Fast Local Search for Steiner Trees in Graphs

Eduardo Uchoa* Renato F. Werneck†

Abstract

We present efficient algorithms that implement four local searches for the Steiner problem in graphs: vertex insertion, vertex elimination, key-path exchange, and key-vertex elimination. In each case, we show how to find an improving solution (or prove that none exists in the neighborhood) in $O(m \log n)$ time on graphs with n vertices and m edges. Many of the techniques and data structures we use are relevant in the study of dynamic graphs in general, beyond Steiner trees. Besides the theoretical interest, our results have practical impact: these local searches have been shown to find good-quality solutions in practice, but high running times limited their applicability.

1 Introduction

In the *Steiner problem in graphs*, we are given an undirected graph $G = (V, E)$ with positive edge costs and a set $T \subseteq V$ of *terminals*, and our goal is to find the minimum-cost tree connecting all terminals. The tree may contain elements from $V \setminus T$, known as *Steiner vertices*. This is a classical NP-hard problem [11], and its best known approximation ratio is 1.55 [25].

Its applications in many areas (such as circuit design, networking, and computational biology [3]) have led to a vast literature on heuristics and exact (exponential) algorithms to deal with instances in practice [7, 12, 19, 21, 22, 24, 32]. (For more theoretically-oriented exact algorithms, see e.g. [16] and references therein.) Practical algorithms use tools such as linear relaxations, branch-and-bound, reduction tests, and primal and dual heuristics. In particular, a wide variety of solvers [1, 6, 7, 18, 23, 24] use local search to find near-optimum solutions for several benchmark instances [13] (including, in some cases, the best currently known solutions). Unfortunately, existing local search implementations are rather slow, which limits their applicability.

A local search algorithm tries to improve an existing solution S (any subtree of G containing T) by modifying it slightly. It examines 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 $\mathcal{N}(S)$. Note that this definition allows opportunistic moves while searching $\mathcal{N}(S)$, possibly leading to improving solutions outside $\mathcal{N}(S)$. In general, larger neighborhoods are more likely to contain an improving solution, but more expensive to evaluate. A local search heuristic repeatedly replaces the current solution by an improving neighbor, eventually reaching a *local minimum*.

This paper shows how to evaluate four natural and well-studied neighborhoods in $O(m \log n)$ time (with $m = |E|$ and $n = |V|$). 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 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 $\mathrm{MST}(G[V_S])$. Section 2 shows how dynamic graph techniques can be used to evaluate in $O(m \log n)$ time the neighborhoods based on the insertion or removal of a single Steiner vertex [15, 17, 29, 34]. In the Steiner tree literature, the best reported bounds were $O(n^2)$ for insertions and $O(mn)$ for removals [1, 23, 24].

The neighborhoods we study in Section 3 describe a solution S in terms of its *key vertices* K_S, i.e., 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 in which each edge represents the corresponding shortest path in G). We show how to evaluate a neighborhood based on the elimination of key vertices in $O(m \log n)$ time. We get the same bound for a neighborhood based on *key paths*, which link vertices from $K_S \cup T$ in S. The corresponding *key-path exchange* local search [5, 6, 33] tries to replace an existing key path by a shorter path between the components it connects. In both cases, the best known bound was $O(|T|(m + n \log n))$.

We believe our techniques are interesting in their own right, as are the theoretical results. The experi-

*Universidade Federal Fluminense, Departamento de Engenharia de Produção, R. Passo da Pátria 156, Niterói, RJ 24210, Brazil. E-mail: uchoa@producao.uff.br.

†Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, United States. E-mail: renatow@microsoft.com.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

ments in Section 5 show that our methods are also practical, confirming that nontrivial data structures and algorithmic techniques can play an important role in the solution of real-world optimization problems.

2 Steiner Vertices

Let $S = (V_S, E_S)$ be a starting solution, where V_S is its set of vertices and E_S its set of edges. We assume that $S = MST(G[V_S])$ and all degree-one vertices are terminals (S can be trivially improved otherwise). This section considers two local searches, based on the insertion or removal of a single Steiner vertex from S. Although general dynamic MST algorithms could be used to evaluate these neighborhoods in $O(m \,\mathrm{polylog}\, n)$ time, they are barely practical [2]. Our application is more restricted in nature (it does not need to support deletions and insertions in arbitrary order), which makes our solutions not only simpler, but also asymptotically (and empirically) faster.

2.1 Steiner-Vertex Insertion. The local search based on vertex insertions must determine if there is a vertex $v \notin V_S$ such that $\mathrm{MST}(G[V_S \cup \{v\}])$ is cheaper than S. Given a candidate $v \notin V_S$, let $E(S, v) = \{(v, w) \mid w \in V_S\}$ be the set of edges in G that connect v to V_S. Instead of computing $\mathrm{MST}(G[V_S \cup \{v\}])$ from scratch, Spira and Pan [29] proved it is enough to determine the MST of $G' = (V_S \cup \{v\}, E_S \cup E(S, v))$. They showed that this MST can be computed in $O(n)$ time, allowing the entire neighborhood to be evaluated in $O(n^2)$ time [15].

We propose a simple technique that is much faster for most instances—the exceptions are extremely dense graphs, which are not common in practical applications. Starting from E_S (which forms a tree), we add the edges in $E(S, v)$ one at a time. Let S_i be the MST of G' restricted to the edges in $E_S \cup \{e_1, e_2, \ldots, e_i\}$, where e_i is the ith edge in $E(S, v)$ (in some arbitrary order). Note that $S_{|E(S,v)|}$ is the neighboring solution we seek. By definition, $S_1 = (V_S \cup \{v\}, E_S \cup \{e_1\})$. For $i \geq 2$, we compute S_i from S_{i-1} by trying to insert edge $e_i = (v, w_i)$. We first look for the longest edge f on the unique path in S_{i-1} between v and w_i. If f is costlier than e_i, we remove f from the tree and insert e_i instead. This is a straightforward application of Tarjan's *red rule* [30]: the heaviest edge on any cycle does not belong to the MST.

