A Robust and Scalable Algorithm for the Steiner Problem in Graphs*

Thomas Pajor[1], Eduardo Uchoa[2], and Renato F. Werneck[3]

[1] Microsoft Research, USA
tpajor@microsoft.com
[2] Universidade Federal Fluminense, Brazil
uchoa@producao.uff.br
[3] San Francisco, USA
rwerneck@cs.princeton.edu

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 quickly dealing with easier cases), 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.

* This work was partly done while R. Werneck was at Microsoft Research Silicon Valley.

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 application in many areas, including computational biology, networking, and circuit design [5]. Unfortunately, it is NP-hard not only to find an optimal solution [19], but also to approximate it within a factor of 96/95 [6]. The best known approximation ratio is 1.39 [4] (see [35] for a 1.55 approximation). Given its practical importance, there is a wealth of exact algorithms [8, 14, 16, 20, 22, 27, 29, 31, 34, 40] and heuristics [1, 3, 13, 14, 26, 33, 34, 41] 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 et al. [34]: it is a multistart algorithm with recombination (a genetic 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 [41], which are asympotically faster (in theory and practice) than previous approaches. Second, we propose a *cascaded combination* strategy, which 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 more 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 [8, 29].

To make our overall approach more robust, we include some basic 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 [46]. 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 [8, 29] or on dynamic programming [16] (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 Section 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 the best and other good solutions found so far. 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 our 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) Generate a solution S' by combining S with solutions in the pool.
   (c) Try to add S and then 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, Section 2.2 describes how fresh solutions are generated, Section 2.3 deals with the cascaded combination algorithm, and Section 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 minimum* (or *local optimum*). Uchoa and Werneck [41] 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 use the representation of a solution $S = (V_S, E_S)$ in terms of its set $V_S \setminus T$ of *Steiner vertices*. The minimum spanning tree (MST) of the subgraph of G induced by V_S (which we denote by $\mathrm{MST}(G[V_S])$) costs no more than S. In particular, if S is optimal, so is $\mathrm{MST}(G[V_S])$. Uchoa and Werneck use dynamic graph techniques to efficiently evaluate neighborhoods defined by the insertion or removal of a single Steiner vertex [24, 25, 37, 43]. Although such neighborhoods had been used in metaheuristics before [1, 33, 34], they were much slower: $O(|V|^2)$ 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 [10, 13, 42], 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 mostly take the local searches 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 [41].

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 main algorithm does not use it.

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 and local search) randomized, we follow Ribeiro et al. [34] and apply perturbations to the edge costs instead, preserving the running time guarantees of all algorithms.

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 [28] of the shortest-path heuristic (SPH) [38].

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. [34] 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 tend to 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 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.

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 (chosen between 1.0 and 1.2 in their case). This is reasonably effective, but has a potential drawback: because edges are independent, the perturbations applied to each incident edge 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 the 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 $[0, 1]$.

To achieve 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 (recall that we do not have a fast local search based on key-vertex insertion).

We stress that the algorithm already works reasonably well with the standard edge-based perturbation proposed by Ribeiro et al. [34]; although we observed some improvement with the vertex-based perturbation (and the non-uniform distribution), the effects were relatively minor.

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 [34], 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* G' 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. 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 merged *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 worse 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 (based on the symmetric difference between their edge sets). This technique has been shown to increase diversification for other problems [32].

2.5 Discussion

As Section 5 will show, our algorithm significantly outperforms the multistart approach by Ribeiro et al. [34]. Although there are many differences between the algorithms, the main reason for our good performance is that we leverage much faster implementations of the underlying local searches [41]. Not only can we run more iterations within the same time limit, but we can also do more in each iteration—with cascaded combinations.

Although relevant, other aspects of our algorithm have less effect than these two factors (fast local searches and cascaded combinations). The parameters we report (and there are many) are the ones we ended up using in the final version of our code. In most cases, however, the algorithm is not very sensitive to the exact value of these parameters, as long as they are reasonable. This includes the size of the pool, the type of perturbation, and the criteria for selecting elements from the pool.

3 Lower Bounds

We now turn our attention to finding lower bounds. We use an efficient implementation of an existing greedy algorithm (due to Wong [46]) associated with a powerful linear programming formulation. We first define the formulation and an abstract version of the algorithm, and 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 [46]. It takes a terminal $r \in T$ as the root. A set $W \subset V$ is a *Steiner cut* if W does not contain the root but contains at least one terminal. 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 basic building block of exact algorithm is a *dual ascent* routine proposed by Wong [46], which finds a greedy feasible solution to the dual formulation. The algorithm maintains a set of C of *active terminals*, initially consisting of $T \setminus \{r\}$. 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 vertex 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 check a 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 $O(\min\{|V||T|, |E|\})$ such iterations, bounding the total running time by $O(|E| \min\{|V||T|, |E|\})$ [11].

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 containing 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 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 will eventually become 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 saturated arcs, which will be part of the new root component of v (after augmentation).

Note that this three-pass approach requires a single graph search (in the first pass); the other 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 et al. [27] (see also [45]) found that using the number of incident arcs as the score works well in practice. Polzin and Vahdati [8, 29] 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, recomputing the new exact score of t can be expensive. Instead, we leverage the fact that we have the set X of the tails of all arcs saturated during the augmentation. We update the score assuming the vertices in the root component are the union of X 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 tighter 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 $\mathrm{vc}(c)$ be its number of vertices and let $\deg(c)$ be the sum of their in-degrees. We use $\deg(c) - (\mathrm{vc}(c) - 1)$ as the score. This is an upper bound on the number of incoming arcs on the component (the $\mathrm{vc}(c) - 1$ 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.

Last component. Typically, the initial cuts found by the dual ascent algorithm have very few arcs, while later ones are much denser. 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 [45].

3.3 Branch-and-Bound

To test the effectiveness of our dual ascent algorithm, we implemented a simple branch-and-bound procedure. We follow the basic principles of most previous

work [8,27,29,45,46], 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* [31], 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 edge if both corresponding arcs (directions) could be fixed by reduced cost. If we fix at least $|E|/5$ edges in a node, we create a single child node rather than branching.

4 Improving Robustness

While the algorithm we defined in Section 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 Section 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, Section 4.3 describes a two-level version of the multistart algorithm that achieves greater diversification on longer runs.

4.1 Guarded Multistart

Pure heuristics (such as the multistart algorithm described in Section 2) can be wasteful. Because our heuristic 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, on easy instances, we would like to stop sooner.

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 Section 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, there are cases in which the branch-and-bound algorithm can find better 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 in the same CPU time. To make this problem less pronounced, we use a simple heuristic to detect cases in which the branch-and-bound computation is obviously unhelpful: if its depth reaches 100, we stop it and proceed only with the multistart computation, saving CPU time.

Another potential drawback is nondeterminism: due to scheduling, different runs of our algorithm 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.

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 implemented 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* [12] 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 (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 restrict ourselves to common (and cheap) special cases: paths with up to two vertices (if both u and v have small degree) as well as some longer paths within a spanning tree of the graph. Both can be seen as restricted versions of existing algorithms [8, 29]. See Appendix A.5 for details.

4.3 Two-Phase Multistart

Even with all the measures we take to increase diversification, the 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 minimum.

Table 1. Classes of instances tested.

CLASS	SERIES	DESCRIPTION
euclidean	x p4e p6e	Euclidean costs [7, 20]
fst	es*fst tspfst cph14	reduced geometric, L1 costs [17, 18, 44]
hard	bip cc hc sp	synthetic hard instances [21, 36]
incidence	i080 i160 i320 i640	random graphs, incidence costs [11]
r	1r 2r	2D and 3D cross-grid graphs [15]
random	b c d e mc p4z p6z	graphs with random costs [2, 7, 20]
vienna	gori gadv isim iadv	road networks [22]
vlsi	alue alut dmxa diw gap lin msm taq	planar grid graphs with holes [20, 21]
wrp	wrp3 wrp4	group Steiner grid instances [47]

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 Section 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 MS to be adversely influenced by a particularly bad initial solution.

