Math. Prog. Comp. (2018) 10:69–118
https://doi.org/10.1007/s12532-017-0123-4

FULL LENGTH PAPER

A robust and scalable algorithm for the Steiner problem in graphs

Thomas Pajor[1] · **Eduardo Uchoa**[2] · **Renato F. Werneck**[3]

Received: 17 April 2015 / Accepted: 23 May 2017 / Published online: 4 September 2017
© Springer-Verlag GmbH Germany and The Mathematical Programming Society 2017

Abstract We present an effective heuristic for the Steiner Problem in Graphs. Its main elements are a multistart algorithm coupled with aggressive combination of elite solutions, both leveraging recently-proposed fast local searches. We also propose a fast implementation of a well-known dual ascent algorithm that not only makes our heuristics more robust (by dealing with easier cases quickly), but can also be used as a building block of an exact branch-and-bound algorithm that is quite effective for some inputs. On all graph classes we consider, our heuristic is competitive with (and sometimes more effective than) any previous approach with similar running times. It is also scalable: with long runs, we could improve or match the best published results for most open instances in the literature.

Keywords Steiner trees · Local search · Metaheuristics · Dual ascent

Mathematics Subject Classification 05C85 · 68R10 · 90C95

R. F. Werneck with work partly done at Microsoft Research Silicon Valley.

The software that was reviewed as part of this submission has been issued the Digital Object Identifier doi:10.5281/zenodo.806231.

✉ Renato F. Werneck
rwerneck@acm.org

Thomas Pajor
microsoft@tpajor.com

Eduardo Uchoa
uchoa@producao.uff.br

[1] Microsoft Research, New York City, NY, USA

[2] Universidade Federal Fluminense, Niterói, RJ, Brazil

[3] San Francisco, CA, USA

1 Introduction

Given an edge-weighted, undirected graph $G = (V, E)$ and a set $T \subseteq V$ of terminals, the Steiner Problem in Graphs (SPG) is that of finding a minimum-cost tree that contains all vertices in T. This has applications in many areas, including computational biology, networking, and circuit design [9]. Unfortunately, it is NP-hard not only to find an optimal solution [30], but also to approximate it within a factor of 96/95 [10]. The best known approximation ratio is 1.39 [8,24] (see [46] for a 1.55 approximation). Given its practical importance, there is a wealth of exact algorithms [12,18,25,31,33,38,40,41,45,51] and heuristics [3,6,17,18,37,44,45,52] to deal with real-world instances. State-of-the-art algorithms use a diverse toolkit that includes linear relaxations, branch-and-bound, reduction tests (preprocessing), and primal and dual heuristics.

Our goal in this paper is to develop an algorithm that is, above all, *robust*. For any input instance, regardless of its characteristics, we want to quickly produce a good solution. Moreover, the algorithm should *scale* well: when given more time to run, it should produce better solutions.

Our basic algorithm follows the principles of a heuristic proposed by Ribeiro, Uchoa, and Werneck [45]: it is a multistart algorithm with an evolutionary component, using perturbation for randomization. Under the hood, however, we introduce significant improvements that lead to much better results.

First, we leverage fast local search algorithms recently proposed by Uchoa and Werneck [52], which are asymptotically faster (in theory and practice) than previous approaches. Second, we propose a *cascaded combination* strategy, which immediately combines each fresh (newly-created) solution with multiple entries from a pool of elite solutions, leading to much quicker convergence. Third, we counterbalance this intensification strategy with a series of diversification measures (including careful perturbation and replacement policies in the pool) in order to explore the search space more comprehensively.

As a result, long runs of our algorithm can match or even improve the best published solutions (at the time of writing) for several open instances in the literature. For easier inputs, our basic algorithm still finds very good results, but for some graph classes it can be slower than alternative approaches that rely heavily on preprocessing and small duality gaps [12,40].

To make our overall approach more robust, we include some preprocessing routines. Moreover, we propose a *Guarded Multistart* algorithm, which runs a branch-and-bound routine at the same time as our basic (primal-only) algorithm. For easy instances, these two threads can help each other, often leading to drastic reductions in total CPU time. To compute lower bounds, we propose a novel and efficient implementation of a well-known combinatorial dual ascent algorithm due to Wong [58]. For several hard instances, our branch-and-bound routine finds provably optimal solutions faster than any published algorithm.

Even with these optimizations, there are important graph classes (such as some VLSI instances) in which our method is not as effective as other approaches, notably those based on advanced reduction techniques and linear programming [12,40] or on dynamic programming [25] (when the number of terminals is small). Even in such

cases, however, the solutions found by our approach are not much worse, confirming its robustness. Overall, our algorithm provides a reliable, general-purpose solution for the Steiner Problem in Graphs.

This paper is organized as follows. Section 2 explains our multistart algorithm. Section 3 discusses our lower-bounding techniques, including branch-and-bound. Section 4 shows how preprocessing and lower-bounding make our basic algorithm more robust. Section 5 has experiments, and we conclude in Sect. 6.

Notation The input to the Steiner Problem in Graphs is an undirected graph $G = (V, E)$ and a set $T \subseteq V$ of *terminals*. Each edge $e = (v, w)$ has an associated nonnegative *cost* (*length*) denoted by $cost(e)$ or $cost(v, w)$. A *solution* $S = (V_S, E_S)$ is a tree with $T \subseteq V_S \subseteq V$ and $E_S \subseteq E$; its cost is the sum of the costs of its edges. Our goal is to find a solution of minimum cost.

2 Basic algorithm

Our basic algorithm follows the multistart approach and runs in M iterations (where M is an input parameter). Each iteration generates a new solution from scratch using a constructive algorithm (with randomization), followed by local search. We also maintain a pool of elite solutions with good solutions found so far, including the very best one. The main feature of our algorithm is a *cascaded combination* strategy, which aggressively combines a new solution with several existing ones. This finds very good solutions soon, but has a very strong intensification effect. To counterbalance it, we look for diversification in other aspects of the algorithm. The outline of the entire algorithm is as follows:

1. Create an empty pool P of elite solutions with capacity $\lceil \sqrt{M/2} \rceil$.
2. Repeat for M iterations:
   (a) Generate a new solution S using a constructive algorithm, local search, and randomization.
   (b) Try to add S to the pool P.
   (c) Generate a solution S' by combining S with solutions in the pool P.
   (d) Try to add S' to the pool P.
3. Return the best solution in the pool P.

The remainder of this section describes details omitted from this outline. Section 2.1 explains the local search routines, Sect. 2.2 describes how fresh solutions are generated, Sect. 2.3 deals with the cascaded combination algorithm, and Sect. 2.4 addresses the insertion and eviction policies for the pool.

2.1 Local search

A local search algorithm tries to improve an existing solution S by examining a *neighborhood* $\mathcal{N}(S)$ of S, a set of solutions obtainable from S by performing a restricted set of operations. *Evaluating* $\mathcal{N}(S)$ consists of either finding an improving solution S' (i.e., one with $cost(S') < cost(S)$) or proving that no such S' exists in $\mathcal{N}(S)$. A local search

heuristic repeatedly replaces the current solution by an improving neighbor until it reaches a *local optimum*. Uchoa and Werneck [52] present algorithms to evaluate in $O(|E| \log |V|)$ time four natural (and well-studied) neighborhoods: Steiner-vertex insertion, Steiner-vertex elimination, key-path exchange, and key-vertex elimination.

The first two represent a solution $S = (V_S, E_S)$ in terms of its set $V_S \backslash T$ of *Steiner vertices*. The minimum spanning tree (MST) of the subgraph of G induced by V_S (which we denote by $\text{MST}(G[V_S])$) costs no more than S. In particular, if S is optimal, so is $\text{MST}(G[V_S])$. Uchoa and Werneck use dynamic graph techniques to efficiently evaluate neighborhoods defined by the insertion or removal of a single Steiner vertex [35,36,48,54]. These neighborhoods had been considered in metaheuristics before [3,44,45], but evaluation required $O(|V|^2)$ time for insertions and $O(|E||V|)$ time for removals.

The other two neighborhoods represent a solution S in terms of its *key vertices* K_S, which are Steiner vertices with degree at least three in S. If S is optimal, it costs the same as the MST of its *distance network* restricted to $K_S \cup T$ (the complete graph on $|K_S \cup T|$ vertices whose edge lengths reflect shortest paths in G). Uchoa and Werneck show that the neighborhood corresponding to the elimination of a single key vertex can be evaluated in $O(|E| \log |V|)$ time. They prove the same bound for the *key-path exchange* local search [14,17,53], which attempts to replace an existing key path (linking two vertices of $K_S \cup T$ in S) by a shorter path between the components it connects. Both implementations improve on previous time bounds by a factor of $O(|T|)$.

In this paper, we take the local searches mostly as black boxes, but use the fact that these implementations work in *passes*. If there is an improving move in the neighborhood, a pass is guaranteed to find one in $O(|E| \log |V|)$ time. To accelerate convergence, the algorithms may perform multiple independent moves in the same pass (within the same time bound). In practice, there are almost always fewer than 10 passes, with most of the improvements achieved early on [52].

We follow Uchoa and Werneck and use what they call the VQ local search within our algorithm. It alternates between a pass that evaluates Steiner-vertex insertion (V) and a pass that evaluates (simultaneously) both key-vertex removal and key-path exchange (Q). Since in practice Steiner vertex removal (U) is rarely better than key-vertex removal, our algorithm does not use it.[1]

2.2 Generating new solutions

New solutions are generated by a constructive algorithm followed by local search, using randomization. Instead of making our algorithms (constructive heuristic and local search) randomized, we follow Ribeiro et al. [45] and apply perturbations to the edge costs instead, preserving the running time guarantees of all algorithms.

[1] Uchoa and Werneck [52] state in passing that key-vertex removal dominates Steiner vertex removal, but this is not strictly true. Some local optima for key-vertex removal can be improved by Steiner vertex removal by transforming a degree-two vertex (on a key-path) into a key vertex. We found such cases to be extremely rare in practice, however.

Using the perturbation To build a constructive solution, we apply a random perturbation to the edge costs (details will be given later), then run a near-linear time (in practice) implementation [39] of the shortest-path heuristic (SPH) [49]. Starting from a random root vertex, SPH greedily adds to the solution the entire shortest path to the terminal that is closest (on the perturbed graph) to previously picked vertices.

Ribeiro et al. [45] suggest applying local search to the constructive solution, but using the original (unperturbed) costs during local search. Since the constructive solution can be quite far from the local optimum, however, the effects of the perturbation may disappear quite soon, hurting diversification.

We propose an alternative approach, which leverages the fact that our local searches work in passes. We start the local search on the perturbed instance and, after each pass it makes, we *dampen* the perturbation, bringing all costs closer to their original values. For each edge e with original (unperturbed) cost $cost(e)$, let $cost_i(e)$ be its (perturbed) cost at the end of pass i. For pass $i + 1$, we set $cost_{i+1}(e) = \alpha cost_i(e) + (1 - \alpha)cost(e)$, where $0 < \alpha < 1$ is a *decay factor* (we use $\alpha = 0.5$). This *dampened* approach makes better use of the guidance provided by the perturbation, thus increasing diversification. For efficiency, after three passes with perturbation, we restore the original (unperturbed) costs and run the local search until a local optimum is reached. Uchoa and Werneck [52] (Table 5) show that, on non-trivial graph classes, the local search takes between three and four passes (on average) to reach a local minimum (on the unperturbed graph). By making the local search operate on a (decreasingly) perturbed graph for the first three iterations, we improve diversity with minimal effect on the time to reach an (unperturbed) local optimum, as Sect. 5.1 will show.

Computing the perturbation We now return to the issue of how initial perturbations are computed. Ribeiro et al. propose a simple *edge-based* approach, in which the cost of each edge is multiplied by a random factor (between 1.0 and 1.2). This is reasonably effective, but has a potential drawback: because edges are independent, the perturbations applied to each edge incident to any particular vertex tend to cancel out one another. We thus propose a *vertex-based* perturbation, which associates an independent random factor to each vertex in the graph. The perturbed cost of an edge (u, v) is then its original cost multiplied by the average factors of its two endpoints (u and v).

To enhance diversification, the choice of parameters that control the perturbation itself is randomized. Each iteration chooses either edge-based or vertex-based perturbation with equal probability. It then picks a maximum perturbation Q uniformly at random in the range [1.25, 2.00]. Finally, it defines the actual perturbation factors: for each element (vertex or edge), it sets the factor to $1 + \rho Q$, where ρ is generated (for each element) uniformly at random in [0, 1].

For further diversification, we actually use a slightly non-uniform distribution parameterized by a small threshold $\tau = (\log_2 n)/n$. If the random number ρ is at least τ (as is usually the case), we use the formula above. Otherwise, we use a perturbation factor of ρ/τ. Note that this factor is between 0.00 and 1.00, while the standard factor is always between 1.25 and 2.00. This means that a small fraction of the elements can become significantly cheaper than others, and are thus more likely to appear in the solution. This allows us to test key-vertices that our standard local searches would

not normally consider, since they do not include a fast implementation of key-vertex insertion.

We stress that the algorithm already works reasonably well with the standard edge-based perturbation proposed by Ribeiro et al. [45]; although we observed some improvement with the vertex-based perturbation (and the non-uniform distribution), the effects were relatively minor. Given that these these alternatives are quite simple to implement, they are still worth using.

2.3 Cascaded combination

The cascaded combination algorithm takes as input an initial solution S_0, the pool of elite solutions, and the *maximum number of allowed failures*, denoted by ϕ (we use $\phi = 3$). The procedure combines S_0 with elements in the pool, generating a (potentially better) solution S^*.

The basic building block of this procedure is the *randomized merge* operation [45], which takes as input two solutions (S_a and S_b) and produces a third (potentially cheaper) one. It does so by first generating a perturbed graph from G by perturbing each edge cost depending on which of the original solutions (S_a and/or S_b) it appears in. If an edge appears in both solutions, it keeps its original cost. If it appears in none of the solutions, its cost is multiplied by 1000. If it appears in exactly one solution (S_a or S_b), its cost is multiplied by a random number between 100 and 500. We run the SPH heuristic on the resulting instance. Note that, regardless of graph size, the combined solution preserves all edges that appear in both S_a and S_b (since they are relatively much cheaper); the remaining edges are likely to come from either S_a or S_b (since such edges are still significantly cheaper than those that appear in neither solution). We then remove all perturbations and apply local search to the combined solution, producing S_c, the result of the perturbed combination.

The cascaded combination procedure maintains an incumbent solution S^*, originally set to S_0. In each step, it performs a randomized merge of S^* and a solution S' picked uniformly at random from the pool. Let S'' be the resulting solution. If $cost(S'') < cost(S^*)$, we make S'' the new incumbent (i.e., we set $S^* \leftarrow S''$). Otherwise, we say that the randomized merge *failed* and keep S^* as the incumbent. When the number of failures reaches ϕ, the cascaded combination algorithm stops and returns S^*.

Note that the resulting solution S^* may have elements from several other solutions in the pool. This makes it a powerful intensification agent, helping achieve good solutions quite quickly. That said, the first few solutions added to the pool will have a disproportionate influence on all others, potentially confining the multistart algorithm to a very restricted region of the search space. This is why we prioritize diversification elsewhere in the algorithm.

On average, each multistart iteration touches a constant number of solutions in the pool. We set the capacity of the elite pool to $\Theta(\sqrt{M})$ to ensure that most pairs of elite solutions are (indirectly) combined with one another at some point during the algorithm. We set the precise capacity to $\lceil\sqrt{M/2}\rceil$, but the algorithm is not too sensitive to this constant; results were not much different with $\lceil\sqrt{M}\rceil$ or $\lceil\sqrt{M/4}\rceil$.

