On the Implementation of MST-based Heuristics for the Steiner Problem in Graphs

Marcus Poggi de Aragão and Renato F. Werneck$^*$

Department of Informatics, Catholic University of Rio de Janeiro, R. Marquês de São Vicente, 225, Rio de Janeiro, RJ, 22453-900, Brazil.

poggi@inf.puc-rio.br, rwerneck@cs.princeton.edu

Abstract. Some of the most widely used constructive heuristics for the Steiner Problem in Graphs are based on algorithms for the Minimum Spanning Tree problem. In this paper, we examine efficient implementations of heuristics based on the classic algorithms by Prim, Kruskal, and Borůvka. An extensive experimental study indicates that the theoretical worst-case complexity of the algorithms give little information about their behavior in practice. Careful implementation improves average computation times not only significantly, but asymptotically. Running times for our implementations are within a small constant factor from that of Prim’s algorithm for the Minimum Spanning Tree problem, suggesting that there is little room for improvement.

1 Introduction

The Steiner Problem in Graphs (SPG) can be stated as follows: given an undirected graph $G = (V, E)$, a set of terminals $T \subseteq V$, and a positive cost function $c(v, w)$ for all $(v, w) \in E$, find a minimum weight connected subgraph of $G$ containing all vertices in $T$. This is one of the most studied NP-hard problems. A wealth of links to recent papers on this subject can be found in [5].

In this paper, we are especially interested in constructive heuristics, algorithms that build feasible solutions from scratch. They can be used as stand-alone algorithms, providing solutions of reasonable quality within a short amount of time. However, their application is much broader. They are often used as subroutines of more elaborate primal heuristics [2, 10], dual heuristics [8], and exact algorithms [4, 8]. Regardless of the application, constructive heuristics should be as fast as possible, while preserving solution quality.

A number of such heuristics are described in the literature (see [3, 12] for surveys). We focus our attention on heuristics that are direct extensions of exact algorithms for the Minimum Spanning Tree (MST) problem. The main contribution of this paper are new, fast implementations for these methods. This includes not only versions of the traditional methods based on Prim’s and Kruskal’s algorithms, but also a Borůvka-based heuristic, a natural extension that has not

$^*$ Current address: Department of Computer Science, Princeton University, Princeton, NJ, 08544.
been previously described in the literature. (Descriptions of the original MST algorithms can be found in [1].) Extensive computational experiments suggest that these implementations allow the heuristics to run within a constant factor of “pure” MST algorithms on average, even though they have higher theoretical worst-case complexities. We show that this is not the case when some of the previous implementations are used.

The remainder of this paper is organized as follows. Section 2 presents the Distance Network Heuristic with its implementations and variations, alongside with a description of the Voronoi diagram for graphs, a data structure that is central in this text. Implementations of heuristics based on Borůvka’s, Kruskal’s, and Prim’s algorithms are described in Sections 3, 4 and 5, respectively. An empirical comparative analysis is presented in Section 6. Final remarks are made in Section 7.

2 Distance Network Heuristic (DNH)

Given a weighted graph $G = (V, E)$, we define the corresponding distance network $D_G = (V, E')$ as the graph containing all vertices in $V$ and such that for all pairs $(v, w) \in V$, there is an edge in $D_G$ with cost equal to the length of the shortest path between $v$ and $w$ in $G$. The Distance Network Heuristic (DNH) consists of the following three steps:

(a) find the MST of the subgraph of $D_G$ induced by $T$, then expand its edges (with the associated shortest paths) to create a valid Steiner tree of $G$;
(b) find the MST of the subgraph of $G$ induced by the solution obtained in (a);
(c) eliminate all non-terminal vertices with degree one.

Since step (a) already generates a valid solution, steps (b) and (c) can be seen as a post-optimization procedure; we call it MST-prune, following [2]. Step (b) takes at most $O(|E| + |V| \log |V|)$ time\(^1\) and step (c) requires $O(|V|)$ time.

The best known implementation of step (a) is due to Melhorn [6]. Essential to his implementation of DNH — and to some of the implementations proposed in this paper for other heuristics — is the concept of Voronoi diagrams in graphs. Given a set $X \subseteq V$ of vertices, the Voronoi diagram is a partition of $V$ into Voronoi regions, one for each vertex in $X$. The Voronoi region of $x_i$ (denoted by $\text{vor}(x_i)$) is defined as the set of vertices in $V$ that are not closer to any other vertex in $X$. Ties are broken arbitrarily. If $v \in \text{vor}(x_i)$, $x_i$ is said to be the base of $v$, denoted by $x_i = \text{base}(v)$.

Let $p(\text{base}(v), v)$ denote the shortest path from $\text{base}(v)$ to $v$, let $d(v)$ be the length of this path, and let $\text{pred}(v)$ be the vertex that immediately precedes $v$ in the path. The Voronoi diagram with respect to $T$, defined by the values $\text{base}(v)$.