5 Experiments

We implemented all algorithms in C++ and compiled them using Visual Studio 2013 (optimizing for speed). We ran our experiments on a machine 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.cs.princeton.edu/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.

We evaluate all instances available by August 1, 2014 from the 11th DIMACS Implementation Challenge [18]. We group each original series into *classes*, as shown in Table 1 (augmented from [41]). More detailed information about the dimensions of the instances in each series can be found in Table 7, in the appendix. Most instances are available from the SteinLib [21], with two exceptions: cph14 (graphs obtained from rectilinear problems [17]) and vienna (road networks from telecommunication applications [22]). When evaluating

Table 2. Multistart Algorithm: Average running time in seconds and average percent error relative to the best known solutions, with number of iterations varying from 1 to 256. (Errors are multiplied by 10^3 for wrp.)

	TIME [S]					ERROR [%]				
CLASS	1	4	16	64	256	1	4	16	64	256
euclidean	0.005	0.011	0.036	0.135	0.520	0.155	0.009	0.000	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	0.000
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*

solution quality, we compare to the best results published by August 1, 2014 (listed at `http://dimacs11.cs.princeton.edu/downloads.html`).

5.1 Multistart

In our first experiment, we ran the default version of our multistart algorithm on all instances from the DIMACS Challenge [18]. Recall that this version is not guarded (no branch-and-bound), but 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. (Results aggregated by series can be found in Tables 8 and 9, in the appendix.) Because percent errors are very small for the wrp instances (a side effect of a reduction from the Group Steiner Tree problem), to improve readability we show the error for wrp multiplied by 1000 in this and other tables. The special entries are marked in *slanted font*.

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 7, 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 known.

Guarded Multistart. Recall that our *Guarded Multistart* (GMS) algorithm runs two parallel threads, the first with the standard multistart and the second with a branch-and-bound algorithm. The algorithm stops as soon as either the multistart algorithm completes its scheduled number of iterations or when the branch-and-bound procedure proves the solution is optimal. We tested GMS with 2, 8, 32,

Table 3. Guarded Multistart: Average CPU time in seconds and average percent error relative to the best known solutions. (Errors are multiplied by 10^3 for wrp.)

	TIME [s]				ERROR [%]			
CLASS	2	8	32	128	2	8	32	128
euclidean	0.009	0.010	0.011	0.010	0.000	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*

and 128 multistart iterations. Table 3 reports the error rates and average running times (see also Tables 10 and 11, 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 quickly prove the optimality of the incumbent. For random, incidence, and especially euclidean, GMS is faster than MS.

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 (for the same amount of 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.

Comparison. Table 4 compares Guarded Multistart (GMS) against the three state-of-the-art heuristics presented by Polzin and Vahdati [8, 29]: PRUNE, ASCENT&PRUNE, and SLACK-PRUNE. In particular, as reported by Polzin and Vahdati [8, 29], they dominate the multistart approach by Ribeiro et al. [34].

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 [8, 29], Table 4 considers the lin series separately from the remaining vlsi instances. Running times for their algorithms are scaled (divided by 6.12) to match our machine; Appendix A.2 explains this factor.

The table shows that the algorithms have very different profiles. Both PRUNE and ASCEND&PRUNE are quite fast, with running times comparable to GMS1,

Table 4. Comparison of various heuristics. For wrp instances, all errors are multiplied by 10^3 (for example, entry 0.6 actually means 0.0006%).

	PREVIOUS ALGORITHMS [8, 29]						GUARDED MULTISTART							
	PRUNE		ASCEND&PRUNE		SLACK-PRUNE		GMS1		GMS8		GMS32		GMS128	
GROUP	ERR.	TIME	ERR.	TIME	ERR.	TIME	ERR.	TIME	ERR.	TIME	ERR.	TIME	ERR.	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

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, 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 [8,29] 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. The fact that GMS is competitive even in the absence of very hard instances is encouraging. It shows 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.

Long Runs. To test the scalability of our algorithm, we consider the 41 SteinLib instances with 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 Section 2. MS2 is the two-phase version of MS, as described in Section 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 [26]); it is very expensive, but can find 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 262144 (2^{18}) iterations (1024 for MSK).

For each variant (and number of iterations), Table 5 shows results aggregated over all 41 instances: (geometric) mean time in seconds and average error with respect to the best published solutions.

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 16384 is cheaper and leads to better solutions. Unsurprisingly, MS and MS2 have comparable running times for the same number of iterations. As argued in Section 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 65536 to 262144. By starting from four independent sets of solutions, MS2 is less likely to be confined to a particularly bad region of the search space.

Table 5. 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
	16384	2094.8	0.213
	65536	8474.3	0.096
	262144	35056.4	0.107
MS2	1024	126.8	0.357
	4096	507.6	0.150
	16384	2022.3	0.050
	65536	8221.0	−0.020
	262144	32609.4	−0.094
MSK	64	271.6	0.623
	256	1096.3	0.360
	1024	4359.1	0.218

Considering all 13 runs from Table 5 (five runs each for MS and MS2, and three for MSK), there were only six cases (out of 41) for which we could not at least match the best bound published by August 1, 2014. (See Table 12, in the appendix.) In 21 cases, we found a strictly better solution. Most of these were found by MS2 (see Table 13, in the appendix); MSK was better only for cc11-2u and cc12-2u.

5.2 Branch-and-Bound

For completeness, 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 our algorithm will not produce a good solution in reasonable time, let alone prove its optimality. On large instances with small duality gaps, algorithms based on dual ascent are generally not competitive with exact algorithms that use linear programming.

We thus focus on small instances with large duality gaps. Series i080, i160, and i320 have been solved to optimality [20, 27], as have 95 of 100 instances from i640 [8, 29]. We also consider all solved instances from the bip and cc series. As shown in Tables 15, 16, 17, 18, and 19 (in the appendix), our method could solve every such instance in less than two hours; most took fractions of a second.

For perspective, Table 6 compares our exact algorithm against the state-of-the-art approach for such instances, due to Polzin and Vahdati [8, 29]. 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

Table 6. Performance of our branch-and-bound algorithm on select hard instances, in comparison with the exact algorithm by Polzin and Vahdati Daneshmand [8, 29].

		AVERAGE TIME			MEAN TIME		
SERIES	COUNT	OURS	[8, 29]	RATIO	OURS	[8, 29]	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

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 Section 3 does pay off.

We stress, however, that Table 6 contains only a very small (and not particularly representative) subset of all instances tested. Because Polzin and Vahdati 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 [16]. Even for some of the instances in Table 6, the algorithm by Polzin and Vahdati has become more competitive since its initial publication. They recently reran [30] their original algorithm on a newer machine with different sets of parameters and using an up-to-date version of CPLEX. 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.

Finally, we note that our branch-and-bound algorithm could prove that 35535 is the optimal solution for i640-313, a formerly open incidence instance. On a machine about 28% faster than the one we used for all other experiments, it took 15.16 days and visited 7.31 billion branch-and-bound nodes. For this particular run, we gave the algorithm 35536 (just above the optimum) as the initial upper bound and used strong branching (as described in Section A.4, in the appendix).

6 Conclusion

We presented a new heuristic approach for the Steiner problem in graphs, based on fast local searches, multistart with diversification, 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 (as of August 1, 2014)

for 21 number hard instances from the literature. We note that some bounds have also been improved by other submissions to the 11th DIMACS Challenge [18]. According to a summary published[4] on September 12, 15 of the 21 improved solutions we found remain the best known (and 13 of those have not been matched by other approaches). Overall, our results show that primal heuristics can be an important component of robust solvers for the Steiner problem in graphs.