2.4 Pool management

We now address the insertion and eviction policies for the pool of elite solutions. When our algorithm attempts to add a solution S to the pool, we must consider three simple cases and a nontrivial one. First, if S is identical to a solution already in the pool, it is not added. Second, if the pool is not full and S is not identical to any solution, S is simply added. Third, if the pool is full and S is not better than any solution in the pool, S is not added.

The nontrivial case happens when the pool is full, S is different from all solutions in the pool, and S is better than the worst current solution. In this case, S replaces a solution that is at least as bad as S, with (randomized) preference for solutions that are similar to S. More precisely, we define the relative symmetric difference between S and S' as $\Delta(S, S') = |E(S) \cap E(S')|/|E(S) \cup E(S')|$; this is 0 if the solutions are identical, and 1 if they have no edge in common. When inserting S into the pool, we pick a solution to replace among all solutions S' that cost at least as much as S, with probability proportional to a score given by $1/((1 - \Delta(S, S'))^2)$. Note that solutions that are similar to S are much more likely to be picked; for example, if S and S' are nearly identical, the score is close to 1; if they share only half their edges, the score is close to 0.25. This technique (biased towards replacing similar solutions) has been shown to increase diversification for other problems [43].

2.5 Discussion

Although our algorithm borrows some elements from the multistart approach of Ribeiro et al. [45], Sect. 5.1.3 will show that our algorithm significantly outperforms theirs. This section outlines the most relevant similarities and differences between the two approaches.

Ribeiro et al. run a pure multistart routine, in which solutions generated by the end of each iteration are mostly independent from previous ones; these solutions are used to populate an elite pool. At the very end, all pairs of elite solutions in the pool are combined with one another. The resulting solutions populate a second pool (generation). This procedure is repeated until it reaches a generation whose average solution value does not improve upon the previous one. The algorithm makes extensive use of randomization: each iteration chooses between three types of constructive algorithm, three types of perturbation (used to randomize the construction), and two local search schemes. The post-optimization phase chooses between two types of combination (randomized merges or path relinking by complementary moves).

Although there are many differences between the algorithms, the main reason for our good performance is that we leverage asymptotically faster implementations of the local searches. Like us, Ribeiro et al. use local searches based on adding and removing Steiner vertices and a local search based on key-path exchanges (we also add a local search based on key-vertex removals). While they must explicitly build each neighboring solution in linear or quasi-linear time, we can evaluate them implicitly in logarithmic amortized time by applying the recent algorithms from Uchoa and Werneck [52]. This allows us to reach local optima much faster in practice. Our

local optima tend to be slightly worse, however, since explicitly building neighboring solutions allows Ribeiro et al. to look for "opportunistic" moves (such as removing non-terminal leaves) during the evaluation. As Sect. 5.1.3 will show, this is a worthy trade-off within the overall multistart algorithm, allowing us to find better results within the same time limit.

Faster local searches allow us not only to run more iterations within the same time limit, but we can also do more in each iteration using cascaded combinations, another contribution of this paper. Both cascaded combinations and the algorithm of Ribeiro et al. use randomized merges as building blocks. While Ribeiro et al. perform randomized merges only at the very end, cascaded combinations (as proposed in this paper) use randomized merges much more aggressively, with combinations happening during the multistart phase itself. The result of one iteration is thus heavily dependent on others. As our experiments will show (see Fig. 2), this allows very good solutions to be generated much earlier in the execution.

Despite being less relevant to explain the performance gap, there are several other differences between the algorithms. We use a single constructive algorithm (Ribeiro et al. choose between three in each iteration) and a single local search scheme (Ribeiro et al. alternate between two). We use a difference-based replacement scheme when updating the pool (Ribeiro et al. do not). We alternate between vertex-based and edge-based perturbations, while Ribeiro et al. use only the latter. We stress that these are minor differences, however. The parameters we reported are the ones we ended up using in the final version of our code, but Sect. 5.1 will show that our algorithm is not very sensitive to these parameters.

3 Lower bounds

We now turn our attention to finding lower bounds. We propose an efficient implementation of a known greedy combinatorial algorithm (due to Wong [58]) associated with a powerful linear programming formulation. We first describe the formulation and an abstract version of the algorithm, then discuss our implementation.

3.1 Formulation and dual ascent

We use the dual of the well-known directed cut formulation for the Steiner problem in graphs [58]. It takes a terminal $r \in T$ as the root. A set $W \subset V$ is a *Steiner cut* if W contains at least one terminal but not the root. Let $\delta^-(W)$ be the set consisting of all arcs (u, v) such that $u \notin W$ and $v \in W$.

The dual formulation associates a nonnegative variable π_W with each Steiner cut W. The set $\mathcal{W}$ of all Steiner cuts has exponential size. Given a dual solution π, the *reduced cost* $\bar{c}(a)$ of an arc a is defined as $cost(a) - \sum_{W \in \mathcal{W}: a \in \delta^-(W)} \pi_W$. The dual formulation maximizes the sum of all π_W variables, subject to all reduced costs being nonnegative. An arc whose reduced cost is zero is said to be *saturated*.

The technique we use for finding lower bounds is a *dual ascent* routine proposed by Wong [58], which finds a greedy feasible solution to the dual formulation. The algorithm maintains a set of C of *active terminals*, initially consisting of $T \backslash \{r\}$. For a

terminal $t \in T$, let $cut(t)$ be the set of vertices that can reach t through saturated arcs only. We say that t induces a *root component* if t is the only active terminal in $cut(t)$.

The algorithm (implicitly) initializes with 0 the variables associated with all Steiner cuts. In each iteration, it picks a vertex $v \in C$ and checks if $cut(v)$ is a root component. If it is not, it makes v inactive and removes it from C; if it is a root component, the algorithm increases $\pi_{\delta^-(cut(v))}$ until one of its arcs is saturated (and keeps v in C). We stop when C becomes empty, at which point every terminal can be reached from the root using only saturated arcs.

Each iteration takes $O(|E|)$ time to process a candidate root component. There are $O(|T|)$ unsuccessful iterations, since each reduces the number of active vertices. A successful iteration saturates at least one arc and increases the size (number of vertices) of at least one root component. There can thus be at most $\min\{|V||T|, |E|\}$ such iterations, bounding the total running time by $O(|E| \min\{|V||T|, |E|\})$ [15].

3.2 Our implementation

We now describe details of our implementation of Wong's algorithm that are crucial to its good performance in practice.

Processing a root component Once a vertex v is picked at the beginning of an iteration, we process its (potential) root component in three passes.

The first pass performs a graph search (we use BFS) from v following only saturated incoming arcs. If the search hits another active vertex, the iteration stops immediately: v does not define a root component. Otherwise, the search finishes with two data structures: a set S consisting of all vertices in $cut(v)$, and a list L which includes all unsaturated arcs (a, b) such that $a \notin cut(v)$ and $b \in cut(v)$. To ensure both structures can be built during the BFS, we allow L to also contain some unsaturated arcs (a, b) such that both a and b belong to $cut(v)$. These may appear because, when the BFS scans b, it may not know yet whether its neighbor a is part of $cut(v)$; to be safe, we add (a, b) to L anyway.

The second pass traverses L with two aims: (1) remove from L all arcs (a, b) that are invalid (with $a \in cut(v)$); and (2) pick, among the remaining arcs, the one with the minimum residual capacity Δ.

The third pass performs an augmentation by reducing the residual capacity of each arc in L by Δ. It also builds a set X with the tails of all arcs that become saturated, which will be part of the new root component of v (after augmentation).

Note that only the first pass of the algorithm performs a graph search; the other two passes are much cheaper, as they merely traverse arrays.

Selection rules and lazy evaluation The bound given by the algorithm depends on which active vertex (root component) it selects in each iteration. Without loss of generality, we assume each iteration picks the active vertex that minimizes some *score* function. Poggi de Aragão, Uchoa, and Werneck [38] (see also [56]) found that using the number of incident arcs as the score works well in practice. Polzin and Vahdati Daneshmand [12,40] show that a related (but coarser) measure, the number of vertices

in the component, also works well. For either score function (and others), the main challenge is to maintain scores efficiently for all root components during the algorithm, since augmenting on one root component may affect several others.

For efficiency, we focus on *nondecreasing* score functions: as the root component grows, its score can either increase or stay the same. This allows us to use *lazy evaluation*. We maintain each active vertex v in a priority queue, with a priority $\sigma(v)$ that is a lower bound on the score of its root component. Each round of the algorithm removes the minimum element t from the queue. It then verifies (using the procedure above) if t defines a root component; if it does not, we just discard t. Otherwise, we perform the corresponding augmentation as long as the actual score is not higher than the priority of the second element in the queue. Finally, we reinsert t into the priority queue.

When we do augment, computing the new exact score of t can be expensive. Instead, we update the score assuming the vertices in the root component are the union of X (the tails of all arcs saturated during the augmentation) and the original vertices (at the beginning of the iteration). Although we may miss some vertices, this is relatively cheap to compute and provides a valid lower bound on the actual score.

Since the number of arcs incident to a root component may decrease, we cannot use it as score function. Instead, we use a refined version of the number of vertices in the root component. Given a component c, let $\text{vc}(c)$ be its number of vertices and let $\deg(c)$ be the sum of their in-degrees. We use $\deg(c) - (\text{vc}(c) - 1)$ as the score. This is an upper bound on the number of incoming arcs on the component (the $\text{vc}(c) - 1$ term discards arcs in a spanning tree of the component, which must exist). This function is nondecreasing, as cheap to compute as the number of vertices, and gives a better estimate on the number of incoming arcs.

Eager evaluation Even with lazy evaluation, we may process a root component multiple times before actually performing an augmentation (or discarding the component). To make the algorithm more efficient, we also use *eager evaluation*: after removing a component from the priority queue, we sometimes perform an augmentation even its real score does not match the priority in the queue. More precisely, as long as the actual score is no more than 25% higher than the priority, the augmentation is performed. This has almost no effect on solution quality but makes the algorithm significantly faster. Note that any constant factor (including 25%) implies a logarithmic bound on the number of times any component can be reevaluated during the algorithm. The maximum time spent reevaluating components (without subsequent augmentations) is thus $O(|E||T|\log|V|)$.

Last component Typically, the initial cuts found by the dual ascent algorithm have much fewer vertices than later ones. In particular, when there is only one active vertex v left, we may have to perform several expensive augmentations until it becomes reachable from the root. We can obtain the same bounds faster by dealing with this case differently: we run (forward) Dijkstra's algorithm from the root to v, using reduced costs as arc lengths. We then use a linear pass to update the reduced costs of the remaining arcs appropriately [56].

3.3 Branch-and-bound

We use our dual ascent algorithm within a simple branch-and-bound procedure. We follow the basic principles of most previous work [12,38,40,56,58], using dual ascent for lower bounds and branching on vertices. The remainder of this section describes other features of our implementation.

The dual ascent root is picked uniformly at random (among the terminals) and independently for each node of the branch-and-bound tree.

To find primal (upper) bounds, we run the SPH heuristic on the (directed) subgraph consisting only of arcs saturated by the dual ascent procedure, using the same root. We then run a single pass of the Steiner vertex insertion and elimination local search procedures (using all edges, not just saturated ones).

We branch on the vertex that has maximum degree in the primal solution found in the current branch-and-bound node. In case of ties, we look beyond the current primal solution and prefer vertices that maximize the sum of incoming saturated arcs, outgoing saturated arcs, and total degree. Remaining ties are broken at random. If v is the chosen vertex, we remove it from the graph on the "zero" side and make it a terminal on the "one" side. We traverse the branch-and-bound tree in DFS order, visiting the "one" side first. This tends to find good primal solutions quicker than other approaches we tried.

We can eliminate an arc (u, v) if its reduced cost is at least as high as the difference between the best known primal solution and the current dual solution. We actually take the *extended reduced cost* [41], which also considers the distance (using reduced costs) from the root to u. Since the root can change between nodes in the branch-and-bound tree, we only eliminate an (undirected) edge when both corresponding arcs (directions) could be fixed by reduced cost. If we eliminate at least $|E|/5$ edges in a branch-and-bound node, we create a single child node rather than branching.

4 Improving robustness

While the algorithm from Sect. 2 works well on many graph classes, there are still opportunities to make it more robust (compared to other approaches) for very easy or very hard instances. Section 4.1 shows how the lower bounds described in Sect. 3 allow our heuristic to stop sooner. Section 4.2 describes some basic preprocessing techniques to reduce the size of the graph on certain classes of instances. Finally, Sect. 4.3 describes a two-level version of the multistart algorithm that achieves greater diversification on longer runs.

4.1 Guarded multistart

Being a pure heuristic, the multistart algorithm described in Sect. 2 can be wasteful. Because it cannot prove that the best solution it found is optimal (even if it actually is), it cannot stop until it completes its scheduled number of iterations. Ideally, we would like to stop sooner on easy instances.

To that end, we propose a *Guarded Multistart* (GMS) algorithm. It runs two threads in parallel: the first runs our standard multistart algorithm, while the second runs the branch-and-bound routine from Sect. 3.3. The algorithm terminates as soon as either the multistart thread completes its iterations, or the branch-and-bound thread proves that the incumbent solution is optimal. Communication between threads is limited: the threads inform one another about termination and share the best incumbent solution.

The benefits of this approach are twofold. First, as already mentioned, on easy instances the branch-and-bound algorithm can often prove optimality well before the scheduled number of multistart iterations is reached, making the algorithm faster. Moreover, sometimes the branch-and-bound algorithm quickly finds better primal solutions by itself, leading to better quality as well.

Of course, these advantages are not free: the cycles spent on the branch-and-bound computation could have been used for the multistart itself. On harder instances, we can only afford to perform roughly half as many iterations within the same CPU time. We could make this problem less pronounced using a simple heuristic to detect cases in which the branch-and-bound computation is obviously unhelpful: if its depth reaches (say) 100, we could stop it and proceed only with the multistart computation, saving CPU time. Another potential drawback is nondeterminism: due to scheduling, multiple runs of GMS (even with the same random seed) may find different results. Although one could make the algorithm deterministic by carefully controlling when communication occurs, it is not clear this is worth the extra effort and complexity.

Similarly, it is possible that greater integration between the algorithms (for instance, by tentatively adding branch-and-bound solutions to the multistart pool) would help. In practice, however, we found that the greatest advantage of GMS is saving time on easy instances by allowing the algorithm to stop sooner. Since the primal solutions found by dual ascent tend to be of lower quality for harder instances (with larger duality gaps), it is unlikely that greater integration would justify the extra effort.

4.2 Reduction techniques

To be competitive with state-of-the-art algorithms on standard benchmark instances, we must deal with "easy" inputs effectively. We thus use some basic reduction (preprocessing) techniques that transform the input into a potentially much smaller instance with the same solution.

In particular, we delete non-terminal vertices of degree one (alongside their incident edges). Also, if there is a non-terminal vertex v with exactly two neighbors, u and w, we replace edges (u, v) and (v, w) with a single edge (u, w) with cost $cost(u, w) = cost(u, v) + cost(v, w)$.

Finally, we implemented a limited version of the *Bottleneck Steiner Distance* [16] test, which states that an edge (u, v) can be removed from the graph if there is a (*bottleneck*) path P_{uv} between u and v such that (1) P_{uv} excludes edge (u, v) and (2) every subpath of P_{uv} without an internal terminal has length at most $cost(u, v)$. Identifying all removable edges can be expensive, so we consider only a couple of common (and cheap) special cases, which can be seen as restricted versions of existing algorithms [12,40].

The first case is simple. If the combined degree of u and v is small (up to 20), we scan both vertices looking for a common neighbor x such that $cost(u, x) + cost(x, v) \leq cost(u, v)$; we also check if a parallel (u, v) edge exists. This heuristic helps with some grid-like graphs, such as VLSI instances.