If we represent the solution as a dynamic tree data structure (such as ST-trees [27, 28], also known as link-cut trees), it takes $O(\log n)$ time to perform each basic operation: finding the edge f, removing (*cutting*) it, and inserting (*linking*) e_i into S_i. If the neighboring solution $S_{|E(S,v)|}$ does not improve on S, we restore the original tree by removing the edges incident to v and reinserting the ones they replaced.

To evaluate the entire neighborhood, we repeat this procedure for all vertices $v \in V \setminus V_S$. Since each edge is a candidate for insertion at most once, we have:

THEOREM 2.1. *Steiner-vertex insertion can be evaluated in $O(m \log n)$ time.*

2.2 Steiner-Vertex Elimination. We now consider the elimination of Steiner vertices. We must determine if there is a vertex $v \in V_S \setminus T$ such that $\mathrm{MST}(G[V_S \setminus \{v\}])$ is cheaper than S. Existing implementations [1, 23, 24] simply evaluate each possible removal by essentially rerunning Kruskal's algorithm on the entire (presorted) list of edges in the induced subgraph, taking $\Omega(mn)$ total time.

The dynamic graphs literature offers an asymptotically better solution to this problem (known there as "all nodes replacement"). Das and Loui [4] proposed an $O(m \log n)$ method based on the simultaneous (implicit) execution of $O(n)$ instances of Kruskal's algorithm, each excluding a single vertex from the tree. We present a different $O(m \log n)$ algorithm, which is conceptually simple and evaluates vertices sequentially. (This allows our implementation to find multiple improvements in a single pass, as Section 4 will explain.) Moreover, we apply the techniques used here in the more general local searches of Section 3.

For efficiency, our method considers S to be a tree rooted at an arbitrary terminal r; all leaves are also terminals. Vertices are processed in post-order with respect to this tree, ensuring parents are only considered after their children. To process v, we first (temporarily) remove v from the tree. Let $S_1, \ldots, S_k$ be the subtrees rooted at v's original children $(v_1, \ldots, v_k)$ and let S_0 be the component containing r. It can be shown [4] that there exists a minimum spanning forest of $G[V_S \setminus \{v\}]$ containing every edge in $E_S \setminus E(v)$, where $E(v)$ is the set of all edges in G incident to v. Hence, it suffices to contract (implicitly) each subtree S_i into a *supervertex*, compute the MST of the subgraph of G induced by the supervertices, and check whether it is cheaper than S.

Examining all (up to $\Theta(m)$) edges in this subgraph can be too expensive, however. Instead, we subdivide them in two groups: a *vertical* edge has exactly one endpoint in S_0 (and another in S_i, for $i > 0$), while a *horizontal* edge has endpoints in S_i and S_j, for $0 < i < j \leq k$. (See Figure 1.) Note that these groups are defined relative to v.

An edge $e = (x, y)$ is horizontal with respect to at most one vertex v, namely the nearest common ancestor in S of x and y (which we denote by $nca(x, y)$). We therefore keep with each vertex w a list $L(w)$ of all

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

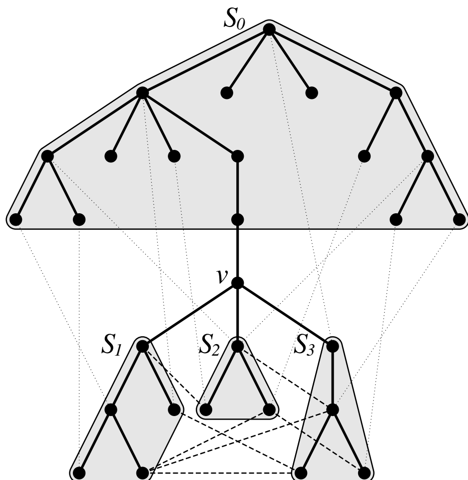

Figure 1: Steiner-vertex elimination: state when processing vertex v. Solution edges are solid, horizontal edges are dashed, and vertical edges are dotted. If v and its incident edges are removed, we must reconnect the resulting subtrees using the horizontal and vertical edges.

edges $e = (x, y)$ in G such that $nca(x, y) = w$. The lists, which have $O(m)$ edges in total, can be built in a preprocessing stage in $O(m \log n)$ time with dynamic trees, or in $O(m)$ time with a specialized algorithm [10].

We cannot afford to keep similar lists of vertical edges, since each edge (x, y) can be vertical with respect to several vertices v (those on the tree paths from x or y to $nca(x, y)$). Fortunately, we do not need to. Since the MST we are computing has at most one edge between S_0 and S_i (for any fixed $i > 0$), we only need to know the *cheapest* vertical edge incident to each such S_i.

We call a vertex *active* if it has already been processed, but its parent has not. To help find vertical edges, each active vertex x keeps a heap $H(x)$ containing edges incident to the subtree rooted at x, with cheaper edges given higher priority. Initially, when only leaves are active, the heap $H(x)$ associated with each leaf x is created with all edges (x, y) with $y \in S$. When an internal vertex becomes active, its heap is created from a combination of its children's heaps (this is why vertices are processed in post-order). Any logarithmic-time meldable heap, such as pairing [8], Fibonacci [9], or leftist heaps [30], can be used.