[4] `http://dimacs11.cs.princeton.edu/instances/bounds20140912.txt`

References

1. M. P. Bastos and C. C. Ribeiro. Reactive tabu search with path-relinking for the Steiner problem in graphs. In C. C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pp. 39–58. Kluwer, 2001.
2. J. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41:1069–1072, 1990. `http://mscmga.ms.ic.ac.uk/info.html`.
3. I. Biazzo, A. Braunstein, and R. Zecchina. Performance of a cavity-method-based algorithm for the prize-collecting steiner tree problem on graphs. *Physical Review E*, 86, 2012.
4. J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An improved LP-based aproximation for Steiner tree. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC)*. ACM Press, 2010.
5. X. Cheng and D.-Z. Du. *Steiner Trees in Industry*. Springer, 2002.
6. M. Chlebík and J. Chlebíková. Approximation hardness of the Steiner tree problem on graphs. In *Proc. 8th Scandinavian Workshop on Algorithm Theory (SWAT)*, LNCS 2368, pp. 95–99. Springer, 2002.
7. S. Chopra, E. R. Gorres, and M. R. Rao. Solving the Steiner tree problem on a graph using branch and cut. *ORSA Journal on Computing*, 4:320–335, 1992.
8. S. V. Daneshmand. *Algorithmic Approaches to the Steiner Problem in Networks*. PhD thesis, Universität Mannheim, 2003.
9. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
10. K. Dowsland. Hill-climbing, simulated annealing and the Steiner problem in graphs. *Engineering Optimization*, 17:91–107, 1991.
11. C. Duin. *Steiner's Problem in Graphs: Aproximation, reduction, variation*. PhD thesis, Institute for Actuarial Science and Economics, University of Amsterdam, 1993.
12. C. Duin and A. Volgenant. Reduction tests for the Steiner problem in graphs. *Networks*, 19:549–567, 1989.
13. C. Duin and S. Voß. Efficient path and vertex exchange in Steiner tree algorithms. *Networks*, 29:89–105, 1997.
14. C. Duin and S. Voß. The Pilot method: A strategy for heuristic repetition with application to the Steiner problem in graphs. *Networks*, 34:181–191, 1999.
15. C. Frey. *Heuristiken und genetisch Algorithmen für modifizierte Steinerbaumprobleme*. PhD thesis, 1997.
16. S. Hougardy, J. Silvanus, and J. Vygen. Dijkstra meets steiner: A fast exact goal-oriented steiner tree algorithm. *CoRR*, abs/1406.0492, 2014.
17. T. Huang and E. F. Y. Young. Obsteiner: An exact algorithm for the construction of rectilinear steiner minimum trees in the presence of complex rectilinear obstacles. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6), 2013.
18. D. S. Johnson, T. Koch, R. F. Werneck, and M. Zachariasen. 11th DIMACS Implementation Challenge in collaboration with icerm: Steiner tree problems. `http://dimacs11.cs.princeton.edu`.
19. R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pp. 85–103. Plenum, 1972.
20. T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

21. T. Koch, A. Martin, and S. Voß. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2000.
22. M. Leitner, I. Ljubic, M. Luipersbeck, M. Prossegger, and M. Resch. New real-world instances for the steiner tree problem in graphs. Technical report, ISOR, Uni Wien, 2014.
23. K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.
24. M. Minoux. Efficient greedy heuristics for Steiner tree problems using reoptimization and supermodularity. *INFOR*, 28:221–233, 1990.
25. L. Osborne and B. Gillett. A comparison of two simulated annealing algorithms applied to the directed Steiner problem on networks. *ORSA J. Comp.*, 3(3), 1991.
26. M. Poggi de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. Hybrid local search for the Steiner problem in graphs. In *Ext. Abstracts of the 4th Metaheuristics International Conference*, pp. 429–433, Porto, Portugal, 2001.
27. M. Poggi de Aragão, E. Uchoa, and R. F. Werneck. Dual heuristics on the exact solution of large Steiner problems. In *Proc. Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO)*, Elec. Notes in Disc. Math. 7, 2001.
28. M. Poggi de Aragão and R. F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In D. M. Mount and C. Stein, editors, *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, LNCS 2409, pp. 1–15. Springer, 2002.
29. T. Polzin. *Algorithms for the Steiner Problem in Networks*. PhD thesis, Universität des Saarlandes, 2003.
30. T. Polzin and S. V. Daneshmand. The steiner tree challenge: An updated study. Unpublished manuscript at `http://dimacs11.cs.princeton.edu/downloads.html`.
31. T. Polzin and S. V. Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112(1–3):263–300, 2001.
32. M. G. C. Resende and R. F. Werneck. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10(1):59–88, 2004.
33. C. C. Ribeiro and M. C. Souza. Tabu search for the Steiner problem in graphs. *Networks*, 36:138–146, 2000.
34. C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS J. Computing*, 14(3):228–246, 2002.
35. G. Robins and A. Zelikovsky. Tighter bounds for graph Steiner tree approximation. *SIAM Journal on Discrete Mathematics*, 19(1):122–134, 2005.
36. I. Rosseti, M. Poggi de Aragão, C. C. Ribeiro, E. Uchoa, and R. F. Werneck. New benchmark instances for the Steiner problem in graphs. In *Ext. Abstracts of the 4th Metaheuristics International Conference*, pp. 557–591, Porto, Portugal, 2001.
37. P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Computing*, 4(3):375–380, 1975.
38. H. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Japonica*, 24:573–577, 1980.
39. R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
40. E. Uchoa, M. Poggi de Aragão, and C. C. Ribeiro. Preprocessing Steiner problems from VLSI layout. *Networks*, 40(1):38–50, 2002.
41. E. Uchoa and R. F. Werneck. Fast local search for the steiner problem in graphs. *ACM Journal of Experimental Algorithms*, 17(2):2.2:1–2.2:22, 2012.
42. M. G. A. Verhoeven, M. E. M. Severens, and E. H. L. Aarts. Local search for Steiner trees in graphs. In V. J. Rayward-Smith, I. H. Osman, and C. R. Reeves, editors, *Modern Heuristic Search Methods*. Wiley, 1996.

43. S. Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40(1):45–72, 1992.
44. D. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: a computational study. In D. Du, J. Smith, and J. Rubinstein, editors, *Advances in Steiner Trees*, Combinatorial Optmization 6. Kluwer, 2000.
45. R. F. Werneck. Steiner problem in graphs: Primal, dual, and exact algorithms. Master's thesis, Catholic University of Rio de Janeiro, 2001. In Portuguese.
46. R. Wong. A dual ascent approach for Steiner tree problems on a directed graph. *Mathematical Programming*, 28:271–287, 1984.
47. M. Zachariasen and A. Rohe. Rectilinear group Steiner trees and applications in VLSI design. Technical Report 00906, Institute for Discrete Mathematics, University of Bonn, 2000.

A Appendix

A.1 Contest Entry

The work presented in this article served as the basis in the contest associated with the 11th DIMACS Implementation Challenge (in Collaboration with ICERMS). In the contest, different programs are analyzed according to the quality of all incumbent solutions within a given time limit τ. The standard version of our multistart algorithm (as described in the main text) is parameterized by the number of iterations, rather than a time limit. Moreover, the size of the elite pool depends on the number of iterations, which must be known in advance. This section describes some small adaptations we made to our algorithm to account for the time limit τ in a meaningful way.

Our entry in the contest is based on the unguarded algorithm described in Section 2, modified to work in three phases.

The first phase runs a subset of the preprocessing routines described in Section 4.2, restricted to operations that only remove edges. The fact that no shortcuts are added avoids the need to implement edge-mapping routines.

The second phase runs the standard multistart iteration with a *single iteration*, which consists of the constructive algorithm (SPH) followed by local search. Unlike our standard algorithm, here we use actual edge weights, without perturbation. Both the constructive solution and the local optimum (if better) are reported as incumbent solutions.