The second case may find more elaborate paths. We first use a modified version of Dijkstra's algorithm [13] to build the Voronoi diagram associated with the terminals of G [34,39,52]. For each vertex v, this structure defines $vb(v)$ (the closest terminal to v), $vd(v)$ (the distance from that terminal), and $vp(v)$ (the parent edge on the path from $vb(v)$). By looking at boundary edges of this Voronoi diagram, we compute the MST of the distance network of G [34,39,52]. Let E_t be the set of edges of this MST, each corresponding to a path in the original graph. Let $E_f \subseteq E$ be the set of *free edges*, i.e., original edges that neither are parents in the Voronoi diagram nor belong to a path in E_t.

We try to eliminate edges in E_f, using E_t and the Voronoi diagram to find bottleneck paths. To do so efficiently, we first sort the union of E_t and E_f in increasing order of cost, breaking ties in favor of entries in E_t. We also initialize a union-find data structure [50] with $|V|$ disjoint sets. We then traverse this list in order. Consider edge $e_i = (u, v)$. If $e_i \in E_t$, we join groups u and v in the union-find data structure. Otherwise (if $e_i \in E_f$) we delete e_i if all of the following three conditions hold: (1) u and v belong to the same component in the union-find data structure; (2) $dist(vb(u), u) \leq cost(e_i)$; and (3) $dist(vb(v), v) \leq cost(e_i)$. This Voronoi-based test is particularly effective on random and dense Euclidean graphs.

4.3 Two-phase multistart

Even with all the measures we take to increase diversification, our multistart algorithm can still be strongly influenced by the first few solutions it finds, since they will be heavily used during cascaded combination. If the algorithm is unlucky in the choice of the first few solutions, it may be unable to escape a low-quality local optimum.

When the number M of iterations is large (in the thousands), we obtain more consistent results with a two-phase version of our algorithm. The first phase independently runs the standard algorithm four times, with $M/8$ iterations each. The second phase runs the standard algorithm with $M/2$ iterations, but starting from a pool of elite solutions obtained from the union of the four pools created in the first phase. Note that the combined size of all pools in the first phase is $4\sqrt{M/16} = \sqrt{M}$, while the second-phase pool can hold only $\sqrt{M/4} = \sqrt{M}/2$ solutions. We thus take the elite solutions from the first phase in random order and try to add them to the (initially empty) final pool, using the criteria outlined in Sect. 2.4 to decide which solutions are kept.

We call this two-phase version of our multistart algorithm MS2. Since it has multiple independent starts, it is less likely than the single-phase version of our algorithm (which we call MS) to be adversely influenced by a particularly bad set of initial solutions.

4.4 Time-bounded algorithm

The standard version of our multistart algorithm (as described in Sect. 2) is parameterized by the number of iterations. In some real-world applications, one would like to have instead a *time-bounded* algorithm, whose goal is to find the best possible solution within a time budget τ. (This was in fact how the software competition from the 11th DIMACS Implementation Challenge [28] was set up.) Recall, however, that our multistart algorithm sets the size of the elite pool to be proportional to the square root of the number of iterations, which is unknown in the time-bounded setup. A solution would be to fix the size of the elite pool to some constant, and just stop the algorithm when it runs out of time.

We get better results by computing an initial estimate on the number of iterations in the obvious way: we run a *single iteration* of the multistart algorithm (i.e., constructive algorithm (SPH) followed by local search). Here we use actual edge weights, without perturbation. Let its running time by τ_1.

We then run the standard multistart algorithm described in Sect. 2, using an (initially empty) elite pool of size $\sqrt{M'}$. Here $M' = \tau/(2.5\tau_1)$ is an estimate on the number of iterations the algorithm can run within the time budget τ, meaning that we estimate that a standard iteration of the algorithm will take about 2.5 times the first one, which does not use combinations. This constant (2.5) was found empirically, and the algorithm is not too sensitive to it. Note that the algorithm actually stops when the time limit is reached, so its number of iterations may be lower or higher than M'.

The same approach can be used to make the two-phase MS2 algorithm time-bounded: we run a single iteration to get an estimate M' on the total number of iterations, then run MS2 parameterized by M'. In fact, we can even have an adaptive algorithm that picks either MS or MS2 for the main runs depending on the value of M'. As Sect. 5 will show, a good strategy is to use MS for $M' \leq 2048$ and MS2 otherwise.

5 Experiments

We implemented all algorithms in C++ and compiled them using Visual Studio 2013 (optimizing for speed). We ran our main experiments on a machine with two 3.33 GHz Intel Xeon X5680 processors running Windows 2008R2 Server with 96 GB of DDR3-1333 RAM. This machine scores 388.914 according to the benchmark code made available for the 11th DIMACS Implementation Challenge (http://dimacs11.zib.de/downloads.html). All runs we report are sequential, except those of the Guarded Multistart algorithm, which use two cores. In every case, we report total CPU times, i.e., the sum of the times spent by each CPU involved in the computation.

Our main experiments evaluate all 1437 instances available by August 1, 2014 from the 11th DIMACS Implementation Challenge [28] (accessible from http://dimacs11.zib.de/downloads.html) and report error rates relative to the best solutions published by then (http://dimacs11.zib.de/instances/bounds20140801.txt). Section 5.3 discusses recent developments.

For ease of exposition, we group the original series into *classes*, as shown in Table 1 (augmented from [52]). More detailed information about the dimensions of

Table 1 Classes of instances tested in our main experiments

CLASS	SERIES	DESCRIPTION
euclidean	x p4e p6e	Euclidean costs [11,31]
fst	es*fst tspfst cph14	Reduced geometric, L1 costs [27,28,55]
hard	bip cc hc sp	Synthetic hard instances [32,47]
incidence	i080 i160 i320 i640	Random graphs, incidence costs [15]
r	1r 2r	2D and 3D cross-grid graphs [21]
random	b c d e mc p4z p6z	Graphs with random costs [4,11,31]
vienna	gori gadv isim iadv	Road networks [33]
vlsi	alue alut dmxa diw gap lin msm taq	Planar grid graphs with holes [31,32]
wrp	wrp3 wrp4	Group Steiner grid instances [59]

the instances in each series can be found in Table 12, in the Appendix. Most instances are available from the SteinLib [32], with two exceptions: cph14 (graphs obtained from rectilinear problems [27]) and vienna (road networks from telecommunication applications [33]).

For experiments on the entire set of benchmark instances, we use a single (identical) random seed for each of the 1437 instances, since they are already quite numerous. Experiments restricted to a subset of the benchmark use multiple random seeds for each instance. These cases will be noted explicitly.

5.1 Multistart

In our first experiment, we ran the default version of our multistart algorithm on all instances from the DIMACS Challenge [28]. Recall that this version is not guarded (no branch-and-bound) and uses the lightweight preprocessing routine. We vary the number of multistart iterations from 1 to 256 (by factors of 4). Table 2 shows average running times (in seconds) and percent errors relative to the best known solutions (http://dimacs11.zib.de/instances/bounds20140801.txt). Error rates are also shown in Fig. 1. Results aggregated by series can be found in Tables 13 and 14, in the Appendix. To improve readability, errors smaller than 0.001% are shown multiplied by 1000 and in brackets. Such small errors are particularly common for wrp instances as a side effect of a reduction from the Group Steiner Tree problem, but occasionally appear for other classes (including euclidean and vienna).

Our algorithm is quite effective. With as little as 16 multistart iterations, the average error rate is below 0.5% on all classes except hard, which consists of adversarial synthetic instances. With 256 iterations, the average error falls below 0.5% for hard, and below 0.06% for all other classes. Average running times are still quite reasonable: the only outlier is vienna, which has much bigger graphs on average (see Table 12, in the Appendix). These instances are relatively easy: a single iteration is enough to find solutions that are on average within 0.1% of the best previously known bounds.

Table 2 Multistart algorithm: average running time in seconds and average percent error relative to the best known solution, with number of iterations (IT.) varying from 1 to 256

CLASS	TIME [s]					ERROR [%]				
	1 IT.	4 IT.	16 IT.	64 IT.	256 IT.	1 IT.	4 IT.	16 IT.	64 IT.	256 IT.
euclidean	0.005	0.011	0.036	0.135	0.520	0.155	0.009	[0.169]	OPT	OPT
fst	0.014	0.089	0.433	1.862	7.560	0.540	0.198	0.069	0.027	0.012
hard	0.047	0.340	1.855	7.626	30.377	4.499	2.726	1.607	0.918	0.492
incidence	0.146	0.445	1.757	7.253	29.496	2.608	0.967	0.374	0.141	0.043
r	0.012	0.042	0.166	0.688	2.698	4.292	1.628	0.445	0.133	0.056
random	0.009	0.027	0.104	0.424	1.687	1.104	0.217	0.049	0.014	0.009
vienna	0.532	3.880	19.822	85.522	356.237	0.074	0.035	0.017	0.008	[−0.061]
vlsi	0.045	0.234	1.086	4.654	18.902	0.809	0.271	0.082	0.028	0.011
wrp	0.015	0.080	0.353	1.480	5.932	[0.496]	[0.129]	[0.026]	[0.004]	[0.001]

Errors smaller than 0.001% are multiplied by 10^3 and shown in brackets; for example, entry [0.496] for wrp actually means 0.000496%

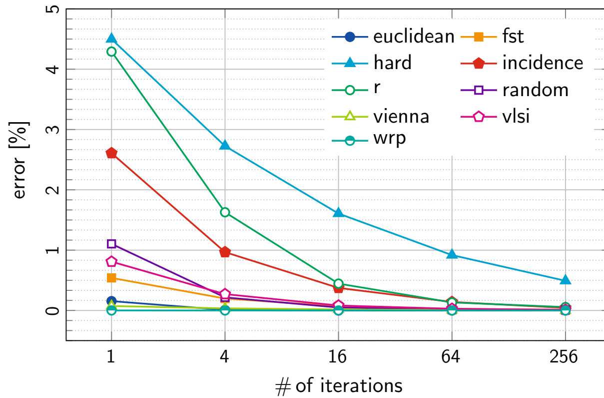

Fig. 1 Multistart algorithm: average error rates as the total number of iterations varies

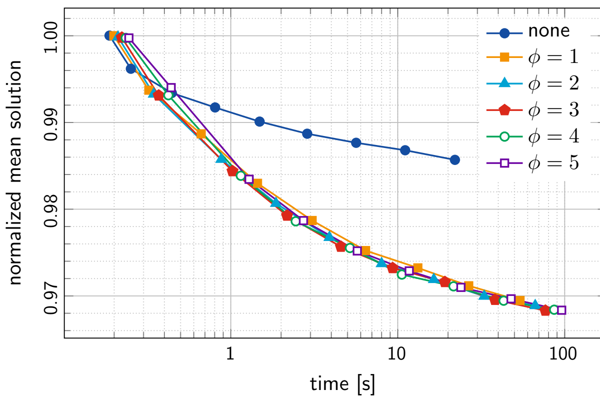

Fig. 2 Solution quality of the multistart algorithm (on 30 representative instances) as a function of the maximum number ϕ of failures within cascaded combination. Mean solution values are shown relative to a single iteration with no cascaded combination

One reason for the success of our approach is its use of cascaded combination. To confirm it is important, we tested six variants of our algorithm on 30 nontrivial representative instances chosen for the DIMACS Challenge competition. (The set is available at http://dimacs11.zib.de/contest/instances/SPG.tgz.) The variants differ only on the maximum number ϕ of failures allowed during cascaded combination. Figure 2 summarizes the results. There is one curve for each value of ϕ (0 to 5, with 0 meaning "no combination"), with different numbers of iterations $(1, 2, 4, \ldots, 256)$. Each point is the mean of 5 runs with different random seeds. The x-axis shows the (geometric) mean running time, whereas the y-axis represents the (geometric) mean solution value for all 30 instances, normalized so that one iteration with $\phi = 0$ has value 1.00 (the actual mean solution for $\phi = 0$ was 36,871.1873).

The figure shows that, even though cascaded combination significantly increases the time per iteration, it leads to much better solutions within the same allotted time.

We use $\phi = 3$ by default, but any small positive ϕ also works well—there is very little difference between the curves with $\phi > 0$.

In contrast to cascaded combination, other parameters we considered have only minor effect on solution quality. For instance, recall that each iteration picks either vertex-based or edge-based perturbation with equal probability. With 256 iterations, our default algorithm found a mean solution value (on the same set of 30 instances, with nine runs per instance) of 35,822.2. Using only edge-based perturbations increases this to 35,827.0; vertex-based perturbations increase it to 35,832.9. Running times are essentially the same in all three cases, and the difference in quality is negligible.

Similarly, using dampened perturbations (i.e., running up to three local search passes on the perturbed graph, with some decay) does improve quality, but only slightly. On the same set of instances (and nine seeds per instance), the mean solution value without dampened perturbations is 35,830.2, which is marginally higher than the baseline of 35,822.2. Running times typically increase by less than 10%.

5.1.1 Guarded multistart

We now consider the *Guarded Multistart* (GMS) algorithm from Sect. 4.1, which runs the standard multistart in parallel with a branch-and-bound algorithm. We tested GMS with 2, 8, 32, and 128 multistart iterations. Table 3 reports the error rates and average running times (see also Tables 15 and 16, in the Appendix). For consistency, we report total CPU times; since GMS uses two cores, the actual wall-clock time is lower.

The CPU time spent by GMS with i iterations cannot be (by design) much worse that the unguarded algorithm (MS) with $2i$ iterations. A comparison of Tables 2 and 3 shows that running times are indeed similar for several classes, such as hard, vlsi, and wrp. But GMS can stop much sooner on "easy" instances, when its branch-and-bound portion can prove the optimality of the incumbent. For random, incidence,

Table 3 Guarded multistart: average CPU time in seconds and average percent error relative to the best known solutions, with the number of iterations (IT.) set to 2, 8, 32, or 128

CLASS	TIME [s]				ERROR [%]			
	2 IT.	8 IT.	32 IT.	128 IT.	2 IT.	8 IT.	32 IT.	128 IT.
euclidean	0.009	0.010	0.011	0.010	[0.429]	OPT	OPT	OPT
fst	0.063	0.398	1.757	7.235	0.282	0.095	0.040	0.018
hard	0.304	1.704	7.313	30.016	3.094	1.709	1.070	0.648
incidence	0.403	1.012	2.617	4.400	0.360	0.065	0.030	0.019
r	0.036	0.117	0.399	1.456	0.876	0.346	0.118	0.056
random	0.021	0.050	0.127	0.481	0.119	0.044	0.014	0.004
vienna	3.663	19.604	87.347	358.916	0.058	0.024	0.011	0.003
vlsi	0.279	1.144	4.497	18.427	0.364	0.067	0.035	0.012
wrp	0.068	0.331	1.424	5.700	[0.175]	[0.033]	[0.008]	[0.002]

Errors in brackets (for euclidean and wrp series) are multiplied by 10^3; for example, the first such entry in the table ([0.429]) represents an error of 0.000429%

and especially euclidean, GMS becomes significantly faster than MS as the number of allowed iterations increases.

The relative solution quality of the two variants also depends on the type of instance. For classes such as hard and wrp, the guarded variant finds slightly worse results (within the same CPU time), since most of the useful computation is done by the multistart portion of the algorithm, which has fewer iterations to work with. For a few classes (such as incidence), the guarded version actually finds much better solutions, thanks to the branch-and-bound portion of the algorithm. In most cases, the difference is quite small. On balance, the guarded version is more robust and should be used unless there is reason to believe the branch-and-bound portion will be ineffective.

5.1.2 Comparison with Polzin and Vahdati Daneshmand

Table 4 compares Guarded Multistart (GMS) against the three state-of-the-art heuristics by Polzin and Vahdati Daneshmand [12,40]: PRUNE, ASCEND&PRUNE, and SLACK- PRUNE. As the authors report [12,40], these heuristics dominate the multistart approach by Ribeiro et al. [45].