With these data structures, processing v is straightforward. We know every child v_i $(i > 0)$ of v is active. We *clean up* each heap $H(v_i)$ by calling *extractmin* until the top element is a vertical edge, with one endpoint in S_i and another in S_0 (edges with both endpoints in S_i or one in $\{v\}$ are discarded). We build a set of candidate edges by taking the top element from each heap (vertical edges) and all elements in $L(v)$ (horizontal edges). Finally, we run an algorithm such as Prim's to find the MST of the graph induced by the sets S_i.[1] This results represents the neighboring solution. We use a union-find data structure [30] to maintain the supervertices representing each S_i $(i > 0)$.

Once v is processed, it becomes active. We create $H(v)$ by merging the heaps $H(v_i)$ associated with its (now inactive) children; similar updates are done to the union-find data structure. Finally, we add to $H(v)$ all edges (v, w) with $w \notin S_i$, $i > 0$.

Theorem 2.2. *Steiner-vertex elimination can be evaluated in $O(m \log n)$ time.*

Proof. The lists $L(v)$ can be built in linear time. Heap operations take $O(m \log n)$ time: each edge is inserted into (and removed from) a heap at most twice, and there are $O(n)$ merges and independent *findmin* queries. Finally, the MST computation when processing vertex v takes $O(m_v \log n)$ time, where m_v is the number of vertical and horizontal edges in the computation. Together, the MST computations take $O(m \log n)$ time, since each edge is horizontal at most once and the MST algorithm looks at only $O(n)$ vertical edges in total (one for each subtree processed). □

3 Key Vertices

We now consider local searches based on *key vertices*, which are more challenging but usually more effective. Given a solution $S = (V_S, E_S)$, a key vertex is a nonterminal $v \in V_S$ with degree at least three in S. We can succinctly describe S by its set K_S of key vertices. Given K_S, we can build a solution S' that is no more expensive than S as follows. Let $C_S = K_S \cup T$ be the set of *crucial vertices* [6] associated with S. Let $D(G, C_S)$ be the *distance network* associated with C_S: a complete graph with C_S as its set of vertices in which the cost of edge (v, w) is given by the length of the shortest path in G between v and w. The MST of $D(G, C_S)$ costs no more than S; if S is optimal, they cost the same.

Section 3.3 shows how to evaluate in $O(m \log n)$ time a natural neighborhood based on the elimination of a single key vertex. (Unfortunately, we could not get similar bounds for insertions.) Section 3.2 studies a neighborhood based on key path exchanges due to Verhoeven et al. [33] (see also [5, 34]). A *key path* connects two crucial vertices in S and has no internal crucial vertex. We can determine in $O(m \log n)$ time

[1] For simplicity, we use standard $O(m \log n)$ algorithms to compute MSTs and single-source shortest paths in this paper, since our local searches have other bottlenecks.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

if there is a key path in S that can be replaced by a shorter one.

3.1 Voronoi Diagrams. Both local searches use *Voronoi diagrams*. Given $G = (V, E)$ and a set $A \subseteq V$, a *Voronoi diagram of* V *with respect to* A is a partition of V into $|A|$ connected regions such that (1) each region contains exactly one vertex of A (the *base* of this region) and (2) each vertex v in $V \setminus A$ is assigned to the region whose base is closest to v. This generalizes Voronoi diagrams in geometric settings.

The diagram associates three pieces of information with each vertex v: $base(v)$ is the base of the region containing v; $p(v)$ is the predecessor of v on the shortest path from $base(v)$ (if v is a base, then $p(v) = v$); and $vdist(v)$ is the distance from $base(v)$ to v. Ties are broken arbitrarily, as long as $base(v) = base(w)$ when $p(v) = w$. The Voronoi diagram can be built with a slightly modified version of Dijkstra's algorithm having all vertices in A as sources. (Equivalently, one can run Dijkstra's algorithm from a single artificial source connected to all vertices in A by zero-length edges.) This takes $O(m + n \log n)$ time with Fibonacci heaps, or $O(m \log n)$ with binary heaps [14].

Mehlhorn [14] suggested using Voronoi diagrams to implement the *distance network heuristic* (DNH), a 2-approximate constructive algorithm for Steiner trees. This heuristic first finds the MST of $D(G, T)$, the distance network associated with T, then transforms it into a Steiner tree of G by "expanding" each MST edge into the corresponding edges in G. A direct implementation would require $|T|$ single-source shortest path computations in the original graph and $\Theta(|T|^2)$ space. Instead, Mehlhorn's implementation computes the MST in $O(m + n \log n)$ time without ever building $D(G, T)$. It first builds the Voronoi diagram with respect to T, then creates an auxiliary graph G' on $|T|$ vertices, corresponding to the original terminals. For each *boundary edge* (v, w) in the Voronoi diagram (i.e., edges with $base(v) \neq base(w)$), G' has an edge between $base(v)$ and $base(w)$ in G' with cost $vdist(v) + cost(v, w) + vdist(w)$. Note that G' has no more edges than G. Finally, it computes the MST of G' and expands its edges to obtain a tree in G. As Mehlhorn showed, the MSTs of G' and $D(G, T)$ cost the same.

Our local searches use this method to compute the MST of $D(G, A)$ for arbitrary sets $A \subseteq V$ (not just T). We also need to modify the set of bases of existing Voronoi diagrams: given the Voronoi diagram of G with respect to a set $A \subseteq V$ of bases, we need the diagram with respect to a subset $B \subset A$. Instead of recomputing from scratch, we show how to *repair* the original diagram (with respect to A), converting it into the diagram with respect to B. Let $base_A(v)$ and $base_B(v)$ be the bases of v with respect to A and B, respectively. If $base_A(v) \in B$, then $base_B(v) = base_A(v)$ (v does not change bases if its original base is still available). Only the set C of vertices with original bases in $A \setminus B$ needs updating.

