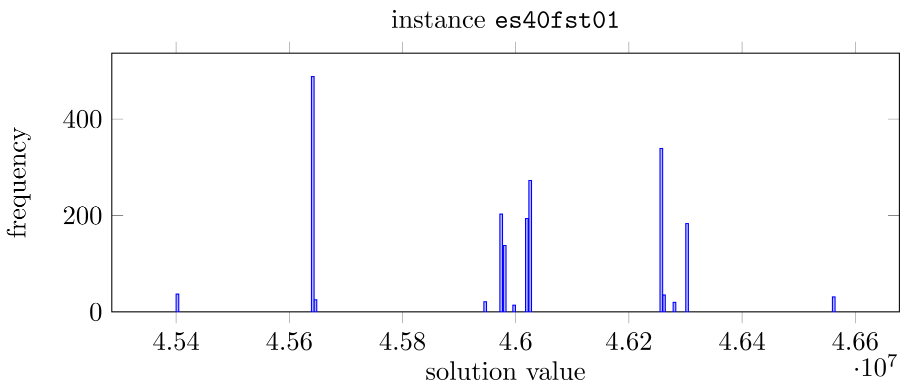
1. Beyer, S., Chimani, M.: Steiner Tree 1.39-Approximation in Practice. MEMICS'14, LNCS 8934, 60–72, Springer, 2014.
2. Beyer, S., Chimani, M.: Strong Steiner Tree Approximations in Practice. Submitted to journal, available at `http://arxiv.org/abs/1409.8318`, 2014.
3. Byrka, J., Grandoni, F., Rothvoß, T., Sanità, L.: Steiner Tree Approximation via Iterative Randomized Rounding. Journal of the ACM 60(1), 6:1–33, 2013.
4. Chimani, M., Woste, M.: Contraction-Based Steiner Tree Approximations in Practice. ISAAC'11, LNCS 7074, 40–49, Springer, 2011.

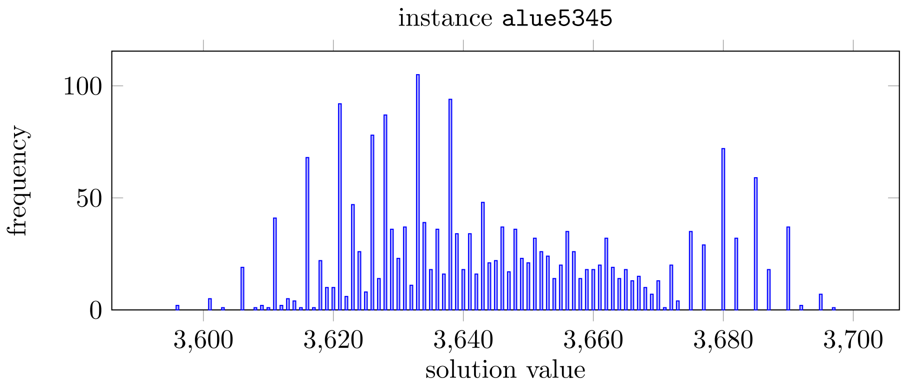
[4] Those results were obtained on different machines, with different preprocessing and memory limits. The machines are compared via the DIMACS benchmark [6]; higher is faster. We: 375, 16 GB limit. PUW [17]: 389, no (relevant) limit. [19]: 307, no limit. All success rates are reported with respect to a 1-hour time limit.