Since the three algorithms have very different time/quality tradeoffs, we report results for GMS with 1, 8, 32, and 128 iterations. For consistency with how the results are reported in [12,40], Table 4 shows the lin series separately from the remaining vlsi instances. Running times for their algorithms are scaled (divided by 6.12) to roughly match our machine.[2]

The table shows that the algorithms have different profiles. Both PRUNE and ASCEND&PRUNE are quite fast, with running times comparable to GMS1, which runs a single multistart iteration. They provide much better solutions on series d and e, whereas GMS1 is significantly better on 1r, i080, i160, i320, and x. Error rates are usually within a factor of two of one another otherwise.

The SLACK- PRUNE algorithm usually finds better solutions, but takes much longer; it should then be compared with GMS with a few dozen iterations. The SLACK- PRUNE approach is superior when advanced reduction techniques (exploiting small duality gaps) work very well: 2r, e, and vlsi are good examples. When these techniques are less effective, our algorithm dominates: see es10000fst, i080, i160, i320, mc, and wrp, for example. Performance is comparable for several cases in between, such as es1000fst, lin, or tspfst.

Unfortunately, Polzin and Vahdati Daneshmand [12,40] only report results for series in which all optimal solutions are known, which consist mostly of small inputs or instances for which reduction techniques work well. It is encouraging that GMS is competitive even in the absence of very hard instances. This confirms that, although reduction and dual-based techniques are powerful, primal heuristics based on local search (the core of our approach) are essential for a truly robust algorithm.

[2] We know from http://www.cpubenchmark.net/singleThread.html that our machine is 3.111 times faster than an 1.53 GHz AMD Athlon XP 1800+, which is 1.967 times faster (based on Tables 5.3 and A.6 from [40]) than the 900 MHz SPARC III+ Sunfire 15,000 used in their main experiments.

Table 4 Comparison of various heuristics in terms of percent errors (multiplied by 10^3 for wrp, in brackets) and average running times in seconds

GROUP	PREVIOUS ALGORITHMS [12,40]						GUARDED MULTISTART							
	PRUNE		ASCEND&PRUNE		SLACK- PRUNE		GMS1		GMS8		GMS32		GMS128	
	ERROR	TIME	ERR.	TIME	ERROR	TIME	ERROR	TIME	ERROR	TIME	ERROR	TIME	ERROR	TIME
1r	1.360	0.021	1.030	0.011	OPT	0.036	0.662	0.014	0.080	0.056	OPT	0.147	OPT	0.407
2r	1.420	0.044	1.590	0.026	OPT	1.783	2.363	0.030	0.612	0.177	0.236	0.651	0.113	2.505
d	0.070	0.016	0.020	0.011	OPT	0.023	0.271	0.018	0.077	0.050	OPT	0.073	OPT	0.092
e	0.310	0.064	0.130	0.041	OPT	0.268	0.413	0.048	0.107	0.192	0.076	0.577	0.024	2.483
es10000fst	1.110	1.235	0.670	5.046	0.380	343.446	0.651	1.266	0.374	28.187	0.218	123.472	0.151	519.184
es1000fst	1.010	0.093	0.530	0.062	0.190	3.034	0.654	0.085	0.229	1.165	0.118	4.980	0.045	20.540
i080	1.150	0.011	1.650	0.003	0.060	0.070	0.349	0.008	OPT	0.011	OPT	0.011	OPT	0.011
i160	1.970	0.051	1.690	0.011	0.100	0.275	0.873	0.028	0.030	0.058	0.003	0.085	OPT	0.088
i320	2.840	0.266	1.810	0.049	0.140	1.219	1.099	0.140	0.075	0.362	0.040	0.658	0.027	1.566
lin	1.440	0.247	0.760	0.178	0.040	25.162	0.876	0.510	0.114	2.590	0.052	10.327	0.021	43.031
mc	1.700	0.008	1.010	0.007	0.420	0.155	0.192	0.009	0.181	0.031	OPT	0.072	OPT	0.085
tspfst	0.420	0.034	0.310	0.062	0.040	5.230	0.350	0.040	0.096	0.469	0.032	2.158	0.016	8.984
vlsi	0.390	0.054	0.350	0.065	0.004	1.172	0.555	0.064	0.053	0.683	0.030	2.637	0.009	10.579
wrp3	[0.6]	0.025	[0.3]	0.023	[0.03]	2.907	[0.218]	0.039	[0.039]	0.403	[0.012]	1.758	[0.003]	7.074
wrp4	[70]	0.016	[0.6]	0.016	[0.06]	1.011	[0.531]	0.027	[0.027]	0.259	[0.003]	1.085	OPT	4.303
x	0.170	0.072	OPT	0.047	OPT	0.038	0.006	0.024	OPT	0.045	OPT	0.045	OPT	0.045

The notation GMSx refers to our Guarded Multistart algorithm with $x \in \{1, 8, 32, 128\}$ iterations

5.1.3 Comparison with Ribeiro et al.

For completeness, we also compare our algorithm to the hybrid multistart heuristic of Ribeiro et al. [45], on which our approach builds. Both codes were compiled with g++ and full optimization and run on an Intel Core i7-6700K CPU with 24 GB of DDR4-2133 RAM; this machine scores 486.387 according to the benchmark routine from the 11th DIMACS Implementation Challenge. For this analysis, we consider the reference set of 30 representative instances from the DIMACS Challenge.[3] For a fair comparison, neither algorithm uses preprocessing.

We first compare (in Table 5) their performance with a single multistart iteration (i.e., a single constructive heuristic followed by local search until a local optimum is reached). It shows that the local search used by Ribeiro et al. [45] tends to find better results than the one we use (from Uchoa and Werneck [52]), since it can exploit opportunistic moves. Average solution values are 1.1% lower for Ribeiro et al.'s local search, and its advantage is clearer on instances with small solution values, which tend to have more ties. Due to its worse asymptotics, however, their approach is prohibitively slow for a robust general-purpose solver. We are hundreds of times faster on the larger instances tested (with a few tens of thousands of vertices), and never slower by much more than a factor of two. These results confirm the findings by Uchoa and Werneck [52].

Table 6 compares the same two algorithms, but now using 128 multistart iterations and a pool of 10 initial solutions, the setup recommended by Ribeiro et al. [45]. Once again, we used nine random seeds for almost all runs. For the six slowest instances (G106ac1, I064ac1, alue7080, es10000fst01, fnl4461fst, s5), we ran Ribeiro et al.'s code only once, since each run took from 4 h to more than a week (in fact, s5 failed to finish within 10 days). Although there are some cases in which Ribeiro et al. have marginally better results (notably some of the cc instances), our approach finds solutions that are 0.2% better on average, overcoming most of the limitations of its more strict local search. Its greatest advantage, however, is speed. On larger instances, it is orders of magnitude faster and still finds better solutions.

5.1.4 Long runs

To test the scalability of our algorithm, we consider the 41 SteinLib instances that had no published proof of optimality by August 1, 2014. We consider three versions of our algorithm. The baseline is MS, the multistart algorithm described in Sect. 2. MS2 is the two-phase version of MS, as described in Sect. 4.3. Finally, MSK augments plain MS by also using the *key-vertex insertion* local search implemented as calls to the SPH algorithm (as proposed in [37]); it is very expensive, but can potentially find

[3] The code from Ribeiro et al. [45] cannot handle instances G106ac and I064ac because they have a small number (454 and 6, respectively) of zero-length edges; we reset these edge lengths to 1 for this experiment, creating instances G106ac1 and I064ac1. This may increase the solution value by a negligible amount (no more than 0.001%).

Table 5 Comparison between a single multistart iteration (constructive algorithm followed by local search) by the heuristic of Ribeiro et al. [45] (RUW02) and our algorithm (MS) on a set of 30 representative instances

INSTANCE	AVERAGE TIME [ms]			AVERAGE SOLUTION		
	RUW02	MS	RUW02/MS	RUW02	MS	RUW02/MS
2r211c	**10**	15	0.68	**94,000**	95,556	0.9837
G106ac1	779,477	**2681**	290.72	37,191,295	**37,054,226**	1.0037
I064ac1	119,901	**1112**	107.87	186,952,803	**186,910,488**	1.0002
alue7080	20,551	**594**	34.61	**62,809**	62,997	0.9970
alut2625	8723	**557**	15.65	**35,740**	35,865	0.9965
bipa2p	224	**107**	2.09	38,136	**38,048**	1.0023
bipa2u	388	**80**	4.85	**363**	375	0.9693
cc12-2n	1106	**137**	8.09	**634**	674	0.9409
cc12-2p	1597	**298**	5.37	128,870	**128,309**	1.0044
cc12-2u	1808	**195**	9.27	**1208**	1276	0.9468
cc3-12n	**54**	85	0.64	**112**	113	0.9876
cc3-12p	**73**	171	0.43	19,623	**19,527**	1.0049
cc3-12u	**49**	107	0.46	**191**	197	0.9675
d18	121	**79**	1.54	**228**	233	0.9819
e18	1161	**342**	3.39	**575**	596	0.9639
es10000fst01	758,450	**1454**	521.77	**720,306,547**	722,273,483	0.9973
fnl4461fst	145,609	**740**	196.80	**183,808**	184,617	0.9956
hc12p	954	**288**	3.32	**245,033**	245,241	0.9992
hc12u	1668	**269**	6.21	**2352**	2384	0.9867
i640-211	**10**	20	0.47	**12,607**	12,675	0.9946
i640-314	**28**	32	0.87	**36,736**	36,743	0.9998
i640-341	490	**219**	2.23	32,524	**32,442**	1.0025
lin36	1207	**436**	2.77	**56,445**	56,687	0.9957
lin37	2199	**592**	3.72	**100,348**	100,996	0.9936
rc09	1102	**109**	10.15	**113,836**	114,466	0.9945
rt05	2762	**303**	9.12	**52,333**	52,839	0.9904
s5	145,583	**2053**	70.90	**25,210**	**25,210**	1.0000
w23c23	145	**32**	4.59	**701**	726	0.9658
world666	**837**	1067	0.78	**122,553**	122,620	0.9995
wrp3-83	**60**	82	0.74	**8,300,942**	8,300,964	1.0000

Running times (in milliseconds) and solution values are averages over nine runs
The best result in each category is marked in bold

better results. None of these variants is guarded, since branch-and-bound is ineffective on hard instances. To find near-optimal solutions, we test up to 262,144 (2^{18}) iterations (1024 for MSK). Since these experiments were very time-consuming, we could only afford to test each set of parameters with a single random seed; we used the same seed for all runs.

Table 6 Comparison between the heuristic of Ribeiro et al. [45] (RUW02) and our algorithm (MS) on a set of 30 representative instances

INSTANCE	AVERAGE TIME [s]			AVERAGE SOLUTION		
	RUW02	MS	RUW02/MS	RUW02	MS	RUW02/MS
2r211c	**2.6**	3.6	0.72	89,111	**89,000**	1.0012
G106ac1	820,877.6	**1173.0**	699.83	36,966,155	**36,948,036**	1.0005
I064ac1	104,692.5	**468.7**	223.34	186,872,140	**186,860,399**	1.0001
alue7080	14,385.6	**221.4**	64.98	62,591	**62,564**	1.0004
alut2625	5357.3	**177.6**	30.16	35,534	**35,502**	1.0009
bipa2p	68.0	**47.9**	1.42	36,656	**35,873**	1.0218
bipa2u	84.7	**31.9**	2.66	354	**346**	1.0231
cc12-2n	246.1	**42.7**	5.77	622	**619**	1.0053
cc12-2p	800.9	**116.5**	6.87	**123,103**	123,461	0.9971
cc12-2u	452.2	**68.9**	6.56	**1188**	1192	0.9971
cc3-12n	**13.9**	19.9	0.70	**111**	**111**	1.0000
cc3-12p	**25.1**	56.6	0.44	19,129	**18,956**	1.0091
cc3-12u	**20.6**	30.6	0.67	**186**	187	0.9957
d18	25.8	**18.8**	1.37	**224**	**224**	0.9991
e18	245.3	**93.9**	2.61	**566**	567	0.9984
es10000fst01	407,950.5	**583.1**	699.66	717,587,060	**717,359,577**	1.0003
fnl4461fst	67,366.1	**280.0**	240.62	182,800	**182,728**	1.0004
hc12p	1144.4	**180.9**	6.32	239,955	**239,146**	1.0034
hc12u	754.2	**154.2**	4.89	2334	**2316**	1.0076
i640-211	**2.4**	6.8	0.35	12,155	**12,114**	1.0034
i640-314	**9.1**	11.2	0.81	35,681	**35,679**	1.0001
i640-341	**81.9**	87.5	0.94	32,130	**32,077**	1.0017
lin36	740.3	**125.5**	5.90	55,665	**55,631**	1.0006
lin37	1754.3	**187.9**	9.34	99,841	**99,651**	1.0019
rc09	386.6	**42.8**	9.03	**111,374**	111,409	0.9997
rt05	1141.9	**108.1**	10.57	**51,506**	51,555	0.9990
s5	–	361.7	–	**25,210**	**25,210**	1.0000
w23c23	33.7	**11.9**	2.83	**696**	697	0.9986
world666	180.2	**170.7**	1.06	**122,467**	**122,467**	1.0000
wrp3-83	38.8	**21.2**	1.83	8,300,908	**8,300,906**	1.0000

Both algorithms are run with 128 iterations and 10 elite solutions. Running times (in seconds) and solution values are averages over up to nine runs. A dash ("–") indicates a run that did not finish within 10 days. The best times and solutions for each instance are marked in bold

For each variant (and number of iterations), Table 7 shows the (geometric) mean time in seconds and average error with respect to the best published solutions. Figure 3 represents the same data visually.

With 1024 iterations, MSK finds better results than other variants, but is much slower: increasing the number of iterations of either MS or MS2 to 16,384 is cheaper

Table 7 Results for three MS variants on 41 open SteinLib instances: (geometric) mean time in seconds and average percent error relative to the best known solution

METHOD	ITER.	TIME [S]	ERR. [%]
MS	1024	128.0	0.407
	4096	520.7	0.334
	16,384	2094.8	0.213
	65,536	8474.3	0.096
	262,144	35,056.4	0.107
MS2	1024	126.8	0.357
	4096	507.6	0.150
	16,384	2022.3	0.050
	65,536	8221.0	−0.020
	262,144	32,609.4	−0.094
MSK	64	271.6	0.623
	256	1096.3	0.360
	1024	4359.1	0.218

and leads to better solutions. Unsurprisingly, MS and MS2 have comparable running times for the same number of iterations. As argued in Sect. 4.3, increasing the number of iterations is more effective for MS2 than for MS. In fact, the average solution quality for MS does not even improve when we increase the number of iterations from 65,536 to 262,144. By starting from four independent sets of solutions, MS2 is less likely to be confined to a particularly bad region of the search space.

Considering all 13 runs from Table 7 (five runs each for MS and MS2, and three for MSK), there were only eight cases (out of 41) for which we could not at least match the best bound published by August 1, 2014. (See Table 17, in the Appendix.) In 19 cases, we found a strictly better solution. Most of these were found by MS2 (see Table 18, in the Appendix); MSK was better only for cc11-2u and cc12-2u.