We update the diagram in two stages. The first is initialization, which processes each $v \in C$ in turn. It sets $base(v) \leftarrow null$, $p(v) \leftarrow null$, and $vdist(v) \leftarrow \infty$, then traverses v's adjacency list: if an edge (v, w) is such that $w \notin C$ and $vdist(v) > vdist(w) + cost(v, w)$, it sets $base(v) \leftarrow base(w)$, $p(v) \leftarrow w$, and $vdist(v) \leftarrow vdist(w) + cost(v, w)$. Intuitively, each vertex in C is initially assigned to the region of its best adjacent vertex outside C, if there is any.

The second stage is a modified version of Dijkstra's algorithm (as in the full Voronoi computation) that scans only the vertices in C. The Voronoi diagram can thus be repaired in $O(|E(C)| \log n)$ time, where $E(C)$ is the set of edges in G with at least one endpoint in C. Using this procedure, we can prove the following lemma (which will be useful for the local search implementations):

LEMMA 3.1. *Given a graph $G = (V, E)$, let $A_1, A_2, \ldots, A_s$ be subsets of $A \subseteq V$. If each vertex in A is in at most k subsets A_i, we can build a sequence of s Voronoi diagrams of G, with respect to each set $A \setminus A_i$, in $O(km \log n)$ total time.*

Proof. Start by building in $O(m \log n)$ time the Voronoi diagram with respect to A. Then, for each i use the *repair* operation to create the Voronoi diagram with respect to $A \setminus A_i$, restoring the original diagram (with respect to A) afterwards. The total repair time is $O(\sum_{i=1}^{s} |E(C_i)| \log n)$, where C_i is the set of vertices whose original bases are in A_i. Since each base belongs to at most k sets A_i, this can be bounded by $O(\log n \sum_{v \in V} k|E(v)|) = O(km \log n)$. □

3.2 Key-Path Exchange. We are now ready to discuss our third local search. Given a solution $S = (V_S, E_S)$, we must determine whether it is possible to remove some key path and reconnect the two resulting components more cheaply. Existing implementations [18, 33] process each of the $\Theta(|T|)$ key paths by simply running Dijkstra's algorithm between the two components it determines, which takes $O(|T|(m + n \log n))$ total time with Fibonacci heaps. We now present an $O(m \log n)$ alternative.

During preprocessing, we compute the Voronoi diagram with all vertices in V_S as bases. (This and other steps of the algorithm are illustrated in Figure 2.) Each base v also keeps a heap $H(v)$ con-