The third phase is a standard run of the multistart algorithm described in Section 2 (starting from an empty elite pool). It sets the size of the elite pool to $\sqrt{\hat{M}/2}$, where $\hat{M}$ is an estimate on the number of iterations the algorithm can run within the time budget τ. Let τ_1 be the running time of the second phase (above). We set $\hat{M} = \tau/(2.5\tau_1)$, meaning that we estimate that a standard iteration of the algorithm will take about 2.5 times the first one (which does not combinations). This constant (2.5) was found empirically, and the algorithm is not too sensitive to it. We set the maximum number of iterations of the algorithm to $M = 65536$, but stop sooner if the time limit τ is reached. Only incumbent solutions found within the time limit are reported.

A.2 Scaling

Tables 4 and 6 compare the running times of our algorithm to those found by Polzin and Vahdati Daneshmand [8, 29]. As previously mentioned, we divide their published times by 6.12 for a fair comparison. This section explains how we arrived at this factor.

In the absence of more precise benchmarks, we use the scores available at `http://www.cpubenchmark.net/singleThread.html`. The Intel Xeon X5680 processors in our test machine have a score of 1484. The relevant experiments in [29] use a Sunfire 15000 with 900 MHz SPARC III+ CPUs. Although this machine is not listed in the page above, the authors also excuted one of their

experiments (the exact algorithm) on an AMD Athlon XP 1800+ running at 1.53 GHz, whose score is 477. To compute the relative speed of these two machines, we look at es10000 and fnl4461, the two hardest instances for which results on both machines are given in [29]. On the Sunfire, Table A.6 [29] states that their code takes 1398.9 s and 12967.0 s, respectively. On the AMD, the corresponding times (given in Table 5.3 [29]) are 758 and 6148. The mean ratio between these values is 1.967, which would give the Sunfire a score of $477/1.967 \approx 242.5$; our machine is thus $1484/242.5 \approx 6.12$ times faster according to this metric.

For reference, to prove the optimality of i640-313 we used an Intel Xeon E5-1620 running at 3.6 GHz. Its score at `http://www.cpubenchmark.net/singleThread.html` is 1911, which is about 28% higher than our main machine.

To compare our results with those in a recent manuscript by Vahdati and Daneshmand [30], we divide their running times by 1.269, the ratio between the 11th DIMACS Challenge Benchmark scores of our machine (388.913664) and theirs (306.508988).

A.3 Notes on Local Search

Uchoa and Werneck [41] state in passing that key-vertex removal "dominates" Steiner vertex removal, but this is not strictly true. One can construct local optima for key-vertex removal that can be improved by Steiner vertex removal, since the latter can make an original degree-two vertex (on a key-path) become a key vertex. In practice, however, we observed that such cases are extremely rare.

The original implementations evaluated by Uchoa and Werneck [41] use the C# programming language; we converted them to C++ for this work. Since C# is interpreted rather than compiled, our new implementation is often faster by a factor of 3 or more. Finally, we note that the running times reported in the appendix of [41] are for a single iteration (and not to reach the local optimum, as the article mistakenly states).

A.4 Strong Branching

For very long runs of our branch-and-bound algorithm, we found it worth it to use strong branching at higher levels of the search tree. We observed that the variance can be quite large, given the random choices of the dual ascent algorithm. To get a reliable measure of the effect of branching on a vertex v, we would have to perform several simulations.

Instead, we evaluate *sets* of vertices at a time, and infer from that the effect of individual vertices. The idea is simple. First, we pick c random sets with s nonterminals each (c and s are parameters of the algorithm). For each such set, we fix all vertices simultaneously to the same side (0 or 1), run dual ascent on the resulting graph, and note the corresponding dual bound. The final score of each vertex is the average dual bound over all sets that contain v. Higher scores are better.

In practice, we found that the dual bound tends to increase less when eliminating a vertex (side 0) than when making it a terminal (side 1). With that in

mind, we consider both cases separately. For the 0 side, we set $c = 1000$ and $s = 32 - d$, where d is the depth in the branch-and-bound tree; for the 1 side, we set $c = 2000$ and $s = 10$. Each vertex receives a separate scores σ_0 and σ_1 for each side (0 and 1), and use $\sigma = (\sigma_0^3 \cdot \sigma_1)^{1/4}$ as the final score.

We call this approach *scatter branching*. Since it is very expensive, we only use it at small depths in the tree (up to 14) and for very large instances (with billions of branch-and-bound nodes). For i640-313 (the only instance in this paper for which we used scatter branching), we saw speedups of about a factor of two. Further tuning may lead to better results.

A.5 Reduction

As anticipated in Section 4.2, we now describe our restricted implementation of the *Bottleneck Steiner Distance* test. Given an edge (u, v), our goal is to find a bottleneck path between u and v that does not use (u, v) itself. We use two quick heuristics to look for a subset of such paths; while this is correct (we preserve the optimal solution), we miss some opportunities for reduction, leading to a instance that may not be as small as possible.

The first heuristic is simple. If the combined degree of u and v is small (up to 20 in our implementation), 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 test we implemented is more global and may find more complicated paths. We first use a modified version of Dijkstra's algorithm [9] to build the Voronoi diagram associated with the terminals of G [23, 28, 41]. For each vertex v, this structure defines $vb(v)$ (the closest terminal to v), $vd(v)$ (the distance to that terminal), and $vp(v)$ (the parent edge on the path to $vb(v)$). By looking at boundary edges of this Voronoi diagram, we compute the MST of the distance network of G [23, 28, 41].

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 are neither 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. Then, we initialize a union-find data structure [39] 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) $dist(u, vb(u)) \leq cost(e_i)$; and (3) $dist(v, vb(v)) \leq cost(e_i)$.

This Voronoi-based test is very effective in random and dense Euclidean graphs, drastically reducing the graph size.

A.6 Additional Results

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

Table 7 presents more detailed information about each of the series tested in this work. Tables 8 and 9 have results for our basic MS algorithms aggregated by series. Tables 10 and 11 are similar, but refer to the GMS algorithm.

Table 12 reports the best solution found considering all 13 runs (10 for MS2 and 3 for MSK) used to build Table 5. It includes the sizes of all instances tested in this experiment. Tables 13 and 14 report results for all individual runs separately, for a more detailed view.

Tables 15, 16, 17, 18, and 19 show detailed results for our pure branch-and-bound algorithm. These runs do not use reduction-based preprocessing, which is generally ineffective for these instances.

Table 7. Instance sizes. Minimum, rounded average, and maximum numbers of vertices, edges, and terminals for each series.

		VERTICES			EDGES			TERMINALS		
CLASS	SERIES	MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
euclidean	p4e	100	136	200	4950	10386	19900	5	26	100
	p6e	100	127	200	180	231	370	5	28	100
	x	52	259	666	1326	74808	221445	16	72	174
fst	cph14	16	2471	15473	23	5969	38928	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	27019	27019	27019	39407	39407	39407	10000	10000	10000
	tsp	89	1756	17127	104	2247	27352	48	1130	11849
hard	bip	550	1690	3300	3982	9013	18073	50	190	300
	cc	64	1165	4096	192	9797	28512	8	112	473
	hc	64	1161	4096	192	6418	24576	32	581	2048
	sp	6	738	3997	9	1974	10278	3	408	2284
incidence	i080	80	80	80	120	884	3160	6	12	20
	i160	160	160	160	240	3327	12720	7	21	40
	i320	320	320	320	480	12843	51040	8	35	80
	i640	640	640	640	960	50350	204480	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	12500	5	95	250
	d	1000	1000	1000	1250	8312	25000	5	186	500
	e	2500	2500	2500	3125	20781	62500	5	461	1250
	mc	97	261	400	760	4208	11175	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	27157	71184	11521	42634	113616	86	1918	6107
	gori	42481	109841	235686	52552	157257	366093	88	1979	6313
	iadv	160	12796	43690	237	19045	66461	23	1157	4138
	isim	1991	30129	89596	3176	49017	148583	38	1393	4991
vlsi	alue	940	8061	34479	1474	12901	55494	16	241	2344
	alut	387	10707	36711	626	19741	68117	34	161	879
	diw	212	3423	11821	381	6434	22516	10	20	50
	dmxa	169	1110	3983	280	1959	7108	10	15	23
	gap	179	1496	10393	293	2579	18043	10	22	104
	lin	53	8587	38418	80	15956	71657	4	31	172
	msm	90	1548	5181	135	2682	8893	10	17	89
	taq	122	2150	6836	194	3648	11715	10	39	136
wrp	wrp3	84	939	3168	149	1833	6220	11	51	99
	wrp4	110	571	1898	188	1131	3616	11	44	76