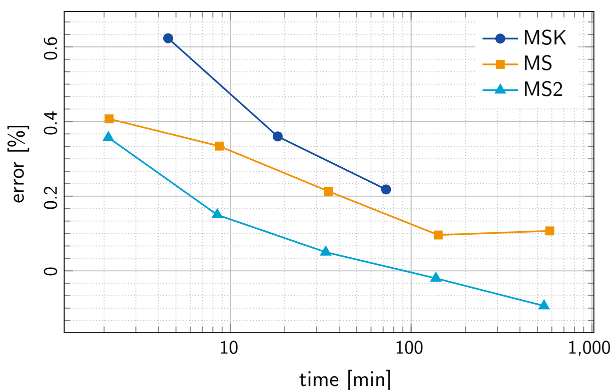

Fig. 3 Performance of three multistart variants on 41 open SteinLib instances. Errors relative to best known solutions by August 1, 2014. Times are geometric means of all runs

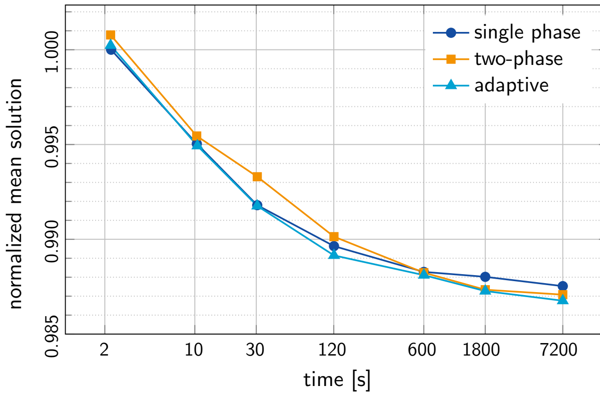

Fig. 4 Relative performance (on a set of 30 representative instances) of time-bounded versions of our multistart algorithm: single-phase multistart (MS), two-phase multistart (MS2), and adaptive (with threshold 2048). Mean solution quality is relative to the single-phase version with a 2-s time bound

5.1.5 Time-bounded algorithms

We now consider the time-bounded versions of our multistart algorithms, as described in Sect. 4.4. We ran three variants: *single-phase* always runs MS; *two-phase* always runs MS2; and *adaptive* runs MS if the expected number of iterations is at most 2048 (and MS2 otherwise). In this experiment, we consider the 30 representative instances tested in Fig. 4, each with seven time bounds, from 2 s to 2 h.

Figure 4 shows how the (geometric) mean solution improves with time for all three variants. For clarity, solution values are given relative to the single-phase algorithm with a 2-s time bound (for which the actual mean solution value was 36,195.018). As expected, all three algorithms scale well with time. Moreover, the adaptive algorithm is effective in picking a good strategy for all regimes (single-phase for short runs and two-phase for longer ones). Within 5 h, the adaptive algorithm achieves a mean solution value of 35,715.321, an improvement of more than 1.3% over the baseline.

5.2 Branch-and-bound

We now consider the effectiveness of our branch-and-bound procedure as a standalone exact algorithm. Unlike our heuristics, it is not robust. There are some graphs (such as large vlsi or fst instances) for which it will not produce a good solution in reasonable time, let alone prove its optimality. On large instances with small duality gaps, exact algorithms based on dual ascent are generally not competitive with those using linear programming.

We thus focus on small instances with large duality gaps. Series i080, i160, and i320 have been solved to optimality [31,38], as have 95 of 100 instances from i640 [12,40] (all but i640-311, i640-312, i640-313, i640-314, and i640-315). We also consider all solved instances from the bip and cc series. Our method could solve every such instance in less than 2 h; most took fractions of a second.

Table 8 Performance of our branch-and-bound algorithm on select hard instances, in comparison with the exact algorithm by Polzin and Vahdati Daneshmand [12,40]

SERIES	COUNT	AVERAGE TIME			MEAN TIME		
		OURS	[12,40]	RATIO	OURS	[12,40]	RATIO
i160	100	0.042	0.384	9.2	0.009	0.066	7.4
i320	100	13.195	211.834	16.1	0.069	0.428	6.2
i640	95	176.852	1363.153	7.7	0.468	2.492	5.3
cc	8	214.706	5384.671	25.1	1.102	32.404	29.4
bip	2	225.316	572.083	2.5	224.113	571.383	2.5

For perspective, Table 8 compares our exact algorithm against a state-of-the-art approach (as of August 1, 2014) for such instances, due to Polzin and Vahdati Daneshmand [12,40]. The table has all instances they solved from series i160, i320, i640, cc, and bip. For each series, we show the number of instances tested, our average time in seconds, the average time of their method (divided by 6.12 to match our machine), and the ratio between them. The remaining columns use geometric means instead of averages.

Our method is quite competitive for these instances. Running times are comparable for bip instances and we are faster for other graph classes. The relative difference is higher when we consider averages rather than mean times, indicating that our advantage is greater on harder instances (which have a more pronounced effect on the average). This confirms that the engineering effort outlined in Sect. 3 does pay off.

We stress, however, that Table 8 contains only a very small (and not particularly representative) subset of all instances tested. Because Polzin and Vahdati Daneshmand use linear programming and advanced reduction techniques, there are several classes of instances (such as vlsi) that they can easily solve but we cannot. This is true for other algorithms as well [25].

For completeness, we also compare our algorithm with the branch-and-ascent implementation proposed by Poggi de Aragão et al. [38] (described in further detail in [56]). This method (which we call B&A) can be seen as a precursor of our approach. We limit our tests to i320, a series that was first solved to optimality by B&A itself [38]. This series has 100 random graphs with 320 vertices and adversarial (incidence) costs. The instances are divided in 20 groups of 5; instances in each group are generated with the same number of terminals and edges, but different random seeds.

For conciseness, Table 9 only reports results for the first instance in each group (see Table 21 in the Appendix for full results). To solve all 100 instances, B&A needs 767.8 s on average; the (geometric) mean time is 770 ms. Our new algorithm takes an average time 11.7 s and a mean time of 55 ms. (Both codes were evaluated using the same compiler and machine as in Tables 5 and 6.) The numbers of branch-and-bound nodes visited is comparable: B&A visits 27,913 nodes on average, while we visit 19,976 (the geometric means are 24.4 and 51.3, respectively). This indicates that most of our advantage comes from processing each node much faster; the bounds themselves are not stronger. For easy instances, in particular, we actually tend to visit more nodes, partly because of different accounting: B&A may run dual ascent multiple

Table 9 Branch-and-bound comparison with the B&A algorithm from Poggi de Aragão et al. [38] and Werneck [56] on i320 instances

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-001	8	480	5.5	0.4	14.50	1	1	1.00
i320-011	8	1845	23.5	8.3	2.83	1	15	0.07
i320-021	8	51,040	120.6	25.0	4.83	1	1	1.00
i320-031	8	640	16.8	4.1	4.05	1	14	0.07
i320-041	8	10,208	191.0	24.8	7.71	3	19	0.16
i320-101	17	480	6.5	1.2	5.53	1	5	0.20
i320-111	17	1845	394.1	19.2	20.50	23	56	0.41
i320-121	17	51,040	196.9	42.3	4.66	1	1	1.00
i320-131	17	640	67.4	4.5	14.97	5	19	0.26
i320-141	17	10,208	3217.2	89.8	35.83	39	66	0.59
i320-201	34	480	33.8	5.0	6.78	9	34	0.26
i320-211	34	1845	18,747.1	134.1	139.80	811	256	3.17
i320-221	34	51,040	554.6	56.0	9.90	1	12	0.08
i320-231	34	640	577.4	11.9	48.59	97	57	1.70
i320-241	34	10,208	21,533.3	187.1	115.11	313	88	3.56
i320-301	80	480	130.7	26.2	4.99	21	152	0.14
i320-311	80	1845	3,500,392.7	79,056.1	44.28	122,261	140,452	0.87
i320-321	80	51,040	7,426.4	710.4	10.45	53	127	0.42
i320-331	80	640	17,902.8	550.3	32.53	2215	3623	0.61
i320-341	80	10,208	2,426,929.1	42,775.3	56.74	58,581	31,266	1.87

Ratios greater than 1.0 favor our method

times on the same node, as long as each run reduces the number of edges by at least 1% (in such cases, we create a new child node). For harder instances, we often visit fewer nodes.

Finally, we note that our branch-and-bound algorithm could prove that 35,535 is the optimal solution for i640-313, a formerly open incidence instance. On a machine about 28% faster than the one we used for our main experiments, it took 15.16 days and visited 7.31 billion branch-and-bound nodes. For this particular run, we gave the algorithm 35,536 (one unit above the optimum) as the initial upper bound and used strong branching.

5.3 Recent developments

So far, we have considered only instances and results available before August 1, 2014; these roughly correspond to the state-of-the-art before the final phase of the 11th DIMACS Implementation Challenge.[4] This section considers subsequent developments, which were motivated by the challenge itself.

Algorithms First, Polzin and Vahdati Daneshmand reran their exact algorithm on a newer machine with different sets of parameters and an up-to-date version of CPLEX and made the results available on the challenge web page [42]. Although the results are mostly consistent with their previous publications [12,40], the additional tuning has made the algorithm more competitive for some "hard" graph classes—notably those in Table 8. For series i160, i320, i640, and bip, their (scaled) average running times in seconds are now 0.08, 65.2, 169.3, and 389.4. These improvements (of at least a factor of three) bring their algorithm closer to ours. For cc, however, scaled average times are only slightly better (4890 instead of 5385), which makes our method still more than 20 times faster. In addition, their algorithm can solve all vienna instances to optimality within an hour or so.

Another contribution to the challenge was the SCIP-Jack algorithm by Gamrath, Koch, Maher, Rehfeldt, and Shinano [22] (see also [23]), which can be seen as an updated version of the MIP-based work by Koch and Martin [31] made massively parallel using the SCIP [1] framework. On instances with small duality gaps, it is essentially dominated by the work of Polzin and Vahdati Daneshmand, which use more modern reduction techniques. For small, hard instances, however, running on hundreds of cores allowed them to improve the best known bounds for several instances. Other notable contributions were the algorithms of Althaus and Blumenstock [2] and Biazzo, Braunstein, and Zecchnia [7]. The latter works particularly well on small adversarial instances, notably the hard class. For VLSI (and other) instances with a small number of terminals, Hougardy, Silvanus, and Vygen [25] (see also [26]) presented an exact algorithm based on dynamic programming that can significantly outperform any other approach. None of these algorithms, however, is particularly robust. Although they slightly outperform our method on their core classes, they are much worse on

[4] Detailed competition results can be found at http://dimacs11.zib.de/contest/results/results.html.

others (often being more than 10% off or failing to produce a solution in reasonable time).

The most robust contribution to the challenge (besides ours) was the algorithm of Fischetti, Leitner, Ljubic, Luipersbeck, Monaci, Resch, Salvagnin, and Sinnl [19] (see also [20]), which combines several techniques. Besides including the local search implementations proposed by Uchoa and Werneck [52], their algorithm relies heavily on mathematical programming techniques, such as using local branching to search large neighborhoods or to fix almost-feasible solutions produced by a set covering heuristic. Depending on some characteristics of the input instance, their algorithm decides which techniques and strategies to use. In particular, on instances with uniform or near-uniform costs, a new lightweight integer programming formulation based on node separators can replace the usual directed cut formulation [58]. This works extremely well on some hard instances, especially those from series hc and bip. On more general instances, however, they do not outperform Polzin and Vahdati Daneshmand [12,40]. One advantage of their approach relative to ours is that it can also handle other variants of the Steiner problem (such as prize-collecting).

With these new developments, some bounds have been improved by other submissions to the 11th DIMACS Challenge [28]. Together, Gamrath et al. [22] and Fischetti et al. [19] managed to improve the best known bounds by August 1, 2014 for 25 of the 41 open SteinLib instances. Compared against these improved bounds, our algorithm is still strictly better on 9 and matches a further 14.

The challenge included a contest comparing long (2-h) runs of the algorithms above on the set of 30 representative instances we considered in Figs. 2 and 4. The best performers were the algorithm of Fischetti et al. and our algorithm (our entry was the time-bounded single-phase multistart approach); Biazzo et al. [7] had the best performance for some hard instances, but was less consistent overall. Despite being very different in nature, the top two contenders had very similar performance. If one simply considers the mean solution quality, Fischetti et al. had a slight advantage; in a points-based system (in the style of Formula 1, with algorithms assigned points for each instance based on their relative rank), the slight advantage was ours. The algorithms were also evaluated on how fast they found good solutions (rather than just what the final solution was), using the primal integral method [5]. Once again, our algorithm was worse in terms of the mean value but better in the points-based system. Since the points-based system requires consistency, this confirms the robustness of our approach. Moreover, we note that the time limit (2 h) was rather large. Figure 4 shows that the algorithm can produce a reasonable solution in a couple of seconds even for large instances, as it does not have to solve a large linear program. Finally, although the adaptive version of our time-bounded algorithm outperforms our (single-phase) entry in the contest (see Fig. 4), it was only developed later, using the results of contest as motivation. We expect other algorithms to have improved since the challenge as well.

New instances We now discuss instances made available after August 1, 2014.

To highlight the strengths of their algorithm, Fischetti et al. [19] introduced the ccn series, created from the existing cc series [57] by setting all edge costs to one. Five of

Table 10 Results on open instances from the ccn series for MS (single-phase multistart) with 256 iterations and for MS2 (two-phase multistart) with 65,536 iterations

INSTANCE					MS (256)			MS2 (65,536)		
NAME	$\|V\|$	$\|E\|$	$\|T\|$	PREV	SOL	ERROR	TIME	SOL	ERROR	TIME
cc10-2n	1024	5120	135	180	180	0.00	7.9	179	−0.56	2127.9
cc11-2n	2048	11,263	244	326	326	0.00	21.7	324	−0.61	5607.5
cc12-2n	4096	24,574	473	620	616	−0.65	60.2	613	−1.13	15,348.7
cc3-10n	1000	13,500	50	75	75	0.00	10.4	75	0.00	2962.7
cc3-11n	1331	19,965	61	92	92	0.00	16.8	92	0.00	4765.4
cc3-12n	1728	28,512	74	111	111	0.00	27.1	111	0.00	7172.2
cc7-3n	2187	15,308	222	289	290	0.35	25.7	288	−0.35	6849.7
cc9-2n	512	2304	64	99	101	2.02	3.0	98	−1.01	780.4

For each algorithm, we show the solution found (SOL), average percent error (ERROR) relative to the solutions found by Fischetti et al. [19] (PREV), and the running time in seconds (TIME)

Table 11 Multistart algorithm on EFST series: average running time in seconds and average percent error relative to the best known solutions, with the number of iterations varying from 1 to 256

SERIES	TIME [s]					ERROR [%]				
	1	4	16	64	256	1	4	16	64	256
tspefst	0.91	4.69	21.65	92.83	383.03	0.57	0.46	0.35	0.27	0.18
r25kefst	0.94	6.47	29.90	131.22	552.69	0.23	0.19	0.15	0.12	0.10
r50kefst	2.42	17.71	92.54	410.18	1742.29	0.24	0.20	0.16	0.13	0.11
r100kefst	7.10	64.63	315.93	1395.32	5909.05	0.23	0.19	0.16	0.14	0.12

its instances can be solved to optimality by both Fischetti et al. and our branch-and-bound (although we are two orders of magnitude slower for cc6-3n, the hardest among those). For the remaining (open) instances, Table 10 shows the performance of two versions of our heuristic (MS with 256 iterations and MS2 with 65,536 iterations). Even the faster version is quite competitive: it finds a better bound on the largest instance and is never worse by more than two edges. With longer runs, our two-phase algorithm improves the best solutions found by Fischetti et al. [19] in five cases.

Table 11 reports the performance of our multistart algorithm on the efst class [29], which consists of graph instances generated by the GeoSteiner package [29,55] from Euclidean inputs (each graph is the union of a small set of *full Steiner trees*). The class is divided into four series, depending on whether the terminals originate from TSPLib instances (tspefst) or from random points on the plane (r25kefst, r50kefst, r100kefst). For each series and number of iterations, the table shows the average running time (in seconds) as well as the percent error relative to the best known solution obtained by Juhl et al. [29], available at http://dimacs11.zib.de/instances/bounds-efst-20150324.txt.