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

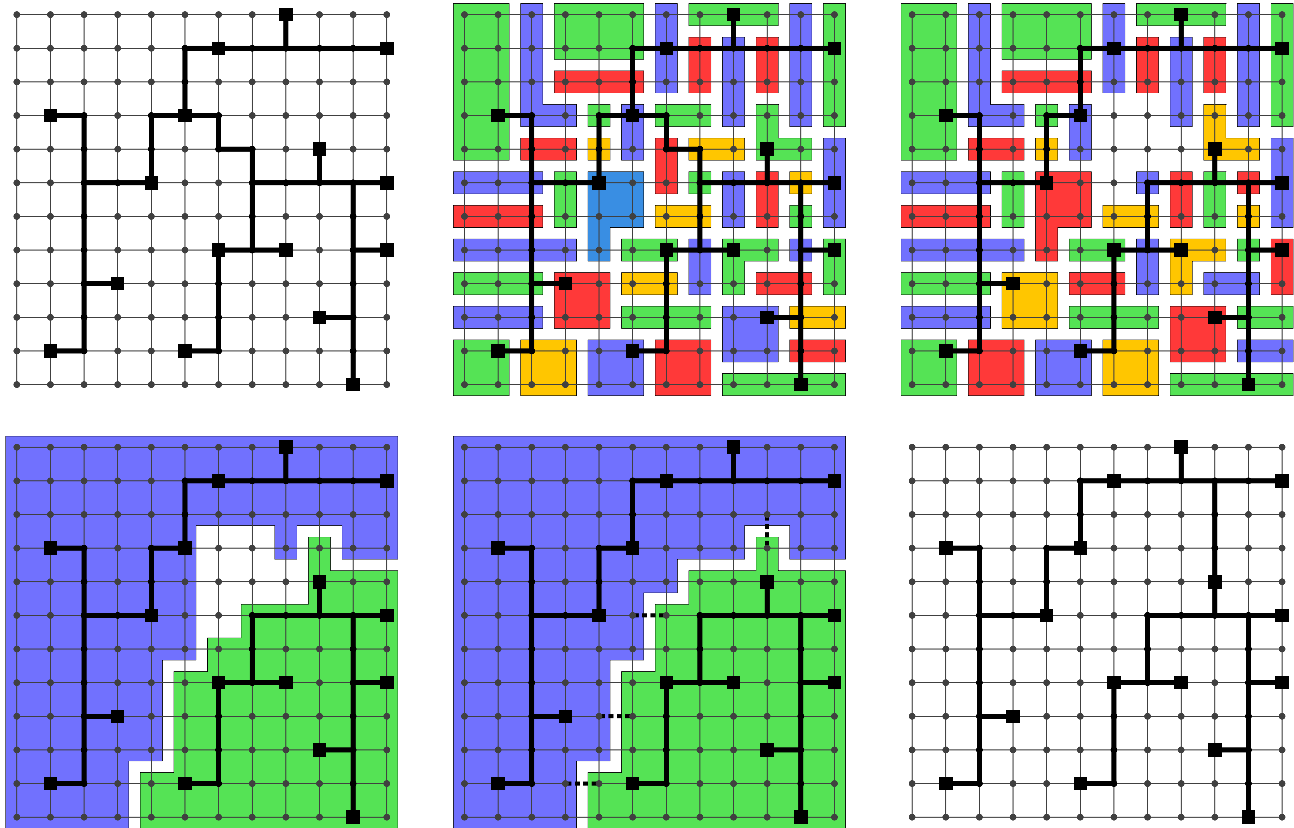

Figure 2: Key-path exchange local search. This example shows a grid graph in which all edges have unit length; nonterminals are represented as small circles and terminals as black squares. From top left: (a) Original solution. (b) Solution with associated Voronoi diagram. (c) State after temporary removal of a key path of length four. (d) Conceptual view of the same state, after the regions associated with the same component are implicitly united; the edges incident to both components are *original boundary edges*. (e) Conceptual view of the Voronoi diagram after it is repaired; there are four boundary edges (shown dashed) representing shortest (length-three) paths between the components (the topmost one is a *new boundary edge*, i.e., it only appeared when the diagram was repaired). (f) Neighboring solution, with a replacement path.

taining all original boundary edges with one endpoint in v's Voronoi region; each boundary edge appears in two heaps. The priority of an edge (x, y) is given by $vdist(x) + cost(x, y) + vdist(y)$, the length of the shortest path between $base(x)$ and $base(y)$ containing (x, y). (We refer to a heap element as a *boundary path*; the *endpoints* of such a path are the Voronoi bases of the endpoints of the corresponding boundary edge.) Since these heaps will be combined (to represent boundary paths incident to entire subtrees of S, instead of a single vertex), we use meldable heaps.

As in Section 2.2, the main loop views S as a tree rooted at a terminal r and processes crucial vertices in post-order. When processing v, we try to replace P_{uv}, the unique key path containing both v and its parent in S. Let I_{uv} be the (possibly empty) set of *internal vertices* of this path (those that are not crucial). Let S_u and S_v be the subtrees created when we remove I_{uv} and its incident edges from S, with $u \in S_u$ and $v \in S_v$. We must find the shortest path between S_u and S_v. (To simplify notation, consider S_u and S_v to be sets of vertices.)

If we had the Voronoi diagram of G with the vertices in $S_u \cup S_v$ as bases, we could just pick the best boundary edge (x, y) that is *relevant*, i.e., with $base(x) \in S_u$ and $base(y) \in S_v$. But the diagram we have has $S_u \cup S_v \cup I_{uv}$ as bases. To process v, we first remove I_{uv} from the set of bases and *repair* the Voronoi diagram using the algorithm in Section 3.1. To find the cheapest boundary path between S_u and S_v in the repaired diagram, we cannot afford to enumerate all corresponding boundary edges. To avoid the enumeration, we conceptually split

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

these edges in two groups: an *original* edge is already on the boundary in the original diagram, while a *new* edge becomes relevant only after the repair. A boundary path shorter than P_{uv} in either group leads to an improving exchange.

We pick the best *new* boundary edge (x, y) (minimizing $vdist(x) + cost(x, y) + vdist(y)$) explicitly, while repairing the diagram. To find the best *original* boundary edge, we use the heap $H(v)$. When processing v, $H(v)$ contains all original boundary paths with one endpoint in S_v (processing vertices in post-order helps ensure this). We must *clean up* $H(v)$ by calling *extractmin* while its top path has two endpoints in S_v or one in I_{uv}; these paths can be discarded. (We keep track of S_v and I_{uv} with a union-find data structure.) The top remaining element represents the shortest original boundary path between S_u and S_v.

Before processing the next key path, we restore the original Voronoi diagram and merge the heaps associated with u, v, and the vertices in I_{uv}, creating an updated version of $H(u)$. The union-find structure must be updated similarly.

Theorem 3.1. *Key-path exchange can be evaluated in* $O(m \log n)$ *time.*

Proof. Computing the original Voronoi diagram takes $O(m \log n)$ time. This is also the total time spent repairing it: just apply Lemma 3.1 with $A = V_S$ and each A_i corresponding to the internal vertices of one key path. The sets A_i are disjoint. Finally, we spend $O(m \log n)$ time on heap operations: each edge is added or removed from a heap at most twice and there are $O(n)$ merges. □

3.3 Key-Vertex Elimination.

We now consider a neighborhood based on the elimination of key vertices. Given a solution S associated with a set C of crucial vertices, we must determine if there is a nonterminal $v \in C$ such that the solution S' associated with $C' = C \setminus \{v\}$ is cheaper. This is analogous to the neighborhood based on the elimination of Steiner vertices when applied to the distance network $D(G, C)$. (As in that case, we assume S is minimal, i.e., it is the result of running DNH with C as the set of bases.) Building $D(G, C)$ explicitly (as in [6]) is too costly, especially if G is sparse. An alternative is to rerun DNH for each removal [18], which saves memory but has the same worst-case complexity: $O(|T|(m + n \log n))$. Instead, we propose an $O(m \log n)$-time algorithm that combines elements of Steiner-vertex elimination (Section 2.2) and key-path exchange (Section 3.2).

As usual, we consider S to be rooted at a terminal r, and process the key vertices in post-order. Intuitively, we process v by temporarily removing v and its incident key paths from S, then computing the MST of the subgraph (of the distance network) induced by the remaining crucial vertices. We do not compute this MST from scratch. The removals create a parent component S_0 and k child components S_i $(1 \le i \le k)$. We treat the components as supervertices and compute the MST of the subgraph they induce on the distance network. As in Section 2.2, this can be made efficient by considering two types of edges: a *horizontal* edge links two child components, while a *vertical* edge links a child component to S_0. Because these edges are actually paths in the original graph, we use Voronoi diagrams to deal with them efficiently, as in Section 3.2.[2]

Given the intuition, we now describe the algorithm in detail. We first compute the Voronoi diagram with all vertices in S as bases. We then associate with each vertex v in S a heap $H(v)$ containing all boundary paths with v as an endpoint. (Once v is processed, $H(v)$ will contain boundary paths with endpoints in the subtree rooted at v.) Shorter paths are given higher priority. We also keep a list $L(v)$ of all horizontal boundary paths associated with v, i.e., boundary edges (u, w) such that $nca(base(u), base(w)) = v$. (As in Section 3.2, we refer to a heap or list element as a boundary path, but actually keep the associated boundary edge.)