Table 8. 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
	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 9. 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
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 10. 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
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 11. Guarded Multistart algorithm: Average CPU times in seconds (actual 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
	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 12. Best results found by long runs. The time (in hours) is the *sum* of the 13 executions of variants of our algorithm from Table 5. We also report the best previously known solution (as of August 1, 2014) and the percent error. Negative errors favor our method.

NAME	$\|V\|$	$\|E\|$	$\|T\|$	TIME [h]	KNOWN	OURS	ERROR
bip42p	1200	3982	200	12.74	**24657**	24764	0.434
bip42u	1200	3982	200	9.48	**236**	**236**	0.000
bip52p	2200	7997	200	28.05	**24535**	24628	0.379
bip52u	2200	7997	200	19.80	**234**	235	0.427
bip62p	1200	10002	200	25.14	22870	**22843**	−0.118
bip62u	1200	10002	200	22.11	**220**	**220**	0.000
bipa2p	3300	18073	300	67.19	**35379**	35413	0.096
bipa2u	3300	18073	300	54.78	341	**340**	−0.293
cc10-2p	1024	5120	135	18.31	35379	**35297**	−0.232
cc10-2u	1024	5120	135	14.03	**342**	**342**	0.000
cc11-2p	2048	11263	244	53.67	63826	**63491**	−0.525
cc11-2u	2048	11263	244	38.16	614	**612**	−0.326
cc12-2p	4096	24574	473	172.24	**121106**	121710	0.499
cc12-2u	4096	24574	473	122.83	1179	**1172**	−0.594
cc3-10p	1000	13500	50	35.86	12860	**12772**	−0.684
cc3-10u	1000	13500	50	22.69	**125**	**125**	0.000
cc3-11p	1331	19965	61	58.18	15609	**15582**	−0.173
cc3-11u	1331	19965	61	36.15	**153**	**153**	0.000
cc3-12p	1728	28512	74	86.89	18838	**18826**	−0.064
cc3-12u	1728	28512	74	57.04	186	**185**	−0.538
cc6-3p	729	4368	76	12.61	20456	**20330**	−0.616
cc6-3u	729	4368	76	8.82	**197**	**197**	0.000
cc7-3p	2187	15308	222	69.14	57088	**56799**	−0.506
cc7-3u	2187	15308	222	48.47	552	**549**	−0.543
cc9-2p	512	2304	64	6.57	17296	**17232**	−0.370
cc9-2u	512	2304	64	4.68	**167**	**167**	0.000
hc10p	1024	5120	512	20.53	60494	**59797**	−1.152
hc10u	1024	5120	512	16.59	581	**575**	−1.033
hc11p	2048	11264	1024	58.74	119779	**119492**	−0.240
hc11u	2048	11264	1024	51.44	**1154**	1156	0.173
hc12p	4096	24576	2048	169.89	**236949**	237033	0.035
hc12u	4096	24576	2048	164.49	2275	**2264**	−0.484
hc9p	512	2304	256	7.61	30258	**30243**	−0.050
hc9u	512	2304	256	5.79	**292**	**292**	0.000
i640-311	640	4135	160	13.63	**35766**	**35766**	0.000
i640-312	640	4135	160	13.53	**35771**	35794	0.064
i640-313	640	4135	160	13.70	**35535**	**35535**	0.000
i640-314	640	4135	160	13.69	**35538**	**35538**	0.000
i640-315	640	4135	160	13.86	**35741**	**35741**	0.000
wrp3-55	1645	3186	55	14.04	**5500888**	**5500888**	0.000
wrp3-83	3168	6220	83	35.60	**8300906**	**8300906**	0.000

Table 13. Solutions found by MS2 (two-phase unguarded multistart algorithm) for open SteinLib instances, when varying the total number of iterations.

NAME	KNOWN	1024	4096	16384	65536	262144
bip42p	**24657**	24888	24818	24811	24811	24811
bip42u	**236**	237	237	**236**	237	237
bip52p	**24535**	24775	24729	24771	24701	24628
bip52u	**234**	237	235	235	235	235
bip62p	22870	22924	22870	**22843**	**22843**	**22843**
bip62u	**220**	221	221	**220**	**220**	**220**
bipa2p	**35379**	35616	35555	35516	35523	35413
bipa2u	341	342	**340**	**340**	**340**	**340**
cc10-2p	35379	35436	35353	35353	35353	**35297**
cc10-2u	**342**	343	343	343	**342**	343
cc11-2p	63826	64056	63760	63508	**63491**	63578
cc11-2u	**614**	618	616	**614**	615	**614**
cc12-2p	**121106**	122873	122340	121960	121901	121710
cc12-2u	1179	1183	1180	**1177**	1178	**1177**
cc3-10p	12860	12865	12789	12775	12778	**12772**
cc3-10u	**125**	**125**	**125**	**125**	**125**	**125**
cc3-11p	15609	15657	15584	15584	15584	**15582**
cc3-11u	**153**	**153**	**153**	**153**	**153**	**153**
cc3-12p	18838	18842	18906	18839	18828	**18826**
cc3-12u	186	**185**	**185**	186	**185**	**185**
cc6-3p	20456	20500	20454	20460	**20330**	**20330**
cc6-3u	**197**	198	198	198	**197**	**197**
cc7-3p	57088	57303	57334	57242	57074	**56799**
cc7-3u	552	555	550	552	551	**549**
cc9-2p	17296	**17293**	17300	**17293**	**17293**	**17293**
cc9-2u	**167**	168	168	168	**167**	**167**
hc10p	60494	60294	59973	**59797**	60186	59836
hc10u	581	582	578	576	576	**575**
hc11p	119779	120038	119776	119743	119691	**119653**
hc11u	**1154**	1161	1159	1157	1157	1156
hc12p	**236949**	238188	237965	237575	237441	237156
hc12u	2275	2305	2297	2284	2274	**2264**
hc9p	30258	30275	30261	30261	**30243**	**30243**
hc9u	**292**	**292**	**292**	**292**	**292**	**292**
i640-311	**35766**	35854	35813	**35766**	**35766**	35798
i640-312	**35771**	35908	35863	35830	35819	35819
i640-313	**35535**	35616	35579	**35535**	35543	**35535**
i640-314	**35538**	35656	35588	35551	35550	35550
i640-315	**35741**	35841	35832	35792	**35741**	**35741**
wrp3-55	**5500888**	**5500888**	**5500888**	**5500888**	**5500888**	**5500888**
wrp3-83	**8300906**	**8300906**	**8300906**	**8300906**	**8300906**	**8300906**

Table 14. Running time in seconds of MS2 (two-phase unguarded multistart algorithm) on open SteinLib instances, when varying the total number of iterations.