---

\(^1\) In the theoretical part of this paper, whenever we need a “pure” algorithm for calculating minimum spanning trees, we use Prim’s algorithm, implemented with Fibonacci heaps. We also assume this data structure is used in implementations of Dijkstra’s algorithm for shortest paths. See [1] for alternative implementations.
\(d(v)\) and \(\text{pred}(v)\) for all \(v \in V\), can be computed in \(O(|E| + |V| \log |V|)\) time with Dijkstra's algorithm (taking all terminal nodes as sources).

An edge \((v, w)\) will be considered a frontier edge if \(\text{base}(v) \neq \text{base}(w)\). Let \(G'\) be a graph with the same set of edges and vertices as \(G\), but edge costs given by cost function \(c'\), defined as follows. If \((v, w)\) is a frontier edge, then \(c'(v, w) = d(v) + c(v, w) + d(w)\), else \(c'(v, w) = 0\). Note that every frontier edge \((v, w)\) is uniquely associated by this cost function to a path between \(\text{base}(v)\) and \(\text{base}(w)\). Melhorn [6] shows that there exists an MST of \(D_G\) containing only paths associated to edges with positive cost (i.e., frontier edges) in the MST of \(G'\). There is no need to actually compute \(D_G\).

Therefore, step (a) of Melhorn's implementation of DNH is actually divided into two substeps: (i) calculating the Voronoi diagram of \(G\); and (ii) finding an MST of \(G'\) and extracting the corresponding \(|T| - 1\) paths in \(G\). We have seen that substep (i) requires \(O(|E| + |V| \log |V|)\) operations. We now consider different implementations of substep (ii).

**DNH-Prim.** If Prim's algorithm is used to determine the MST of \(G'\), as suggested by Melhorn, DNH will require \(O(|E| + |V| \log |V|)\) operations. We refer to this implementation as DNH-Prim.

**DNH-Borůvka.** Alternatively, a slightly modified version of Borůvka's algorithm (also known as Sollin's algorithm) can be used in substep (ii). The classic algorithm starts with \(|V|\) components, one associated to each vertex in \(V\). It proceeds by repeatedly performing a step in which every component is connected (by a single edge) to the closest neighboring component. Because the number of components is at least halved in each step, there will be only one component left after at most \(\log |V|\) steps. Each step can be performed in \(O(|E|)\) time, thus leading to a \(O(|E| \log |V|)\) algorithm.\(^2\)

Our "modified version" considers only the vertices in \(T\) as initial components. In each of the connecting steps, every component is linked to its closest counterpart by a path with one or more edges. Finding all links in a given step amounts to checking the nonzero-cost edges in \(G'\) and using the Voronoi diagram to determine the associated paths in \(G\). This can be done in \(O(|E|)\) time for all paths in a given step. The Steiner tree can therefore be found in \(O(|E| \log |T|)\) time, increasing the algorithm's complexity to \(O(|E| \log |T| + |V| \log |V|)\). We note that \(T\) may be a small set when compared to \(V\), so this alternative implementation, called DNH-Borůvka, is often faster than DNH-Prim.

**DNH².** A slight modification in DNH-Borůvka can improve average solution quality while preserving computational complexity. Whenever a non-terminal vertex \(v\) is added to the solution in a connecting step, we can set \(d(v)\) (its distance to the closest terminal) to zero. After all, \(v\) is now part of a component, as all terminals are, so it can be used to link different components. Since edge

\(^2\) This can actually be implemented in \(O(|E| \log \log |V|)\) time, but our algorithm for the SPG is based on the less efficient basic implementation.
costs in $G'$ are computed dynamically, some will be reduced by this operation. This allows for future connecting steps to take into account (to a certain extent) the fact that there is already a partial solution, leading to final solutions of better quality. We call DNH$z$ this modification of DNH-Borůvka.

3 Borůvka

This section describes a heuristic to the SPG based on Borůvka's algorithm for the MST problem. To the best of our knowledge, there has been no previous description of such heuristic in the literature. However, this is just a natural extension of an existing idea, adapting MST algorithms to solve the Steiner problem in graphs. When compared to Prim's and Kruskal's heuristics, it turns out that Borůvka's has smaller worst-case running times, but tends to produce solutions of slightly worse quality, as Section 6 shows.

The heuristic starts with the set $T$ of terminals as its initial set of components. Then, in each iteration every component in the partial solution is linked to the closest component by the shortest possible path. Every such link creates a new component, formed by the original vertices and those in the shortest path. After at most $\log |T|$ iterations, only one component (a Steiner tree) will be left.

To implement this algorithm efficiently, we keep a Voronoi diagram, updated before each step to include (as bases) new vertices added to the solution. With the diagram, we can find the closest counterpart of each component in $O(|E|)$ time (for all components) by checking all edges $(v, w)$ such that $\text{base}(v)$ and $\text{base}(w)$ belong to different components (this is a subset of the frontier edges).