To process a key vertex v, we temporarily remove v and its adjacent key paths from the solution, splitting it into a component containing the root (S_0) and $k \ge 2$ child components $(S_1, \ldots, S_k)$. For $0 \le i \le k$, let $v_i \in S_i$ be the other endpoint of the key path from v to S_i, and let I_i be the (possibly empty) set of internal vertices on this path. Processing v entails the following operations:

1. Remove from $L(v)$ all paths with at least one endpoint in I_i (for any $i > 0$). The remaining elements of $L(v)$ represent the original boundary paths between S_i and S_j, with $0 < i < j \le k$.
2. For every $1 \le i \le k$, clean up heap $H(v_i)$ by repeatedly calling *extractmin* until the top path has one endpoint in S_i and another in S_0. This eliminates paths from S_i to v, to S_j (with $0 < j \le k$), and to I_j $(0 \le j \le k)$, which either are inside a single component or will be once v or v_0 are processed. After the cleanup, the top element of $H(v_i)$ represents the shortest original boundary path between S_i and S_0.

[2] Using the original graph allows the algorithm to find paths between degree-two Steiner vertices, thus creating new key vertices. This makes this implementation more powerful than pure key-vertex elimination.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

3. Remove $\{v\} \cup I_0 \cup I_1 \cup \cdots \cup I_k$ from the set of bases and repair the Voronoi diagram. Let $R(v)$ be the set of new boundary paths between S_i and S_j (for $0 \leq i < j \leq k$) found during the execution of the repairing algorithm.
4. Build a set of candidate paths from the union of $R(v)$ (the new boundary edges), $L(v)$ (the original horizontal boundary edges), and the top element of each heap $H(v_i)$ (the original vertical boundary edges). The MST of the graph with $S_0, S_1, \ldots, S_k$ as supervertices and these candidates as edges is the neighboring solution.

We use a union-find data structure to keep track of the relevant components: when processing v, each S_i ($1 \leq i \leq k$) and I_i ($0 \leq i \leq k$) corresponds to a different set.

After processing v, we must restore the invariants for subsequent iterations. We update $H(v)$ by merging it with all heaps $H(u)$ such that $u \in I_i \cup \{v_i\}$, for $i \geq 1$. The heaps associated with v_0 and the vertices in I_0 should not be merged with $H(v)$ yet—this will be done when v_0 is processed. The union-find structure must be updated similarly. Finally, we restore the original Voronoi diagram.

THEOREM 3.2. *Key-vertex elimination can be evaluated in $O(m \log n)$ time.*

Proof. Computing the initial Voronoi diagram and building the lists of horizontal paths takes $O(m \log n)$ time. During the main loop, each base u is removed from the Voronoi diagram at most twice (when the endpoints of the key paths containing u are processed), so Lemma 3.1 ensures all repair operations take $O(m \log n)$ time. Heap operations also cost $O(m \log n)$, since each edge is inserted or removed at most twice. Finally, MST computations deal with three classes of edges: original horizontal edges, original vertical edges, and new boundary edges (found when repairing the diagram). An edge (x, y) is horizontal at most once (when $nca(base(x), base(y))$ is processed); a total of $O(n)$ vertical edges are used in MST computations (one out of each child subtree); and the number of new boundary edges found during repair operations is $O(m)$ (since each original key path is temporarily removed from the tree at most twice, we can apply the proof of Lemma 3.1 with $k = 2$). An MST algorithm takes $O(m \log n)$ total time to process these $O(m)$ edges. □

4 Passes

We call a single execution of an $O(m \log n)$ local search algorithm a *pass*: given a solution S, it must either find an improving solution S' or show that none exists in the neighborhood. In fact, each algorithm we presented explicitly evaluates every element of its neighborhood. An actual implementation could pick the best neighbor after each pass, but we observed that, in practice, a single pass often finds more than one improving move (insertion, removal, or exchange). Performing several moves in a pass can help speed up convergence to a local minimum.

This is easy for Steiner-vertex insertions. Since each successful move is immediately reflected in the dynamic tree data structure, we can safely perform several moves within a pass. In our implementation of this local search, a pass evaluates each vertex once, in random order.

In contrast, the other local searches use data structures that cannot be efficiently updated as soon as a move is performed (such as heaps, DFS trees, Voronoi diagrams, and union-find); we must keep them until the end of the pass. Because they are not up-to-date, however, we must use these data structures conservatively. In all three local searches, a move disconnects a component S_0 containing the root from one or more components S_i, then reconnects them in a different fashion. To prevent conflicts with future moves in the same pass, we first mark as *pinned* the vertices in S_0 that are endpoints of the new paths added by the move. This prevents these endpoints for being removed later during the same pass. Second, we mark all original vertices that are not in S_0 as *forbidden*. These vertices can no longer be used to reconnect the solution until the end of the pass (using these vertices might lead to a disconnected solution, for example).

Maintaining these additional pieces of information takes $O(m)$ time per pass. In particular, whenever we mark a vertex as forbidden, we must also mark all of its descendants in the original tree. Because we do so in DFS fashion, each new move can skip the subtrees marked by previous moves, ensuring each vertex is scanned at most once during a pass.