Although we cannot match the quality of their specialized algorithm (which is not based on a graph problem), a single iteration of our algorithm is enough to get within half a percent of the best known solution on average. With more iterations, the error rate decreases to below 0.2%. These graphs are quite sparse, with average degrees ranging from 2.4 (for random instances) to about 4.3 (for very regular tsp instances). In contrast, the ratio of nonterminals to terminals, which is roughly 0.58 for random instances (such as rk*efst), reaches more than 240 for some tspefst instances (such as fl1400efst and p654efst). These high-ratio instances are the hardest for our method. That said, even the highest error we observed with 256 iterations (1.37% on instance u1432efst, with 98 nonterminals for each terminal) was still fairly small.

Finally, we consider a class of synthetic instances created to illustrate duality gaps for certain graph classes. Series gap-csd, gap-g, and gap-smc consist of very small instances, every one of which we can solve to optimality in less than 10 ms (by either preprocessing or branch-and-bound). For series gap-s, a single iteration of our multistart algorithm can find the optimal solutions (as proven by [19]) of all five instances.

6 Conclusion

We presented a new heuristic approach for the Steiner problem in graphs, based on fast local searches, multistart with an evolutionary component, and fast combinatorial algorithms for finding lower bounds. Although the algorithm could be further improved, notably by incorporating more elaborate preprocessing techniques, it is already quite robust. For short runs, it is competitive with any previous approach on a wide variety of instance classes. Moreover, it is scalable: when given more time, it improved the best published solutions for several hard instances from the literature. Overall, our results show that primal heuristics can be an important component of robust solvers for the Steiner problem in graphs.

Acknowledgements We thank two anonymous referees for their suggestions on how to improve this work, and all 11th DIMACS Implementation Challenge participants for their comments. We are most grateful to David S. Johnson, not only for providing essential feedback to an early version of this paper, but also (among numerous other accomplishments) for creating and nurturing the DIMACS Challenge series and championing the field of Algorithm Engineering. We dedicate this paper to his memory.

A Appendix

A.1 Additional results

This section presents detailed results and data omitted from the main text due to space constraints.

Table 12 presents more detailed information about each of the series tested in this work. Tables 13 and 14 have results for our basic MS algorithms aggregated by series. Tables 15 and 16 are similar, but refer to the GMS algorithm.

Table 17 reports the best solution found considering all 13 runs (5 for MS, 5 for MS2, and 3 for MSK) used to build Table 7. It includes the sizes of all instances tested in this experiment. Tables 18 and 19 report results for individual MS2 runs separately, for a more detailed view. Table 20 shows detailed results for our pure branch-and-bound algorithm on some hard instances. These runs do not use reduction-based preprocessing, which is generally ineffective for these instances (Table 21).

Table 12 Instance sizes

CLASS	SERIES	VERTICES			EDGES			TERMINALS		
		MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
euclidean	p4e	100	136	200	4950	10,386	19,900	5	26	100
	p6e	100	127	200	180	231	370	5	28	100
	x	52	259	666	1326	74,808	221,445	16	72	174
fst	cph14	16	2471	15,473	23	5969	38,928	10	143	1000
	es0010	12	17	24	11	19	32	10	10	10
	es0020	27	39	57	26	47	83	20	20	20
	es0030	43	69	118	44	92	188	30	30	30
	es0040	55	90	121	55	121	180	40	40	40
	es0050	83	116	143	96	160	211	50	50	50
	es0060	109	140	188	133	192	280	60	60	60
	es0070	142	167	209	181	232	314	70	70	70
	es0080	147	189	236	180	259	343	80	80	80
	es0090	175	217	284	221	303	430	90	90	90
	es0100	188	241	339	233	336	522	100	100	100
	es0250	542	624	713	719	880	1053	250	250	250
	es0500	1172	1303	1477	1627	1871	2204	500	500	500
	es1000	2532	2747	2984	3615	4023	4484	1000	1000	1000
	es10000	27,019	27,019	27,019	39,407	39,407	39,407	10,000	10,000	10,000
	tsp	89	1756	17,127	104	2247	27,352	48	1130	11,849
hard	bip	550	1690	3300	3982	9013	18,073	50	190	300
	cc	64	1165	4096	192	9797	28,512	8	112	473
	hc	64	1161	4096	192	6418	24,576	32	581	2048
	sp	6	738	3997	9	1974	10,278	3	408	2284
incidence	i080	80	80	80	120	884	3160	6	12	20
	i160	160	160	160	240	3327	12,720	7	21	40
	i320	320	320	320	480	12,843	51,040	8	35	80
	i640	640	640	640	960	50,350	204,480	9	61	160
r	1r	1250	1250	1250	2319	2341	2352	6	32	60
	2r	2000	2000	2000	5725	5766	5800	9	51	98
random	b	50	75	100	63	122	200	9	23	50
	c	500	500	500	625	4156	12,500	5	95	250
	d	1000	1000	1000	1250	8312	25,000	5	186	500

Table 12 continued

CLASS	SERIES	VERTICES			EDGES			TERMINALS		
		MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
	e	2500	2500	2500	3125	20,781	62,500	5	461	1250
	mc	97	261	400	760	4208	11,175	45	126	213
	p4z	100	100	100	4950	4950	4950	5	18	50
	p6z	100	127	200	180	231	370	5	28	100
vienna	gadv	7565	27,157	71,184	11,521	42,634	113,616	86	1918	6107
	gori	42,481	109,841	235,686	52,552	157,257	366,093	88	1979	6313
	iadv	160	12,796	43,690	237	19,045	66,461	23	1157	4138
	isim	1991	30,129	89,596	3176	49,017	148,583	38	1393	4991
vlsi	alue	940	8061	34,479	1474	12,901	55,494	16	241	2344
	alut	387	10,707	36,711	626	19,741	68,117	34	161	879
	diw	212	3423	11,821	381	6434	22,516	10	20	50
	dmxa	169	1110	3983	280	1959	7108	10	15	23
	gap	179	1496	10,393	293	2579	18,043	10	22	104
	lin	53	8587	38,418	80	15,956	71,657	4	31	172
	msm	90	1548	5181	135	2682	8893	10	17	89
	taq	122	2150	6836	194	3648	11,715	10	39	136
wrp	wrp3	84	939	3168	149	1833	6220	11	51	99
	wrp4	110	571	1898	188	1131	3616	11	44	76

Minimum, rounded average, and maximum numbers of vertices, edges, and terminals for each series

Table 13 Multistart algorithm: average percent error relative to best known solution

CLASS	SERIES	1	4	16	64	256
euclidean	p4e	0.354079	OPT	OPT	OPT	OPT
	p6e	0.017396	0.016565	OPT	OPT	OPT
	x	0.111395	0.002994	0.001633	OPT	OPT
fst	cph14	1.448691	0.742808	0.403645	0.173233	0.079506
	es0010	0.004975	OPT	OPT	OPT	OPT
	es0020	0.258192	0.001403	0.001403	OPT	OPT
	es0030	0.538273	0.095129	OPT	0.007655	OPT
	es0040	0.376601	0.064947	0.012512	OPT	OPT
	es0050	0.564110	0.173306	0.008416	OPT	OPT
	es0060	0.593695	0.117862	0.003685	OPT	OPT
	es0070	0.522834	0.061763	0.011418	OPT	OPT
	es0080	0.544692	0.149718	0.021355	OPT	OPT

Table 13 continued

CLASS	SERIES	1	4	16	64	256
	es0090	0.557080	0.131118	0.014802	0.003146	0.003146
	es0100	0.541163	0.135542	0.015631	0.004006	OPT
	es0250	0.535736	0.248894	0.081227	0.012991	0.002332
	es0500	0.636633	0.273873	0.119918	0.037057	0.016075
	es1000	0.658757	0.326103	0.160499	0.078633	0.028934
	es10000	0.716213	0.440336	0.281747	0.179433	0.134925
	tsp	0.423169	0.201457	0.061733	0.026143	0.012092
hard	bip	7.327444	4.536031	2.825376	1.861770	1.131055
	cc	5.141251	2.964033	1.766326	0.897492	0.497459
	hc	3.101166	2.045988	1.101811	0.704332	0.248315
	sp	1.319787	0.882730	0.449628	0.182346	0.103500
incidence	i080	2.526005	0.554687	0.186856	0.008158	OPT
	i160	2.853316	1.070750	0.388198	0.119041	0.016433
	i320	2.532908	1.093114	0.405381	0.205291	0.065077
	i640	2.520669	1.149782	0.514386	0.232517	0.088959
r	1r	3.479879	0.824871	0.082394	0.100265	OPT
	2r	5.104557	2.430135	0.807973	0.164885	0.112902
random	b	0.387073	OPT	OPT	OPT	OPT
	c	1.581749	0.156989	OPT	OPT	OPT
	d	1.156902	0.402404	0.077101	0.038551	0.038551
	e	1.920698	0.180831	0.065410	0.038196	0.012771
	mc	3.273730	1.474736	0.415901	OPT	OPT
	p4z	OPT	OPT	OPT	OPT	OPT
	p6z	0.032752	OPT	OPT	OPT	OPT
vienna	gadv	0.296004	0.144022	0.067019	0.026995	−0.006757
	gori	0.315021	0.147975	0.078782	0.040818	0.004134
	iadv	0.010265	0.004938	0.002139	0.000946	0.000236
	isim	0.011645	0.005279	0.002684	0.001108	0.000320
vlsi	alue	0.661669	0.280747	0.150350	0.068250	0.027503
	alut	1.306957	0.617830	0.153181	0.056028	0.039895
	diw	0.554114	0.109545	0.037860	0.003217	OPT
	dmxa	0.662836	0.157417	0.026072	0.014047	0.014047
	gap	0.534932	0.071154	OPT	OPT	OPT
	lin	1.031740	0.382177	0.091563	0.042338	0.014215
	msm	0.858975	0.311446	0.110709	0.019593	OPT
	taq	0.730216	0.201021	0.069302	0.019074	0.012070
wrp	wrp3	0.000340	0.000114	0.000024	0.000007	0.000002
	wrp4	0.000654	0.000145	0.000029	OPT	OPT

Table 14 Multistart algorithm: average running times in seconds

CLASS	SERIES	1	4	16	64	256
euclidean	p4e	0.006	0.014	0.047	0.169	0.633
	p6e	0.001	0.003	0.010	0.038	0.141
	x	0.016	0.041	0.127	0.496	1.999
fst	cph14	0.042	0.260	1.250	5.302	21.960
	es0010	0.000	0.001	0.002	0.005	0.020
	es0020	0.000	0.001	0.004	0.016	0.065
	es0030	0.001	0.003	0.010	0.040	0.169
	es0040	0.001	0.003	0.014	0.053	0.220
	es0050	0.002	0.005	0.019	0.081	0.323
	es0060	0.002	0.005	0.023	0.099	0.392
	es0070	0.002	0.007	0.029	0.123	0.493
	es0080	0.002	0.008	0.033	0.138	0.560
	es0090	0.003	0.010	0.041	0.177	0.710
	es0100	0.003	0.011	0.046	0.193	0.780
	es0250	0.008	0.037	0.166	0.692	2.773
	es0500	0.017	0.099	0.460	1.964	7.646
	es1000	0.039	0.264	1.266	5.313	21.422
	es10000	0.654	5.823	27.689	121.841	503.952
	tsp	0.019	0.106	0.544	2.354	9.425
hard	bip	0.047	0.274	1.543	6.268	24.913
	cc	0.059	0.429	2.082	8.449	33.158
	hc	0.046	0.356	2.448	10.316	41.847
	sp	0.012	0.103	0.471	1.941	8.095
incidence	i080	0.004	0.012	0.047	0.191	0.781
	i160	0.016	0.049	0.198	0.805	3.314
	i320	0.084	0.264	1.006	4.175	16.854
	i640	0.481	1.454	5.775	23.842	97.034
r	1r	0.008	0.026	0.105	0.427	1.601
	2r	0.016	0.057	0.228	0.949	3.794
random	b	0.001	0.002	0.004	0.016	0.059
	c	0.006	0.014	0.049	0.193	0.760
	d	0.011	0.030	0.113	0.456	1.782
	e	0.030	0.092	0.370	1.522	6.115
	mc	0.005	0.014	0.058	0.243	0.961
	p4z	0.002	0.003	0.009	0.036	0.123
	p6z	0.001	0.004	0.014	0.049	0.180
vienna	gadv	0.679	6.095	29.585	126.941	535.205
	gori	1.604	11.786	63.403	272.846	1135.290
	iadv	0.245	1.810	9.290	40.353	168.097
	isim	0.489	3.212	15.921	68.797	285.147

Table 14 continued

CLASS	SERIES	1	4	16	64	256
vlsi	alue	0.070	0.430	2.046	8.696	35.130
	alut	0.096	0.600	2.748	12.462	49.884
	diw	0.026	0.095	0.389	1.658	6.484
	dmxa	0.008	0.026	0.096	0.394	1.565
	gap	0.011	0.047	0.205	0.826	3.198
	lin	0.097	0.508	2.425	10.265	42.312
	msm	0.010	0.037	0.148	0.645	2.543
	taq	0.017	0.074	0.319	1.372	5.332
wrp	wrp3	0.018	0.097	0.436	1.826	7.397
	wrp4	0.012	0.062	0.268	1.128	4.443

Table 15 Guarded Multistart algorithm: average percent error relative to best known solution

CLASS	SERIES	8	32	128
euclidean	p4e	OPT	OPT	OPT
	p6e	OPT	OPT	OPT
	x	OPT	OPT	OPT
fst	cph14	0.477741	0.256166	0.134948
	es0010	OPT	OPT	OPT
	es0020	OPT	OPT	OPT
	es0030	OPT	OPT	OPT
	es0040	0.000223	OPT	OPT
	es0050	0.015014	OPT	OPT
	es0060	0.024934	OPT	OPT
	es0070	0.015242	OPT	OPT
	es0080	0.021082	0.002182	OPT
	es0090	0.034372	0.011561	0.003146
	es0100	0.023469	0.003237	OPT
	es0250	0.118876	0.033715	0.007312
	es0500	0.194932	0.069319	0.020367
	es1000	0.228739	0.118464	0.045216
	es10000	0.374229	0.218046	0.151464
	tsp	0.096273	0.032074	0.016116
hard	bip	2.268220	1.850187	1.409930
	cc	1.993846	1.113872	0.644426
	hc	1.436478	0.864165	0.373133
	sp	0.557966	0.310501	0.188937
incidence	i080	OPT	OPT	OPT
	i160	0.030159	0.003275	OPT
	i320	0.075458	0.040413	0.027477
	i640	0.155963	0.076967	0.048300

Table 15 continued

CLASS	SERIES	8	32	128
r	1r	0.079756	OPT	OPT
	2r	0.611559	0.235590	0.112902
random	b	OPT	OPT	OPT
	c	OPT	OPT	OPT
	d	0.077101	OPT	OPT
	e	0.107210	0.076466	0.024327
	mc	0.181159	OPT	OPT
	p4z	OPT	OPT	OPT
	p6z	OPT	OPT	OPT
vienna	gadv	0.095218	0.036673	0.005620
	gori	0.102350	0.052840	0.020755
	iadv	0.003422	0.001217	0.000501
	isim	0.003866	0.001696	0.000707
vlsi	alue	0.169160	0.135619	0.029104
	alut	0.133225	0.081263	0.052188
	diw	0.011798	OPT	OPT
	dmxa	0.014047	OPT	OPT
	gap	OPT	OPT	OPT
	lin	0.113884	0.051941	0.021476
	msm	0.034357	0.015696	OPT
	taq	0.063997	0.019074	0.007004
wrp	wrp3	0.000039	0.000012	0.000003
	wrp4	0.000027	0.000003	OPT

Table 16 Guarded Multistart algorithm: average CPU times in seconds (wall times are lower)