NAME	1024	4096	16384	65536	262144
bip42p	61	242	933	3790	15539
bip42u	40	163	652	2696	11558
bip52p	133	521	1951	7879	30967
bip52u	80	325	1348	5551	22679
bip62p	132	512	1943	8011	31593
bip62u	96	384	1615	6692	27724
bipa2p	329	1305	4850	18904	73150
bipa2u	222	901	3708	15554	63098
cc10-2p	91	353	1417	5653	22274
cc10-2u	61	245	980	4135	16906
cc11-2p	258	1032	3938	16094	60872
cc11-2u	165	667	2633	10644	43882
cc12-2p	804	3114	12174	48835	181427
cc12-2u	491	1905	7940	33221	128552
cc3-10p	171	684	2792	11216	46535
cc3-10u	101	415	1679	6920	28483
cc3-11p	284	1167	4659	17260	74962
cc3-11u	170	655	2623	11150	42409
cc3-12p	441	1787	7403	27695	109187
cc3-12u	243	1074	4431	18191	70120
cc6-3p	61	233	976	4111	16172
cc6-3u	39	162	658	2728	10771
cc7-3p	331	1374	5230	21256	80143
cc7-3u	211	779	3317	13814	54852
cc9-2p	31	122	501	2057	8387
cc9-2u	22	86	347	1411	5819
hc10p	106	438	1589	7188	25400
hc10u	86	336	1370	5425	20835
hc11p	329	1239	4687	19055	72619
hc11u	258	1039	3972	16590	64509
hc12p	992	3743	13878	54043	199814
hc12u	774	2888	12249	49925	184367
hc9p	37	161	622	2459	10008
hc9u	27	112	449	1851	7432
i640-311	68	275	1106	4522	17426
i640-312	69	275	1105	4252	17295
i640-313	64	266	1100	4398	17922
i640-314	66	269	1118	4421	17738
i640-315	69	281	1071	4400	18006
wrp3-55	62	253	1005	4102	16685
wrp3-83	149	618	2429	9668	39522

Table 15. Branch-and-bound on series i080: dimensions, solution, branch-and-bound nodes, and running time in seconds.

INSTANCE	$\|V\|$	$\|E\|$	$\|T\|$	OPT	BB	TIME [s]
i080-001	80	120	6	1787	1	0.000
i080-002	80	120	6	1607	2	0.001
i080-003	80	120	6	1713	1	0.000
i080-004	80	120	6	1866	5	0.001
i080-005	80	120	6	1790	2	0.000
i080-011	80	350	6	1479	3	0.001
i080-012	80	350	6	1484	4	0.001
i080-013	80	350	6	1381	1	0.001
i080-014	80	350	6	1397	9	0.001
i080-015	80	350	6	1495	4	0.002
i080-021	80	3160	6	1175	1	0.003
i080-022	80	3160	6	1178	1	0.003
i080-023	80	3160	6	1174	2	0.003
i080-024	80	3160	6	1161	1	0.003
i080-025	80	3160	6	1162	1	0.003
i080-031	80	160	6	1570	1	0.001
i080-032	80	160	6	2088	5	0.001
i080-033	80	160	6	1794	2	0.001
i080-034	80	160	6	1688	6	0.001
i080-035	80	160	6	1862	2	0.001
i080-041	80	632	6	1276	4	0.001
i080-042	80	632	6	1287	1	0.001
i080-043	80	632	6	1295	2	0.001
i080-044	80	632	6	1366	9	0.002
i080-045	80	632	6	1310	3	0.001
i080-101	80	120	8	2608	4	0.001
i080-102	80	120	8	2403	1	0.001
i080-103	80	120	8	2603	4	0.001
i080-104	80	120	8	2486	4	0.001
i080-105	80	120	8	2203	1	0.001
i080-111	80	350	8	2051	8	0.002
i080-112	80	350	8	1885	10	0.002
i080-113	80	350	8	1884	3	0.001
i080-114	80	350	8	1895	1	0.001
i080-115	80	350	8	1868	2	0.001
i080-121	80	3160	8	1561	1	0.003
i080-122	80	3160	8	1561	1	0.003
i080-123	80	3160	8	1569	1	0.003
i080-124	80	3160	8	1555	1	0.003
i080-125	80	3160	8	1572	1	0.003
i080-131	80	160	8	2284	14	0.001
i080-132	80	160	8	2180	7	0.001
i080-133	80	160	8	2261	1	0.001
i080-134	80	160	8	2070	1	0.001
i080-135	80	160	8	2102	2	0.001
i080-141	80	632	8	1788	5	0.003
i080-142	80	632	8	1708	1	0.001
i080-143	80	632	8	1767	4	0.001
i080-144	80	632	8	1772	11	0.004
i080-145	80	632	8	1762	6	0.003
i080-201	80	120	16	4760	1	0.000
i080-202	80	120	16	4650	10	0.001
i080-203	80	120	16	4599	2	0.001
i080-204	80	120	16	4492	19	0.002
i080-205	80	120	16	4564	1	0.000
i080-211	80	350	16	3631	17	0.004
i080-212	80	350	16	3677	19	0.004
i080-213	80	350	16	3678	42	0.006
i080-214	80	350	16	3734	26	0.006
i080-215	80	350	16	3681	18	0.005
i080-221	80	3160	16	3158	1	0.006
i080-222	80	3160	16	3141	1	0.006
i080-223	80	3160	16	3156	1	0.005
i080-224	80	3160	16	3159	1	0.005
i080-225	80	3160	16	3150	1	0.006
i080-231	80	160	16	4354	9	0.002
i080-232	80	160	16	4199	15	0.002
i080-233	80	160	16	4118	5	0.001
i080-234	80	160	16	4274	6	0.001
i080-235	80	160	16	4487	9	0.002
i080-241	80	632	16	3538	95	0.022
i080-242	80	632	16	3458	8	0.004
i080-243	80	632	16	3474	17	0.006
i080-244	80	632	16	3466	20	0.007
i080-245	80	632	16	3467	49	0.012
i080-301	80	120	20	5519	1	0.001
i080-302	80	120	20	5944	4	0.001
i080-303	80	120	20	5777	8	0.001
i080-304	80	120	20	5586	1	0.000
i080-305	80	120	20	5932	14	0.001
i080-311	80	350	20	4554	16	0.004
i080-312	80	350	20	4534	66	0.011
i080-313	80	350	20	4509	9	0.003
i080-314	80	350	20	4515	38	0.006
i080-315	80	350	20	4459	9	0.002
i080-321	80	3160	20	3932	1	0.006
i080-322	80	3160	20	3937	1	0.006
i080-323	80	3160	20	3946	1	0.006
i080-324	80	3160	20	3932	1	0.006
i080-325	80	3160	20	3924	1	0.006
i080-331	80	160	20	5226	9	0.002
i080-332	80	160	20	5362	12	0.002
i080-333	80	160	20	5381	15	0.003
i080-334	80	160	20	5264	2	0.001
i080-335	80	160	20	4953	11	0.002
i080-341	80	632	20	4236	12	0.004
i080-342	80	632	20	4337	36	0.010
i080-343	80	632	20	4246	46	0.012
i080-344	80	632	20	4310	23	0.007
i080-345	80	632	20	4341	81	0.020

Table 16. Branch-and-bound on series i160: dimensions, solution, branch-and-bound nodes, and running time in seconds.