5 Experiments

We implemented all four $O(m \log n)$ local searches we proposed: Steiner-vertex insertion (denoted by V), Steiner-vertex elimination (U), key-path exchange (P), and key-vertex elimination (J). We also tested Q, which combines P and J on the same pass: to process a key vertex v, we first check if v itself can be removed, then whether the key path with v at the bottom can be exchanged (vertices are still processed in post-order with respect to a DFS tree). This is faster than running P and J in sequence, just as effective, and also dominates U. Finally, we tested method VQ, which simply runs V and Q in sequence in each pass. As explained in Sec-

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Table 1: Instances tested (refer to the SteinLib [13] for details).

CLASS	SERIES	DESCRIPTION
random	b c d e mc p4z p6z	graphs with random costs
hard	bip cc hc sp	synthetic hard instances
fst	es*fst tspfst	reduced geometric instances, L1 costs
euclidean	x p4e p6e	Euclidean costs
vlsi	alue alut dmxa diw gap lin msm taq	planar grid graphs with holes
incidence	i080 i160 i320 i640	random graphs, incidence costs

tion 4, all local searches we implemented perform moves greedily, with multiple independent moves allowed per pass. The root terminal is picked uniformly at random in each pass.

We ran all local searches on solutions found by Mehlhorn's $O(m \log n)$-time distance network heuristic (DNH). We also ran VQ on solutions found by the shortest path heuristic (SPH): like Prim's algorithm, it starts from a single vertex and in each step adds to the solution the shortest path to the closest remaining terminal. Its worst case is $O(|T| m \log n)$, but it is about as fast as DNH in practice and usually finds better solutions [20]. After finding a solution, both DNH and SPH compute the MST of the induced subgraph and prune degree-one nonterminals (this step, MST-prune, is considered part of the constructive heuristic itself). The third construction we tested was RSPH, which runs SPH on $\min\{n, 100\}$ random roots (with priority given to terminals, as in [21]) and returns the best solution found. Finally, we ran a simple multistart heuristic (MS): it is similar to RSPH, but runs VQ after each execution of SPH.

The algorithms were implemented in C# and tested on a 2.4 GHz AMD Opteron CPU with 16 GB of RAM running Windows Server 2003. We use pairing heaps as meldable heaps, binary heaps for Dijkstra-like computations, and a linear-time version of dynamic trees. With this version, path-traversing operations (like nca) take time proportional to the number of vertices visited; as shown in [31], it is faster in practice than using logarithmic versions of dynamic trees when the average path is much shorter than n, which is the case on the graphs we tested. Even in applications in which the strict $O(m \log n)$ upper bound is essential, one could use the linear version by default, but monitoring the total path size. In case it became bigger than $cm \log n$ (for some constant c), the algorithm could simply switch to a logarithmic version of dynamic trees, such as ST-trees. In our experiments, this limit would not be reached even for very small c.

All methods were tested on 1020 instances from the SteinLib, shown in Table 1, with various sizes (up to more than 100,000 vertices), densities (from planar to complete graphs), and numbers of terminals (up to thousands). See [13] for details.

In our first experiment, each local search was run until reaching a local minimum. Table 2 shows the average percentage errors relative to the best known solutions [13, 21]. Table 3 reports average running times, in milliseconds for the basic constructive algorithm (DNH) and relative to DNH for other algorithms (each entry is an average of ratios). The results are thus mostly language- and machine-independent. For accurate timing, for any given instance we ran each algorithm repeatedly for at least one second and took the average time. The times to start the .NET framework and reading the input graph are not included, since they are the same for all methods. But the times we report do include algorithm-specific data structure initialization and explicit garbage collection. Local search times do not include finding the starting solution; MS times include both SPH and local search.

As expected [20, 21], RSPH finds the best solutions among the constructive algorithms, followed by SPH and DNH. All three can be substantially improved by local search. Even individual local searches can be very effective: for instance, P works particularly well on vlsi and fst instances (in which key paths have several hops), V helps the most when the diameter is small, and U is good at hard instances. But the best results are usually found by VQ: in most cases, SPH+VQ and even DNH+VQ find better solutions than RSPH. The solutions obtained by RSPH+VQ were on average within 0.5% of the best known, except on adversarial classes (hard and incidence). MS improves the results even further.

Table 3 shows that local search is fast, even though multiple passes are allowed. A single pass of VQ is actually only two or three times slower than DNH (this is not shown in the table). Indeed, VQ needed fewer than six passes on average, and never more than nineteen. But allowing multiple moves per pass is key to good performance: instance es10000fst01, for example, required more than 1000 moves. On average, VQ needs more than 80 moves to reach a local minimum on hard instances, but only about 5 passes.

If worst-case running times are essential, strictly limiting the number of passes to a small constant has

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Table 2: Local search (until a local minimum is found): average percentage error with respect to the best known solutions. CST refers to the constructive algorithm.

	DNH							SPH		RSPH		MS
CLASS	CST	V	U	P	J	Q	VQ	CST	VQ	CST	VQ	
euclidean	1.95	1.53	1.65	1.17	1.65	0.93	0.52	1.17	0.20	0.25	0.06	0.00
fst	2.31	1.99	1.68	0.86	1.20	0.62	0.55	1.64	0.64	1.09	0.51	0.35
hard	24.06	22.28	11.72	12.68	9.72	8.05	8.02	9.19	5.03	6.41	4.02	3.18
incidence	23.38	9.80	22.05	15.00	19.63	14.11	2.57	17.89	2.77	3.20	1.27	0.54
random	4.88	4.46	3.88	1.60	3.14	0.96	0.92	2.68	1.00	0.86	0.44	0.28
vlsi	5.42	5.34	5.24	1.00	3.90	0.89	0.91	2.81	0.73	1.36	0.37	0.15

Table 3: Local search (until a local minimum is found): average running time (in milliseconds for DNH-CST, relative to DNH-CST for other methods).