CLASS	SERIES	8	32	128
euclidean	p4e	0.009	0.009	0.009
	p6e	0.004	0.005	0.004
	x	0.045	0.045	0.045
fst	cph14	1.139	5.056	20.791
	es0010	0.001	0.001	0.002
	es0020	0.002	0.002	0.002
	es0030	0.006	0.005	0.004
	es0040	0.007	0.013	0.012
	es0050	0.016	0.036	0.051
	es0060	0.019	0.043	0.066
	es0070	0.025	0.056	0.059
	es0080	0.031	0.089	0.212
	es0090	0.037	0.138	0.339

Table 16 continued

CLASS	SERIES	8	32	128
	es0100	0.045	0.142	0.413
	es0250	0.162	0.694	2.838
	es0500	0.416	1.871	7.550
	es1000	1.165	4.980	20.540
	es10000	28.187	123.472	519.184
	tsp	0.469	2.158	8.984
hard	bip	1.372	5.914	24.463
	cc	1.959	8.080	32.929
	hc	2.223	10.007	41.234
	sp	0.384	1.859	7.855
incidence	i080	0.011	0.011	0.011
	i160	0.058	0.085	0.088
	i320	0.362	0.658	1.566
	i640	3.618	9.715	15.934
r	1r	0.056	0.147	0.407
	2r	0.177	0.651	2.505
random	b	0.002	0.002	0.002
	c	0.014	0.014	0.015
	d	0.050	0.073	0.092
	e	0.192	0.577	2.483
	mc	0.031	0.072	0.085
	p4z	0.003	0.003	0.003
	p6z	0.004	0.004	0.003
vienna	gadv	29.845	130.705	531.566
	gori	64.433	284.922	1167.976
	iadv	8.910	39.788	166.381
	isim	15.395	69.712	285.812
vlsi	alue	2.045	8.263	35.097
	alut	3.357	12.570	49.949
	diw	0.362	1.457	5.302
	dmxa	0.074	0.180	0.524
	gap	0.159	0.566	2.104
	lin	2.590	10.327	43.031
	msm	0.126	0.409	1.347
	taq	0.275	1.150	4.626
wrp	wrp3	0.403	1.758	7.074
	wrp4	0.259	1.085	4.303

Table 17 Best results found by long runs

NAME	$\|V\|$	$\|E\|$	$\|T\|$	TIME [h]	KNOWN	OURS	ERROR
bip42p	1200	3982	200	12.74	**24,657**	24,764	0.434
bip42u	1200	3982	200	9.48	**236**	**236**	0.000
bip52p	2200	7997	200	28.05	**24,535**	24,628	0.379
bip52u	2200	7997	200	19.80	**234**	235	0.427
bip62p	1200	10,002	200	25.14	22,870	**22,843**	−0.118
bip62u	1200	10,002	200	22.11	**220**	**220**	0.000
bipa2p	3300	18,073	300	67.19	**35,379**	35,413	0.096
bipa2u	3300	18,073	300	54.78	341	**340**	−0.293
cc10-2p	1024	5120	135	18.31	35,379	**35,297**	−0.232
cc10-2u	1024	5120	135	14.03	**342**	**342**	0.000
cc11-2p	2048	11,263	244	53.67	63,826	**63,491**	−0.525
cc11-2u	2048	11,263	244	38.16	614	**612**	−0.326
cc12-2p	4096	24,574	473	172.24	**121,106**	121,710	0.499
cc12-2u	4096	24,574	473	122.83	1179	**1172**	−0.594
cc3-10p	1000	13,500	50	35.86	12,860	**12,772**	−0.684
cc3-10u	1000	13,500	50	22.69	**125**	**125**	0.000
cc3-11p	1331	19,965	61	58.18	15,609	**15,582**	−0.173
cc3-11u	1331	19,965	61	36.15	**153**	**153**	0.000
cc3-12p	1728	28,512	74	86.89	18,838	**18,826**	−0.064
cc3-12u	1728	28,512	74	57.04	186	**185**	−0.538
cc6-3p	729	4368	76	12.61	20,456	**20,330**	−0.616
cc6-3u	729	4368	76	8.82	**197**	**197**	0.000
cc7-3p	2187	15,308	222	69.14	57,088	**56,799**	−0.506
cc7-3u	2187	15,308	222	48.47	552	**549**	−0.543
cc9-2p	512	2304	64	6.57	17,296	**17,232**	−0.370
cc9-2u	512	2304	64	4.68	**167**	**167**	0.000
hc10p	1024	5120	512	20.53	60,494	**59,797**	−1.152
hc10u	1024	5120	512	16.59	581	**575**	−1.033
hc11p	2048	11,264	1024	58.74	119,779	**119,492**	−0.240
hc11u	2048	11,264	1024	51.44	**1154**	1156	0.173
hc12p	4096	24,576	2048	169.89	**236,949**	237,033	0.035
hc12u	4096	24,576	2048	164.49	2275	**2264**	−0.484
hc9p	512	2304	256	7.61	30,258	**30,243**	−0.050
hc9u	512	2304	256	5.79	**292**	**292**	0.000
i640-311	640	4135	160	13.63	**35,766**	**35,766**	0.000
i640-312	640	4135	160	13.53	**35,771**	35,794	0.064
i640-313	640	4135	160	13.70	**35,535**	**35,535**	0.000
i640-314	640	4135	160	13.69	**35,538**	**35,538**	0.000
i640-315	640	4135	160	13.86	**35,741**	**35,741**	0.000

Table 17 continued

NAME	$\|V\|$	$\|E\|$	$\|T\|$	TIME [h]	KNOWN	OURS	ERROR
wrp3-55	1645	3186	55	14.04	**5,500,888**	**5,500,888**	0.000
wrp3-83	3168	6220	83	35.60	**8,300,906**	**8,300,906**	0.000

The time (in hours) is the *sum* of the 13 executions of variants of our algorithm from Table 7. We also report the best previously known solution (as of August 1, 2014) and the percent error. Negative errors favor our method

The best solution for each instance is marked in bold

Table 18 Solutions found by MS2 (two-phase unguarded multistart algorithm) for open SteinLib instances, when varying the total number of iterations

NAME	KNOWN	1024	4096	16,384	65,536	262,144
bip42p	**24,657**	24,888	24,818	24,811	24,811	24,811
bip42u	**236**	237	237	**236**	237	237
bip52p	**24,535**	24,775	24,729	24,771	24,701	24,628
bip52u	**234**	237	235	235	235	235
bip62p	22,870	22,924	22,870	**22,843**	**22,843**	**22,843**
bip62u	**220**	221	221	**220**	**220**	**220**
bipa2p	**35,379**	35,616	35,555	35,516	35,523	35,413
bipa2u	341	342	**340**	**340**	**340**	**340**
cc10-2p	35,379	35,436	35,353	35,353	35,353	**35,297**
cc10-2u	**342**	343	343	343	**342**	343
cc11-2p	63,826	64,056	63,760	63,508	**63,491**	63,578
cc11-2u	**614**	618	616	**614**	615	**614**
cc12-2p	**121,106**	122,873	122,340	121,960	121,901	121,710
cc12-2u	1179	1183	1180	**1177**	1178	**1177**
cc3-10p	12,860	12,865	12,789	12,775	12,778	**12,772**
cc3-10u	**125**	**125**	**125**	**125**	**125**	**125**
cc3-11p	15,609	15,657	15,584	15,584	15,584	**15,582**
cc3-11u	**153**	**153**	**153**	**153**	**153**	**153**
cc3-12p	18,838	18,842	18,906	18,839	18,828	**18,826**
cc3-12u	186	**185**	**185**	186	**185**	**185**
cc6-3p	20,456	20,500	20,454	20,460	**20,330**	**20,330**
cc6-3u	**197**	198	198	198	**197**	**197**
cc7-3p	57,088	57,303	57,334	57,242	57,074	**56,799**
cc7-3u	552	555	550	552	551	**549**
cc9-2p	17,296	**17,293**	17,300	**17,293**	**17,293**	**17,293**
cc9-2u	**167**	168	168	168	**167**	**167**
hc10p	60,494	60,294	59,973	**59,797**	60,186	59,836
hc10u	581	582	578	576	576	**575**
hc11p	119,779	120,038	119,776	119,743	119,691	**119,653**

Table 18 continued

NAME	KNOWN	1024	4096	16,384	65,536	262,144
hc11u	**1154**	1161	1159	1157	1157	1156
hc12p	**236,949**	238,188	237,965	237,575	237,441	237,156
hc12u	2275	2305	2297	2284	2274	**2264**
hc9p	30,258	30,275	30,261	30,261	**30,243**	**30,243**
hc9u	**292**	**292**	**292**	**292**	**292**	**292**
i640-311	**35,766**	35,854	35,813	**35,766**	**35,766**	35,798
i640-312	**35,771**	35,908	35,863	35,830	35,819	35,819
i640-313	**35,535**	35,616	35,579	**35,535**	35,543	**35,535**
i640-314	**35,538**	35,656	35,588	35,551	35,550	35,550
i640-315	**35,741**	35,841	35,832	35,792	**35,741**	**35,741**
wrp3-55	**5,500,888**	**5,500,888**	**5,500,888**	**5,500,888**	**5,500,888**	**5,500,888**
wrp3-83	**8,300,906**	**8,300,906**	**8,300,906**	**8,300,906**	**8,300,906**	**8,300,906**

The best solution for each instance is marked in bold

Table 19 Running times in seconds of MS2 (two-phase unguarded multistart algorithm) on open SteinLib instances, when varying the total number of iterations

NAME	1024	4096	16,384	65,536	262,144
bip42p	61	242	933	3790	15,539
bip42u	40	163	652	2696	11,558
bip52p	133	521	1951	7879	30,967
bip52u	80	325	1348	5551	22,679
bip62p	132	512	1943	8011	31,593
bip62u	96	384	1615	6692	27,724
bipa2p	329	1305	4850	18,904	73,150
bipa2u	222	901	3708	15,554	63,098
cc10-2p	91	353	1417	5653	22,274
cc10-2u	61	245	980	4135	16,906
cc11-2p	258	1032	3938	16,094	60,872
cc11-2u	165	667	2633	10,644	43,882
cc12-2p	804	3114	12,174	48,835	181,427
cc12-2u	491	1905	7940	33,221	128,552
cc3-10p	171	684	2792	11,216	46,535
cc3-10u	101	415	1679	6920	28,483
cc3-11p	284	1167	4659	17,260	74,962
cc3-11u	170	655	2623	11,150	42,409
cc3-12p	441	1787	7403	27,695	109,187
cc3-12u	243	1074	4431	18,191	70,120
cc6-3p	61	233	976	4111	16,172
cc6-3u	39	162	658	2728	10,771

Table 19 continued

NAME	1024	4096	16,384	65,536	262,144
cc7-3p	331	1374	5230	21,256	80,143
cc7-3u	211	779	3317	13,814	54,852
cc9-2p	31	122	501	2057	8387
cc9-2u	22	86	347	1411	5819
hc10p	106	438	1589	7188	25,400
hc10u	86	336	1370	5425	20,835
hc11p	329	1239	4687	19,055	72,619
hc11u	258	1039	3972	16,590	64,509
hc12p	992	3743	13,878	54,043	199,814
hc12u	774	2888	12,249	49,925	184,367
hc9p	37	161	622	2459	10,008
hc9u	27	112	449	1851	7432
i640-311	68	275	1106	4522	17,426
i640-312	69	275	1105	4252	17,295
i640-313	64	266	1100	4398	17,922
i640-314	66	269	1118	4421	17,738
i640-315	69	281	1071	4400	18,006
wrp3-55	62	253	1005	4102	16,685
wrp3-83	149	618	2429	9668	39,522

Table 20 Branch-and-bound results on select hard instances: dimensions, solution, branch-and-bound nodes, and running time in seconds

INSTANCE	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	OPT	BB	TIME [s]
bipe2p	550	5013	50	5616	158,071	202.060
bipe2u	550	5013	50	54	473,584	248.572
cc3-4p	64	288	8	2338	339	0.040
cc3-4u	64	288	8	23	273	0.018
cc3-5p	125	750	13	3661	20,660	4.399
cc3-5u	125	750	13	36	39,892	3.669
cc5-3p	243	1215	27	7299	2,102,429	1196.743
cc5-3u	243	1215	27	71	1,904,887	512.745
cc6-2p	64	192	12	3271	213	0.025
cc6-2u	64	192	12	32	167	0.012
hc6p	64	192	32	4003	3036	0.250
hc6u	64	192	32	39	3215	0.194
hc7p	128	448	64	7905	1,979,435	366.055
hc7u	128	448	64	77	6,471,809	768.549

Table 21 Comparison between our algorithm and the branch-and-ascent (B&A) algorithm by Poggi de Aragão et al. [38] on i320 instances

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-001	8	480	5.5	0.4	14.50	1	1	1.00
i320-002	8	480	4.9	0.6	8.46	1	2	0.50
i320-003	8	480	6.2	1.6	3.88	1	7	0.14
i320-004	8	480	8.9	0.8	10.76	1	2	0.50
i320-005	8	480	23.5	0.8	29.50	1	2	0.50
i320-011	8	1845	23.5	8.3	2.83	1	15	0.07
i320-012	8	1845	32.4	2.8	11.57	5	4	1.25
i320-013	8	1845	170.7	9.1	18.82	7	12	0.58
i320-014	8	1845	64.9	30.3	2.14	5	21	0.24
i320-015	8	1845	52.4	11.3	4.64	1	10	0.10
i320-021	8	51,040	120.6	25.0	4.83	1	1	1.00
i320-022	8	51,040	210.5	24.6	8.54	1	1	1.00
i320-023	8	51,040	130.3	24.7	5.27	1	1	1.00
i320-024	8	51,040	123.4	24.0	5.14	1	1	1.00
i320-025	8	51,040	138.4	25.8	5.36	1	1	1.00
i320-031	8	640	16.8	4.1	4.05	1	14	0.07
i320-032	8	640	11.5	2.1	5.49	1	8	0.12
i320-033	8	640	18.7	1.4	13.42	1	2	0.50
i320-034	8	640	8.9	1.0	9.32	1	2	0.50
i320-035	8	640	10.7	1.0	10.51	1	8	0.12
i320-041	8	10,208	191.0	24.8	7.71	3	19	0.16
i320-042	8	10,208	69.0	13.0	5.32	1	1	1.00
i320-043	8	10,208	108.1	30.2	3.58	1	14	0.07

Table 21 continued

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-044	8	10,208	115.9	34.1	3.40	1	1	1.00
i320-045	8	10,208	157.6	32.7	4.83	1	1	1.00
i320-101	17	480	6.5	1.2	5.53	1	5	0.20
i320-102	17	480	8.1	2.9	2.82	1	23	0.04
i320-103	17	480	20.0	8.3	2.41	1	18	0.06
i320-104	17	480	12.4	9.4	1.31	1	29	0.03
i320-105	17	480	55.0	8.0	6.88	9	47	0.19
i320-111	17	1845	394.1	19.2	20.50	23	56	0.41
i320-112	17	1845	1179.0	50.1	23.55	65	139	0.47
i320-113	17	1845	646.2	38.9	16.62	41	104	0.39
i320-114	17	1845	1231.3	16.5	74.44	47	50	0.94
i320-115	17	1845	213.4	39.0	5.48	7	49	0.14
i320-121	17	51,040	196.9	42.3	4.66	1	1	1.00
i320-122	17	51,040	206.0	44.3	4.65	1	1	1.00
i320-123	17	51,040	397.3	44.1	9.01	1	1	1.00
i320-124	17	51,040	226.0	42.4	5.33	1	1	1.00
i320-125	17	51,040	391.0	44.6	8.76	1	1	1.00
i320-131	17	640	67.4	4.5	14.97	5	19	0.26
i320-132	17	640	9.2	1.5	6.36	1	3	0.33
i320-133	17	640	12.3	3.6	3.44	1	8	0.12
i320-134	17	640	35.7	6.3	5.64	1	23	0.04
i320-135	17	640	58.6	10.1	5.81	13	50	0.26
i320-141	17	10,208	3217.2	89.8	35.83	39	66	0.59