INSTANCE	$\|V\|$	$\|E\|$	$\|T\|$	OPT	BB	TIME [s]
i160-001	160	240	7	2490	5	0.002
i160-002	160	240	7	2158	1	0.000
i160-003	160	240	7	2297	1	0.001
i160-004	160	240	7	2370	1	0.001
i160-005	160	240	7	2495	2	0.001
i160-011	160	812	7	1677	5	0.003
i160-012	160	812	7	1750	5	0.003
i160-013	160	812	7	1661	1	0.001
i160-014	160	812	7	1778	12	0.004
i160-015	160	812	7	1768	20	0.009
i160-021	160	12720	7	1352	1	0.011
i160-022	160	12720	7	1365	1	0.011
i160-023	160	12720	7	1351	1	0.010
i160-024	160	12720	7	1371	1	0.011
i160-025	160	12720	7	1366	1	0.011
i160-031	160	320	7	2170	1	0.001
i160-032	160	320	7	2330	7	0.002
i160-033	160	320	7	2101	9	0.001
i160-034	160	320	7	2083	2	0.001
i160-035	160	320	7	2103	3	0.001
i160-041	160	2544	7	1494	6	0.005
i160-042	160	2544	7	1486	2	0.007
i160-043	160	2544	7	1549	10	0.008
i160-044	160	2544	7	1478	1	0.004
i160-045	160	2544	7	1554	4	0.006
i160-101	160	240	12	3859	2	0.001
i160-102	160	240	12	3747	1	0.001
i160-103	160	240	12	3837	2	0.001
i160-104	160	240	12	4063	2	0.001
i160-105	160	240	12	3563	1	0.001
i160-111	160	812	12	2869	22	0.009
i160-112	160	812	12	2924	40	0.011
i160-113	160	812	12	2866	11	0.005
i160-114	160	812	12	2989	42	0.014
i160-115	160	812	12	2937	34	0.013
i160-121	160	12720	12	2363	1	0.019
i160-122	160	12720	12	2348	1	0.018
i160-123	160	12720	12	2355	1	0.017
i160-124	160	12720	12	2352	1	0.019
i160-125	160	12720	12	2351	1	0.019
i160-131	160	320	12	3356	9	0.002
i160-132	160	320	12	3450	10	0.001
i160-133	160	320	12	3585	5	0.001
i160-134	160	320	12	3470	1	0.001
i160-135	160	320	12	3716	8	0.002
i160-141	160	2544	12	2549	5	0.006
i160-142	160	2544	12	2562	30	0.020
i160-143	160	2544	12	2557	30	0.028
i160-144	160	2544	12	2607	14	0.014
i160-145	160	2544	12	2578	26	0.021
i160-201	160	240	24	6923	18	0.003
i160-202	160	240	24	6930	2	0.001
i160-203	160	240	24	7243	20	0.002
i160-204	160	240	24	7068	20	0.003
i160-205	160	240	24	7122	25	0.003
i160-211	160	812	24	5583	126	0.041
i160-212	160	812	24	5643	205	0.069
i160-213	160	812	24	5647	192	0.050
i160-214	160	812	24	5720	273	0.092
i160-215	160	812	24	5518	29	0.016
i160-221	160	12720	24	4729	1	0.026
i160-222	160	12720	24	4697	1	0.024
i160-223	160	12720	24	4730	1	0.027
i160-224	160	12720	24	4721	1	0.028
i160-225	160	12720	24	4728	1	0.027
i160-231	160	320	24	6662	64	0.008
i160-232	160	320	24	6558	72	0.011
i160-233	160	320	24	6339	15	0.003
i160-234	160	320	24	6594	6	0.001
i160-235	160	320	24	6764	14	0.004
i160-241	160	2544	24	5086	119	0.075
i160-242	160	2544	24	5106	200	0.126
i160-243	160	2544	24	5050	63	0.046
i160-244	160	2544	24	5076	167	0.147
i160-245	160	2544	24	5084	63	0.050
i160-301	160	240	40	11816	30	0.004
i160-302	160	240	40	11497	19	0.002
i160-303	160	240	40	11445	17	0.002
i160-304	160	240	40	11448	46	0.005
i160-305	160	240	40	11423	48	0.008
i160-311	160	812	40	9135	636	0.196
i160-312	160	812	40	9052	986	0.287
i160-313	160	812	40	9159	1206	0.392
i160-314	160	812	40	8941	234	0.100
i160-315	160	812	40	9086	581	0.196
i160-321	160	12720	40	7876	8	0.040
i160-322	160	12720	40	7859	4	0.048
i160-323	160	12720	40	7876	4	0.028
i160-324	160	12720	40	7884	8	0.055
i160-325	160	12720	40	7862	12	0.076
i160-331	160	320	40	10414	41	0.007
i160-332	160	320	40	10806	130	0.023
i160-333	160	320	40	10561	29	0.006
i160-334	160	320	40	10327	25	0.005
i160-335	160	320	40	10589	31	0.007
i160-341	160	2544	40	8331	252	0.215
i160-342	160	2544	40	8348	659	0.483
i160-343	160	2544	40	8275	298	0.199
i160-344	160	2544	40	8307	217	0.160
i160-345	160	2544	40	8327	493	0.382

Table 17. Branch-and-bound on series i320: dimensions, solution, branch-and-bound nodes, and running time in seconds.

INSTANCE	$\|V\|$	$\|E\|$	$\|T\|$	OPT	BB	TIME [s]
i320-001	320	480	8	2672	1	0.001
i320-002	320	480	8	2847	1	0.001
i320-003	320	480	8	2972	5	0.002
i320-004	320	480	8	2905	5	0.002
i320-005	320	480	8	2991	5	0.004
i320-011	320	1845	8	2053	16	0.013
i320-012	320	1845	8	1997	5	0.007
i320-013	320	1845	8	2072	19	0.014
i320-014	320	1845	8	2061	18	0.014
i320-015	320	1845	8	2059	17	0.015
i320-021	320	51040	8	1553	1	0.050
i320-022	320	51040	8	1565	1	0.050
i320-023	320	51040	8	1549	1	0.052
i320-024	320	51040	8	1553	1	0.049
i320-025	320	51040	8	1550	1	0.051
i320-031	320	640	8	2673	7	0.005
i320-032	320	640	8	2770	7	0.002
i320-033	320	640	8	2769	1	0.002
i320-034	320	640	8	2521	1	0.001
i320-035	320	640	8	2385	2	0.001
i320-041	320	10208	8	1707	23	0.031
i320-042	320	10208	8	1682	2	0.024
i320-043	320	10208	8	1723	9	0.023
i320-044	320	10208	8	1681	1	0.020
i320-045	320	10208	8	1686	1	0.018
i320-101	320	480	17	5548	3	0.001
i320-102	320	480	17	5556	20	0.003
i320-103	320	480	17	6239	19	0.004
i320-104	320	480	17	5703	13	0.005
i320-105	320	480	17	5928	48	0.008
i320-111	320	1845	17	4273	46	0.020
i320-112	320	1845	17	4213	103	0.057
i320-113	320	1845	17	4205	60	0.041
i320-114	320	1845	17	4104	57	0.027
i320-115	320	1845	17	4238	82	0.044
i320-121	320	51040	17	3321	1	0.092
i320-122	320	51040	17	3314	1	0.090
i320-123	320	51040	17	3332	1	0.098
i320-124	320	51040	17	3323	1	0.094
i320-125	320	51040	17	3340	1	0.099
i320-131	320	640	17	5255	13	0.006
i320-132	320	640	17	5052	12	0.003
i320-133	320	640	17	5125	5	0.004
i320-134	320	640	17	5272	14	0.005
i320-135	320	640	17	5342	15	0.005
i320-141	320	10208	17	3606	69	0.153
i320-142	320	10208	17	3567	54	0.122
i320-143	320	10208	17	3561	8	0.061
i320-144	320	10208	17	3512	2	0.020
i320-145	320	10208	17	3601	55	0.083
i320-201	320	480	34	10044	19	0.005
i320-202	320	480	34	11223	20	0.005
i320-203	320	480	34	10148	89	0.017
i320-204	320	480	34	10275	43	0.010
i320-205	320	480	34	10573	44	0.007
i320-211	320	1845	34	8039	291	0.177
i320-212	320	1845	34	8044	407	0.235
i320-213	320	1845	34	7984	586	0.405
i320-214	320	1845	34	8046	610	0.337
i320-215	320	1845	34	8015	4347	2.282
i320-221	320	51040	34	6679	9	0.119
i320-222	320	51040	34	6686	7	0.144
i320-223	320	51040	34	6695	5	0.145
i320-224	320	51040	34	6694	3	0.156
i320-225	320	51040	34	6691	8	0.185
i320-231	320	640	34	9862	127	0.025
i320-232	320	640	34	9933	61	0.021
i320-233	320	640	34	9787	11	0.004
i320-234	320	640	34	9517	89	0.026
i320-235	320	640	34	9945	49	0.015
i320-241	320	10208	34	7027	97	0.223
i320-242	320	10208	34	7072	885	1.134
i320-243	320	10208	34	7044	435	0.628
i320-244	320	10208	34	7078	441	0.694
i320-245	320	10208	34	7046	253	0.441
i320-301	320	480	80	23279	121	0.030
i320-302	320	480	80	23387	74	0.019
i320-303	320	480	80	22693	60	0.025
i320-304	320	480	80	23451	102	0.028
i320-305	320	480	80	22547	88	0.019
i320-311	320	1845	80	17945	98464	85.034
i320-312	320	1845	80	18122	367281	280.508
i320-313	320	1845	80	17991	312917	216.998
i320-314	320	1845	80	18088	394681	318.779
i320-315	320	1845	80	17987	309504	265.875
i320-321	320	51040	80	15648	36	0.583
i320-322	320	51040	80	15646	128	1.119
i320-323	320	51040	80	15654	48	0.662
i320-324	320	51040	80	15667	150	1.247
i320-325	320	51040	80	15649	46	0.663
i320-331	320	640	80	21517	2524	0.554
i320-332	320	640	80	21674	107	0.037
i320-333	320	640	80	21339	1249	0.288
i320-334	320	640	80	21415	262	0.067
i320-335	320	640	80	21378	519	0.165
i320-341	320	10208	80	16296	37178	67.850
i320-342	320	10208	80	16228	871	2.132
i320-343	320	10208	80	16281	6122	12.222
i320-344	320	10208	80	16295	11974	22.977
i320-345	320	10208	80	16289	16871	32.567