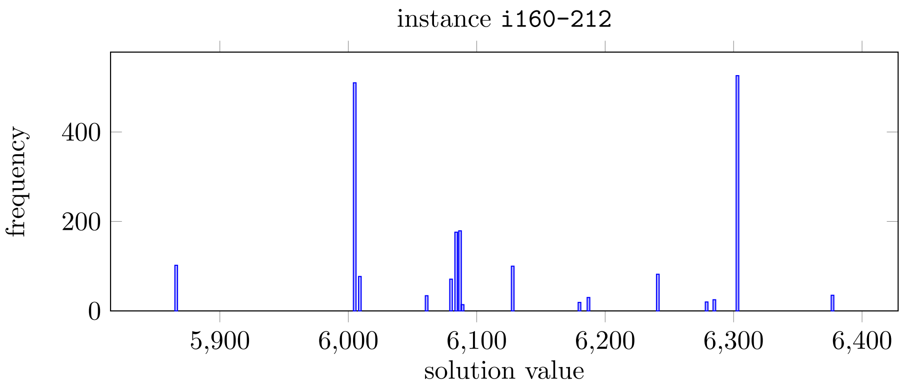
5. Ciebiera, K., Godlewski, P., Sankowski, P., Wygocki, P.: Approximation Algorithms for Steiner Tree Problems Based on Universal Solution Frameworks. arXiv abs/1410.7534, `http://arxiv.org/abs/1410.7534`, 2014.
6. 11th DIMACS Challenge. `http://dimacs11.cs.princeton.edu`. Bounds: 9/12/14.
7. Dreyfus, S.E., Wagner, R.A.: The Steiner Problem in Graphs. Networks 1(3), 195–207, 1971.
8. Fischetti, M., Leitner, M., Ljubic, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., Sinnl, M.: Thinning out Steiner trees: a node-based model for uniform edge costs. 11th DIMACS Challenge, 2014.
9. Goemans, M.X., Olver, N., Rothvoß, T., Zenklusen, R.: Matroids and Integrality Gaps for Hypergraphic Steiner Tree Relaxations. STOC'12, 1161–1176, ACM, 2012.
10. Jünger, M., Thienel, S.: The ABACUS System for Branch-and-Cut-and-Price Algorithms in Integer Programming and Combinatorial Optimization. Software: Practice and Experience 30, 1325–1352, 2000.
11. Koch, T., Martin, A., Voß, S.: SteinLib: An Updated Library on Steiner Tree Problems in Graphs. ZIB-Report 00-37, 2000. See also `http://steinlib.zib.de`.
12. Kou, L.T., Markowsky, G., Berman, L.: A Fast Algorithm for Steiner Trees. Acta Informatica 15, 141–145, 1981.
13. Ljubic, I., Weiskircher, R., Pferschy, U., Klau, G.W., Mutzel, P., Fischetti, M.: An Algorithmic Framework for the Exact Solution of the Prize-Collecting Steiner Tree Problem. Math. Program. 105(2–3), 427–449, 2006.
14. Mehlhorn, K.: A Faster Approximation Algorithm for the Steiner Problem in Graphs. Information Processing Letters 27(3), 125–128, 1988.
15. Papadimitriou, C.H., Yannakakis, M.: Optimization, Approximation, and Complexity Classes. STOC'88, 229–234, ACM, 1988.
16. Poggi de Aragão, M., Ribeiro, C.C., Uchoa, E., Werneck, R.F.: Hybrid Local Search for the Steiner Problem in Graphs. MIC'01, 2001.
17. Pajor, T., Uchoa, E., Werneck, R.F.: A Robust and Scalable Algorithm for the Steiner Problem in Graphs. 11th DIMACS Challenge, 2014.
18. Polzin, T., Vahdati Daneshmand, S.: Improved Algorithms for the Steiner Problem in Networks. Discrete Applied Mathematics 112(1–3), 263–300, 2001.
19. Polzin, T., Vahdati Daneshmand, S.: The Steiner Tree Challenge: An updated Study. 11th DIMACS Challenge, 2014.
20. Robins, G., Zelikovsky, A.: Tighter Bounds for Graph Steiner Tree Approximation. SIAM Journal on Discrete Mathematics 19(1), 122–134, 2005.
21. Takahashi, H., Matsuyama, A.: An Approximate Solution for the Steiner Problem in Graphs. Mathematica Japonica 24, 573–577, 1980.
22. Zelikovsky, A.: An 11/6-Approximation Algorithm for the Steiner Problem on Graphs. Annals of Discrete Mathematics 51, 351–354, Elsevier, 1992.
23. Zelikovsky, A.: A Faster Approximation Algorithm for the Steiner Tree Problem in Graphs. Information Processing Letters 46(2), 79–83, 1993.
24. Zelikovsky, A.: An 11/6-Approximation Algorithm for the Network Steiner Problem. Algorithmica 9(5), 463–470, 1993.
25. Zelikovsky, A.: Better Approximation Bounds for the Network and Euclidean Steiner Tree Problems. Tech. rep. CS-96-06, University of Virginia, 1995.