	DNH							SPH		RSPH		MS
CLASS	CST	V	U	P	J	Q	VQ	CST	VQ	CST	VQ	
euclidean	*4.43*	0.66	0.69	1.85	1.61	2.42	3.96	0.67	3.88	43.45	3.00	427.85
fst	*2.56*	0.65	1.68	4.20	5.31	6.17	6.48	0.96	5.86	65.48	5.29	601.16
hard	*4.53*	0.76	3.96	9.18	16.88	14.68	14.83	1.00	13.05	75.34	11.10	1309.29
incidence	*5.67*	0.69	0.62	4.98	5.82	8.97	10.85	1.18	9.31	79.58	5.75	942.47
random	*3.87*	0.54	1.16	3.58	4.13	6.15	6.19	0.83	5.12	62.64	4.16	590.51
vlsi	*8.75*	0.30	0.52	5.79	4.38	7.66	7.74	0.85	6.78	75.52	5.52	766.08

little adverse effect on solution quality. For example, if at most three passes are executed, the average percentage errors for DNH+VQ are less than 25% higher that what Table 2 shows, for all classes. The worst case happens on incidence, when the relative error increases from 2.57% (as shown in Table 2) to 3.16% if only three passes are allowed, an increase of almost 23%. On the non-adversarial classes, in contrast, the relative error increases by less than 10%.

The goal of our experiments was to illustrate the effectiveness of local search. Indeed, local search by itself is good enough for some applications, especially those with strict time constraints. We emphasize, however, that a state-of-the-art solver must include other techniques as well, such as reduction tests and duality-based approaches [12, 19, 21, 22, 32]. In particular, the elaborate algorithms of Polzin and Daneshmand can quickly find optimum or near-optimal solutions to several classes of instances [21]. Our techniques are orthogonal and can be combined in several straightforward ways. For example, they could use solutions found by local search (instead of by RSPH) to guide heuristic reductions. More generally, reduction and dual-based techniques can significantly reduce the size of the input instance, so local search could be applied to the final result.

When reduction tests and dual-based techniques are less effective, primal-only metaheuristics based on local search have been instrumental in finding good solutions [1, 7, 18, 24]. For almost all open SteinLib instances (in classes hard and incidence), the best known solutions were found by methods based on local search [18, 26]. Our $O(m \log n)$ implementations can make heuristics such as these much faster—asympotically so. For example, on instance es10000fst01 (with $n = 27\,019$, $m = 39\,407$, and $|T| = 10\,000$), the local search described in [18] (a combination of U, V, and P) is about 500 times slower to reach a local optimum than our implementation (on the same machine).

Acknowledgements. We thank numerous anonymous referees for their helpful suggestions.

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*, pages 39–58. Kluwer, 2001.

[2] G. Cattaneo, P. Faruolo, U. F. Petrillo, and G. F. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. In *Proc. 4th ALENEX*, volume 2409 of *LNCS*, pages 111–125. Springer, 2002.

[3] X. Cheng and D.-Z. Du. *Steiner Trees in Industry.* Springer, 2002.

[4] B. Das and M. C. Loui. Reconstructing a minimum spanning tree after deletion of any node. *Algorithmica*, 31(4):530–547, 2001.

[5] K.A. Dowsland. Hill-climbing, simulated annealing and the Steiner problem in graphs. *Engineering Optimization*, 17:91–107, 1991.

[6] C. Duin and S. Voß. Efficient path and vertex exchange in Steiner tree algorithms. *Networks*, 29:89–105, 1997.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

[7] 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.

[8] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[9] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[10] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing*, 17:338–355, 1984.

[11] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[12] T. Koch and A. Martin. Solving Steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[13] 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. `http://elib.zib.de/steinlib`.

[14] K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.

[15] M. Minoux. Efficient greedy heuristics for Steiner tree problems using reoptimization and supermodularity. *INFOR*, 28:221–233, 1990.

[16] J. Nederlof. Fast polynomial-space algorithms using möbius inversion: Improving on steiner tree and related problems. In *Proc. 36th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 5555 of *LNCS*, pages 713–725. Springer, 2009.

[17] 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.

[18] 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*, pages 429–433, Porto, Portugal, 2001.

[19] 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*, volume 7 of *Elec. Notes in Disc. Math.*, 2001.

[20] 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 ALENEX*, volume 2409 of *LNCS*, pages 1–15. Springer, 2002.

[21] T. Polzin. *Algorithms for the Steiner Problem in Networks*. PhD thesis, Universität des Saarlandes, 2003.

[22] T. Polzin and S. Vahdati Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112(1–3):263–300, 2001.

[23] C. C. Ribeiro and M. C. Souza. Tabu search for the Steiner problem in graphs. *Networks*, 36:138–146, 2000.

[24] 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.

[25] G. Robins and A. Zelikovsky. Tighter bounds for graph Steiner tree approximation. *SIAM Journal on Discrete Mathematics*, 19(1):122–134, 2005.

[26] 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*, pages 557–591, Porto, Portugal, 2001.

[27] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[28] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[29] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Computing*, 4(3):375–380, 1975.

[30] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[31] R. E. Tarjan and R. F. Werneck. Dynamic trees in practice. In C. Demetrescu, editor, *Proc. 6th WEA*, volume 4525 of *LNCS*, pages 80–93, 2007.

[32] E. Uchoa, M. Poggi de Aragão, and C. C. Ribeiro. Preprocessing Steiner problems from VLSI layout. *Networks*, 40(1):38–50, 2002.

[33] 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.

[34] S. Voß. Steiner's problem in graphs: Heuristic methods. *Discrete Applied Mathematics*, 40(1):45–72, 1992.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.