Table 21 continued

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-142	17	10,208	8001.4	63.2	126.63	75	16	4.69
i320-143	17	10,208	1121.9	46.1	24.36	5	10	0.50
i320-144	17	10,208	158.1	13.9	11.34	1	4	0.25
i320-145	17	10,208	1501.4	76.3	19.67	19	51	0.37
i320-201	34	480	33.8	5.0	6.78	9	34	0.26
i320-202	34	480	82.2	11.0	7.47	5	36	0.14
i320-203	34	480	69.4	19.8	3.50	7	114	0.06
i320-204	34	480	60.7	1.9	31.79	9	6	1.50
i320-205	34	480	190.3	5.4	34.98	69	56	1.23
i320-211	34	1845	18,747.1	134.1	139.80	811	256	3.17
i320-212	34	1845	8657.1	179.6	48.20	361	361	1.00
i320-213	34	1845	6125.8	859.6	7.13	227	2676	0.08
i320-214	34	1845	11,654.6	408.3	28.54	457	1186	0.39
i320-215	34	1845	45,388.2	1677.7	27.05	2141	4107	0.52
i320-221	34	51,040	554.6	56.0	9.90	1	12	0.08
i320-222	34	51,040	587.9	84.0	7.00	1	5	0.20
i320-223	34	51,040	641.0	154.7	4.14	1	13	0.08
i320-224	34	51,040	565.2	148.7	3.80	1	5	0.20
i320-225	34	51,040	662.9	78.1	8.49	1	10	0.10
i320-231	34	640	577.4	11.9	48.59	97	57	1.70
i320-232	34	640	531.9	20.7	25.74	65	89	0.73
i320-233	34	640	38.3	5.3	7.17	1	12	0.08
i320-234	34	640	119.4	36.6	3.26	9	292	0.03

Table 21 continued

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-235	34	640	86.5	20.3	4.27	11	110	0.10
i320-241	34	10,208	21,533.3	187.1	115.11	313	88	3.56
i320-242	34	10,208	48,601.4	693.6	70.07	989	785	1.26
i320-243	34	10,208	22,988.1	309.7	74.22	311	268	1.16
i320-244	34	10,208	40,837.8	465.7	87.70	675	394	1.71
i320-245	34	10,208	25,674.2	195.6	131.29	331	123	2.69
i320-301	80	480	130.7	26.2	4.99	21	152	0.14
i320-302	80	480	81.3	17.8	4.58	21	111	0.19
i320-303	80	480	251.9	43.2	5.83	31	143	0.22
i320-304	80	480	675.8	20.7	32.72	135	99	1.36
i320-305	80	480	275.9	25.7	10.74	51	132	0.39
i320-311	80	1845	3,500,392.7	79,056.1	44.28	122,261	140,452	0.87
i320-312	80	1845	10,461,540.5	198,391.0	52.73	396,313	343,403	1.15
i320-313	80	1845	5,863,080.6	168,805.8	34.73	257,545	338,432	0.76
i320-314	80	1845	18,567,507.1	319,225.4	58.16	716,091	577,182	1.24
i320-315	80	1845	29,198,177.0	283,656.2	102.94	1,078,597	497,745	2.17
i320-321	80	51,040	7426.4	710.4	10.45	53	127	0.42
i320-322	80	51,040	32,985.9	544.2	60.61	333	106	3.14
i320-323	80	51,040	14,571.0	306.3	47.57	105	34	3.09
i320-324	80	51,040	61,771.8	1088.4	56.76	687	247	2.78
i320-325	80	51,040	8921.1	352.8	25.28	79	36	2.19
i320-331	80	640	17,902.8	550.3	32.53	2215	3623	0.61
i320-332	80	640	983.4	36.0	27.33	113	122	0.93

Table 21 continued

INSTANCE			TOTAL TIME [ms]			BRANCH- AND- BOUND NODES		
NAME	$\|T\|$	$\|E\|$	[38]	OURS	RATIO	[38]	OURS	RATIO
i320-333	80	640	12,832.3	124.1	103.42	1753	727	2.41
i320-334	80	640	826.5	43.4	19.02	53	184	0.29
i320-335	80	640	4449.8	245.2	18.15	491	1300	0.38
i320-341	80	10,208	2,426,929.1	42,775.3	56.74	58,581	31,266	1.87
i320-342	80	10,208	124,999.9	1498.6	83.41	2803	875	3.20
i320-343	80	10,208	1,991,966.2	26,675.4	74.67	40,917	20,288	2.02
i320-344	80	10,208	2,528,871.1	15,281.1	165.49	64,497	11,288	5.71
i320-345	80	10,208	1,673,011.2	24,035.5	69.61	40,209	17,529	2.29

References

1. Achterberg, T.: SCIP: solving constraint integer programs. Math. Program. Comput. **1**(1), 1–41 (2009)
2. Althaus, E., Blumenstock, M.: Algorithms for the maximum weight connected subgraph and prize-collecting Steiner tree problems. In: Manuscript Presented at the 11th DIMACS/ICERM Implementation Challenge (2014)
3. Bastos, M.P., Ribeiro, C.C.: Reactive tabu search with path-relinking for the Steiner problem in graphs. In: Ribeiro, C.C., Hansen, P. (eds.) Essays and Surveys in Metaheuristics, pp. 39–58. Kluwer, Alphen aan den Rijn (2001)
4. Beasley, J.: OR-Library: distributing test problems by electronic mail. J. Oper. Res. Soc. **41**, 1069–1072 (1990). http://mscmga.ms.ic.ac.uk/info.html
5. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013)
6. Biazzo, I., Braunstein, A., Zecchina, R.: Performance of a cavity-method-based algorithm for the prize-collecting Steiner tree problem on graphs. Phys. Rev. E **86** (2012). http://arxiv.org/abs/1309.0346
7. Biazzo, I., Muntoni, A., Braunstein, A., Zecchina, R.: On the performance of a cavity method based algorithm for the prize-collecting Steiner tree problem on graphs. In: Presentation at the 11th DIMACS/ICERM Implementation Challenge (2014)
8. Byrka, J., Grandoni, F., Rothvoß, T., Sanità, L.: Steiner tree approximation via iterative randomized rounding. J. ACM **60**(1), 6:1–6:33 (2013)
9. Cheng, X., Du, D.Z.: Steiner Trees in Industry. Springer, Berlin (2002)
10. Chlebík, M., Chlebíková, J.: Approximation hardness of the Steiner tree problem on graphs. In: Proceedings of 8th Scandinavian Workshop on Algorithm Theory (SWAT), *LNCS*, vol. 2368, pp. 95–99. Springer (2002)
11. Chopra, S., Gorres, E.R., Rao, M.R.: Solving the Steiner tree problem on a graph using branch and cut. ORSA J. Comput. **4**, 320–335 (1992)
12. Daneshmand, S.V.: Algorithmic approaches to the Steiner problem in networks. Ph.D. thesis, Universität Mannheim. http://d-nb.info/970511787/34 (2003)
13. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
14. Dowsland, K.: Hill-climbing, simulated annealing and the Steiner problem in graphs. Eng. Optim. **17**, 91–107 (1991)
15. Duin, C.: Steiner's problem in graphs: approximation, reduction, variation. Ph.D. thesis, Institute for Actuarial Science and Economics, University of Amsterdam (1993)
16. Duin, C., Volgenant, A.: Reduction tests for the Steiner problem in graphs. Networks **19**, 549–567 (1989)
17. Duin, C., Voß, S.: Efficient path and vertex exchange in Steiner tree algorithms. Networks **29**, 89–105 (1997)
18. Duin, C., Voß, S.: The Pilot method: a strategy for heuristic repetition with application to the Steiner problem in graphs. Networks **34**, 181–191 (1999)
19. 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. In: Manuscript Presented at the 11th DIMACS/ICERM Implementation Challenge (2014)
20. Fischetti, M., Leitner, M., Ljubić, I., Luipersbeck, M., Monaci, M., Resch, M., Salvagnin, D., Sinnl, M.: Thinning out Steiner trees: a node-based model for uniform edge costs. Math. Program. Comput. **9**(2), 203–229 (2017)
21. Frey, C.: Heuristiken und genetisch algorithmen für modifizierte Steinerbaumprobleme. Ph.D. thesis (1997)
22. Gamrath, G., Koch, T., Maher, S.J., Rehfeldt, D., Shinano, Y.: SCIP-Jack: a solver for STP and variants with parallelization extensions. In: Manuscript Presented at the 11th DIMACS/ICERM Implementation Challenge (2014)
23. Gamrath, G., Koch, T., Maher, S.J., Rehfeldt, D., Shinano, Y.: SCIP-Jack–a solver for STP and variants with parallelization extensions. Math. Program. Comput. **9**(2), 231–296 (2017)
24. Goemans, M.X., Olver, N., Rothvoß, T., Zenklusen, R.: Matroids and integrality gaps for hypergraphic Steiner tree relaxations. In: ACM Symposium on Theory of Computing (STOC), pp. 1161–1176. ACM (2012)
25. Hougardy, S., Silvanus, J., Vygen, J.: Dijkstra meets Steiner: A fast exact goal-oriented Steiner tree algorithm. Technical Report abs/1406.0492, CoRR (2014)

26. Hougardy, S., Silvanus, J., Vygen, J.: Dijkstra meets Steiner: a fast exact goal-oriented Steiner tree algorithm. Math. Program. Comput. **9**(2), 135–202 (2017)
27. Huang, T., Young, E.F.Y.: ObSteiner: an exact algorithm for the construction of rectilinear Steiner minimum trees in the presence of complex rectilinear obstacles. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 882–893
28. Johnson, D.S., Koch, T., Werneck, R.F., Zachariasen, M.: 11th DIMACS Implementation Challenge in Collaboration with ICERM: Steiner Tree Problems. http://dimacs11.zib.de
29. Juhl, D., Warme, D.M., Winter, P., Zachariasen, M.: The GeoSteiner software package for computing Steiner trees in the plane: an updated computational study. In: Manuscript Presented at the 11th DIMACS/ICERM Implementation Challenge (2014)
30. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum, New York (1972)
31. Koch, T., Martin, A.: Solving Steiner tree problems in graphs to optimality. Networks **32**, 207–232 (1998)
32. Koch, T., Martin, A., Voß, S.: SteinLib: an updated library on Steiner tree problems in graphs. Tech. Rep. ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin. http://elib.zib.de/steinlib (2000)
33. Leitner, M., Ljubic, I., Luipersbeck, M., Prossegger, M., Resch, M.: New real-world instances for the Steiner tree problem in graphs. Tech. rep., ISOR, Uni Wien. http://homepage.univie.ac.at/ivana.ljubic/research/STP/realworld-stp-report-short.pdf (2014)
34. Mehlhorn, K.: A faster approximation algorithm for the Steiner problem in graphs. Inf. Process. Lett. **27**, 125–128 (1988)
35. Minoux, M.: Efficient greedy heuristics for Steiner tree problems using reoptimization and supermodularity. INFOR **28**, 221–233 (1990)
36. Osborne, L., Gillett, B.: A comparison of two simulated annealing algorithms applied to the directed Steiner problem on networks. ORSA J. Comp. **3**(3), 213–225 (1991)
37. Poggi de Aragão, M., Ribeiro, C.C., Uchoa, E., Werneck, R.F.: Hybrid local search for the Steiner problem in graphs. In: Ext. Abstracts of the 4th Metaheuristics International Conference, pp. 429–433. Porto (2001)
38. Poggi de Aragão, M., Uchoa, E., Werneck, R.F.: Dual heuristics on the exact solution of large Steiner problems. In: Proceedings of Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO), Elec. Notes in Disc. Math. vol. 7 (2001)
39. Poggi de Aragão, M., Werneck, R.F.: On the implementation of MST-based heuristics for the Steiner problem in graphs. In: Mount, D.M., Stein, C., (eds.) Proceedings of 4th Workshop on Algorithm Engineering and Experiments (ALENEX), LNCS, vol. 2409, pp. 1–15. Springer (2002)
40. Polzin, T.: Algorithms for the Steiner problem in networks. Ph.D. thesis, Universität des Saarlandes (2003)
41. Polzin, T., Vahdati Daneshmand, S.: Improved algorithms for the Steiner problem in networks. Discrete Appl. Math. **112**(1–3), 263–300 (2001)
42. Polzin, T., Vahdati Daneshmand, S.: The Steiner tree challenge: an updated study. In: Manuscript Contributed to the 11th DIMACS/ICERM Implementation Challenge. http://dimacs11.zib.de/downloads.html (2014)
43. Resende, M.G.C., Werneck, R.F.: A hybrid heuristic for the p-median problem. J. Heuristics **10**(1), 59–88 (2004)
44. Ribeiro, C.C., Souza, M.C.: Tabu search for the Steiner problem in graphs. Networks **36**, 138–146 (2000)
45. Ribeiro, C.C., Uchoa, E., Werneck, R.F.: A hybrid GRASP with perturbations for the Steiner problem in graphs. Informs J. Comput. **14**(3), 228–246 (2002)
46. Robins, G., Zelikovsky, A.: Tighter bounds for graph Steiner tree approximation. SIAM J. Discrete Math. **19**(1), 122–134 (2005)
47. Rosseti, I., Poggi de Aragão, M., Ribeiro, C.C., Uchoa, E., Werneck, R.F.: New benchmark instances for the Steiner problem in graphs. In: Ext. Abstracts of the 4th Metaheuristics International Conference, pp. 557–591. Porto (2001)
48. Spira, P.M., Pan, A.: On finding and updating spanning trees and shortest paths. SIAM J. Comput. **4**(3), 375–380 (1975)
49. Takahashi, H., Matsuyama, A.: An approximate solution for the Steiner problem in graphs. Math. Japonica **24**, 573–577 (1980)

50. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM, Philadelphia (1983)
51. Uchoa, E., Poggi de Aragão, M., Ribeiro, C.C.: Preprocessing Steiner problems from VLSI layout. Networks **40**(1), 38–50 (2002)
52. Uchoa, E., Werneck, R.F.: Fast local search for the Steiner problem in graphs. ACM J. Exper. Algorithms **17**(2), 2.2:1–2.2:22 (2012)
53. Verhoeven, M.G.A., Severens, M.E.M., Aarts, E.H.L.: Local search for Steiner trees in graphs. In: Rayward-Smith, V.J., Osman, I.H., Reeves, C.R. (eds.) Modern Heuristic Search Methods. Wiley, New York (1996)
54. Voß, S.: Steiner's problem in graphs: heuristic methods. Discrete Appl. Math. **40**(1), 45–72 (1992)
55. Warme, D., Winter, P., Zachariasen, M.: Exact algorithms for plane Steiner tree problems: a computational study. In: Du, D., Smith, J., Rubinstein, J. (eds.) Advances in Steiner Trees, Combinatorial Optimization, vol. 6. Kluwer, Alphen aan den Rijn (2000)
56. Werneck, R.F.: Steiner problem in graphs: primal, dual, and exact algorithms (In Portuguese). Master's thesis, Catholic University of Rio de Janeiro (2001)
57. Werneck, R.F., Rosseti, I., de Aragao, M.P., Ribeiro, C.C., Uchoa, E.: New benchmark instances for the Steiner problem in graphs. In: Resende, M.G.C., Souza, J. (eds.) Metaheuristics: Computer Decision-Making, pp. 601–614. Kluwer, Alphen aan den Rijn (2003)
58. Wong, R.: A dual ascent approach for Steiner tree problems on a directed graph. Math. Program. **28**, 271–287 (1984)
59. Zachariasen, M., Rohe, A.: Rectilinear group Steiner trees and applications in VLSI design. Tech. Rep. 00906, Institute for Discrete Mathematics, University of Bonn (2000)