A Distribution of Solution Values

For each of the three specific instances below, we generated 2000 shufflings. The plots below show the distribution of the corresponding TM solution values.







B More Detailed Table for Influence of Preprocessing

The table below shows detailed information (number of instances, average χ in %, and statistical data on the gaps) averaged over the instance groupings proposed in [2]. Statistical data is: the mean μ of the gaps (averaged and how often it was <u>b</u>etter, <u>n</u>ot changed, or <u>w</u>orse); the deviation σ, and the skewness (how often it was negative, zero, or positive). All data is given for <u>orig</u>inal and <u>prep</u>rocessed instances; for each instance we consider 50 different random shufflings.

			avg μ		# μ			avg σ		# orig skew			# prep skew		
Group	#	avgχ	orig	prep	b	n	w	orig	prep	<0	$=0$	>0	<0	$=0$	>0
EuclidSparse	15	73.68	13.06	13.73	6	1	8	15.43	17.01	0	1	14	0	2	13
EuclidComplete	14	21.11	10.84	10.84	0	14	0	13.06	13.06	0	1	13	0	1	13
RandomSparse	96	30.41	25.52	22.26	64	12	20	21.58	19.00	1	11	84	1	24	71
RandomComplete	13	9.75	31.90	25.14	7	2	4	28.53	20.86	0	2	11	0	6	7
IncidenceSparse	280	84.44	133.47	127.36	154	50	76	53.82	50.68	2	2	276	3	2	275
IncidenceComplete	73	97.84	393.61	393.61	0	73	0	88.51	88.51	0	0	73	0	0	73
ConstructedSparse	9	85.57	143.19	141.86	5	1	3	50.48	51.54	0	1	8	0	1	8
SimpleRectilinear	218	46.45	16.35	12.53	158	17	43	16.86	13.90	1	22	195	0	37	181
HardRectilinear	54	64.27	23.04	19.40	52	0	2	20.88	19.30	0	0	54	0	0	54
VLSI / Grid	206	92.36	35.49	35.76	100	12	94	27.36	27.13	1	6	199	1	5	200
WireRouting	115	98.48	0.01	0.02	28	1	86	0.44	0.51	0	5	110	0	5	110
Coverage 10	531	86.77	73.09	71.42	223	84	224	33.69	32.75	3	16	512	4	20	507
Coverage 20	130	78.27	118.35	115.13	56	34	40	40.16	38.50	1	7	122	1	8	121
Coverage 30	127	78.92	184.86	179.50	62	41	24	55.90	52.67	0	4	123	0	7	120
Coverage 40	91	71.20	25.46	21.50	73	1	17	22.79	20.82	0	0	91	0	0	91
Coverage 50	119	45.51	16.15	12.46	90	5	24	17.17	14.32	0	1	118	0	9	110
Coverage 60	41	32.33	13.61	9.44	36	1	4	15.09	10.88	1	2	38	0	12	29
Coverage 70	18	14.77	8.01	3.60	12	4	2	9.62	6.04	0	8	10	0	8	10
Coverage 80	11	9.06	4.89	3.09	5	6	0	6.56	4.56	0	6	5	0	6	5
Coverage 90	14	5.87	3.47	1.95	11	2	1	7.32	5.02	0	2	12	0	4	10
Coverage 100	11	0.75	1.11	0.22	6	5	0	3.31	0.89	0	5	6	0	9	2
All	1093	73.15	75.69	72.86	574	183	336	32.32	30.53	5	51	1037	5	83	1005