Table 18. Branch-and-bound on series i640 (excluding i640-31*): dimensions, solution, branch-and-bound nodes, and running time in seconds.

INSTANCE	$\|V\|$	$\|E\|$	$\|T\|$	OPT	BB	TIME [s]
i640-001	640	960	9	4033	2	0.002
i640-002	640	960	9	3588	3	0.004
i640-003	640	960	9	3438	2	0.002
i640-004	640	960	9	4000	5	0.009
i640-005	640	960	9	4006	4	0.006
i640-011	640	4135	9	2392	1	0.011
i640-012	640	4135	9	2465	8	0.014
i640-013	640	4135	9	2399	9	0.023
i640-014	640	4135	9	2171	2	0.008
i640-015	640	4135	9	2347	11	0.009
i640-021	640	204480	9	1749	1	0.242
i640-022	640	204480	9	1756	1	0.245
i640-023	640	204480	9	1754	1	0.246
i640-024	640	204480	9	1751	1	0.247
i640-025	640	204480	9	1745	1	0.247
i640-031	640	1280	9	3278	4	0.004
i640-032	640	1280	9	3187	1	0.002
i640-033	640	1280	9	3260	12	0.008
i640-034	640	1280	9	2953	3	0.002
i640-035	640	1280	9	3292	3	0.008
i640-041	640	40896	9	1897	9	0.180
i640-042	640	40896	9	1934	14	0.206
i640-043	640	40896	9	1931	24	0.116
i640-044	640	40896	9	1938	14	0.102
i640-045	640	40896	9	1866	1	0.079
i640-101	640	960	25	8764	27	0.012
i640-102	640	960	25	9109	18	0.009
i640-103	640	960	25	8819	6	0.006
i640-104	640	960	25	9040	15	0.010
i640-105	640	960	25	9623	54	0.025
i640-111	640	4135	25	6167	479	0.410
i640-112	640	4135	25	6304	590	0.541
i640-113	640	4135	25	6249	750	0.894
i640-114	640	4135	25	6308	656	0.670
i640-115	640	4135	25	6217	372	0.451
i640-121	640	204480	25	4906	1	0.483
i640-122	640	204480	25	4911	1	0.434
i640-123	640	204480	25	4913	1	0.482
i640-124	640	204480	25	4906	2	0.515
i640-125	640	204480	25	4920	1	0.488
i640-131	640	1280	25	8097	30	0.019
i640-132	640	1280	25	8154	49	0.022
i640-133	640	1280	25	8021	22	0.023
i640-134	640	1280	25	7754	15	0.006
i640-135	640	1280	25	7696	15	0.010
i640-141	640	40896	25	5199	189	0.770
i640-142	640	40896	25	5193	125	0.525
i640-143	640	40896	25	5194	140	0.600
i640-144	640	40896	25	5205	126	0.478
i640-145	640	40896	25	5218	455	1.306
i640-201	640	960	50	16079	143	0.037
i640-202	640	960	50	16324	27	0.017
i640-203	640	960	50	16124	97	0.041
i640-204	640	960	50	16239	115	0.042
i640-205	640	960	50	16616	57	0.016
i640-211	640	4135	50	11984	711303	830.893
i640-212	640	4135	50	11795	16698	24.899
i640-213	640	4135	50	11879	77595	73.637
i640-214	640	4135	50	11898	198891	219.998
i640-215	640	4135	50	12081	117768	135.971
i640-221	640	204480	50	9821	39	1.605
i640-222	640	204480	50	9798	29	1.088
i640-223	640	204480	50	9811	24	1.354
i640-224	640	204480	50	9805	24	0.928
i640-225	640	204480	50	9807	30	1.620
i640-231	640	1280	50	15014	376	0.161
i640-232	640	1280	50	14630	135	0.055
i640-233	640	1280	50	14797	165	0.119
i640-234	640	1280	50	15203	51	0.034
i640-235	640	1280	50	14803	1511	0.553
i640-241	640	40896	50	10230	1770	5.761
i640-242	640	40896	50	10195	523	1.944
i640-243	640	40896	50	10215	697	2.484
i640-244	640	40896	50	10246	5528	14.551
i640-245	640	40896	50	10223	1179	4.289
i640-301	640	960	160	45005	584	0.250
i640-302	640	960	160	45736	1422	0.623
i640-303	640	960	160	44922	269	0.111
i640-304	640	960	160	46233	404	0.176
i640-305	640	960	160	45902	646	0.285
i640-321	640	204480	160	31094	5157	152.268
i640-322	640	204480	160	31068	1028	36.353
i640-323	640	204480	160	31080	755	34.559
i640-324	640	204480	160	31092	1984	67.018
i640-325	640	204480	160	31081	2161	70.447
i640-331	640	1280	160	42796	24756	11.496
i640-332	640	1280	160	42548	37632	21.777
i640-333	640	1280	160	42345	80508	36.600
i640-334	640	1280	160	42768	210040	136.386
i640-335	640	1280	160	43035	99065	59.873
i640-341	640	40896	160	32042	962716	6753.292
i640-342	640	40896	160	31978	71500	586.469
i640-343	640	40896	160	32015	589608	3834.407
i640-344	640	40896	160	31991	237684	1772.469
i640-345	640	40896	160	31994	308439	1892.752

Table 19. Branch-and-bound

INSTANCE	$\|V\|$	$\|E\|$	$\|T\|$	OPT	BB	TIME [s]
bipe2p	550	5013	50	5616	158071	202.060
bipe2u	550	5013	50	54	473584	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	20660	4.399
cc3-5u	125	750	13	36	39892	3.669
cc5-3p	243	1215	27	7299	2102429	1196.743
cc5-3u	243	1215	27	71	1904887	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	1979435	366.055
hc7u	128	448	64	77	6471809	768.549