Building the first Voronoi diagram requires a complete execution of Dijkstra's algorithm, having all terminal nodes as sources. In each of the following iterations, the Voronoi diagram is just updated: Dijkstra's algorithm is run having as sources just the vertices added to the solution in the previous iteration (with zero distance label, since they become potential bases). Because distance labels of previous diagrams are still valid upper bounds, updating can be significantly faster than building a diagram from scratch; they tend to be limited to a few local operations. In the worst case, however, the update will require $O(|E| + |V| \log |V|)$ operations. The overall complexity of Borůvka’s heuristic, therefore, is $O((\log |T|) (|E| + |V| \log |V|))$.

Note that Borůvka uses essentially the same strategy as DNH$z$ to improve solution quality, but takes it one step further. In DNH$z$, every vertex added to the solution has its distance label set to zero. This helps the algorithm find shortcuts — paths whose ends are not both terminals — between components in future iterations. In Borůvka’s heuristic, the entire Voronoi diagram is updated; not only do vertices already in the solution have their distances set to zero, but also their neighbors may have their distances reduced. Shortcuts are more likely to be found, thus increasing average solution quality. Although there is an extra $\log |T|$ factor in the algorithm’s worst-case performance, Section 6 shows that in practice Borůvka is only marginally slower than DNH-Borůvka or DNH$z$. This confirms that updating the Voronoi diagram is usually a “cheap” operation.
4 Kruskal

Kruskal’s algorithm for the MST problem can also be used as a starting point for a constructive heuristic for the SPG. The classic algorithm starts with a partial solution with all \(|V|\) vertices and no edges. Then, it adds edges to the solution sequentially, always selecting the shortest edge that does not create a cycle. The algorithm ends after \(|V| - 1\) edges are selected, when only one component will be left. Wang [13] suggested a Kruskal-based heuristic for the SPG. In his algorithm, the initial partial solution is made up only by the set of terminals \(T\). The algorithm then sequentially adds paths to the solution (including intermediate non-terminal vertices), always selecting the shortest path connecting different components. After \(|T| - 1\) such paths are added, a valid solution is found.

4.1 Basic Implementation (Kruskal-B)

Kruskal’s heuristic can be implemented as follows. Let \(\text{dist}\) be a structure that maps each connected component to its closest counterpart (and contains the distance between them). In the beginning, each component is made up by a single terminal. The initialization of \(\text{dist}\) requires \(O(|T|)\) calls to Dijkstra’s algorithm, each having one terminal as source and any other terminal as target. In each of the \(|T| - 1\) iterations that follow, we (i) scan \(\text{dist}\) to determine the closest pair of components; (ii) join these components by the shortest path between them; and (iii) update \(\text{dist}\) by running Dijkstra’s algorithm once, with all vertices in the newly created component as sources (all other terminals must be reached). The heuristic runs in \(O(|T|(|E| + |V| \log |V|))\) time, corresponding to \(O(|T|)\) executions of Dijkstra’s algorithm. We call this implementation Kruskal-B.

4.2 Improved Implementation (Kruskal-I)

We now propose an improved implementation, named Kruskal-I, that is much faster in practice. In each of its \(|T| - 1\) iterations, Kruskal-I uses the Voronoi diagram associated with the partial solution \(S\) to identify the closest pair of components. We use the fact that there must be a frontier edge in the diagram that represents the shortest path in \(G\) between the closest pair of components. This is analogous to the the property stating that the Delaunay triangulation of a set of points contains the line segment connecting the two closest points (see [9], for instance).

The set of candidate shortest paths is in fact even more limited than that. Among all frontier edges \((v, w)\) incident to \(v\) such that \(v < w\), we define \(\mathcal{E}(v)\) as the one whose associated path is the shortest (ties are broken arbitrarily). If there is no frontier edge \((v, w)\) with \(v < w\), we regard \(\mathcal{E}(v)\) as undefined. Clearly, if the path associated with \(\mathcal{E}(v)\) is not the shortest path in the graph, neither are the paths associated with the other frontier edges \((v, w')\), with \(v < w'\).\(^3\) This

\(^3\) By considering only edges in which \(v\) is the endpoint with the smallest label, we avoid associating the same edge to two different vertices.
definition ensures that there is always a vertex \( v \in V \) such that \( \mathcal{E}(v) \) represents the shortest path between the closest pair of components.

Kruskal-I maintains two data structures: the Voronoi diagram associated to the partial solution and a heap to select among candidate paths. It is actually a heap of vertices, since each vertex is mapped to a unique edge, which in turn is mapped to a path. The algorithm begins by calculating the Voronoi diagram associated with the initial solution, which contains all terminals and no edges. For each vertex \( v \) for which \( \mathcal{E}(v) \) is defined, we insert \( v \) into the heap with cost equal to that of its associated path (if \( \mathcal{E}(v) = (v, w) \), this cost is \( d(v) + c(v, w) + d(w) \)). The algorithm then starts adding paths to the solution. Each of its \( |T| - 1 \) iterations is divided into four steps:

1. **Determine the closest pair of components.** Remove the first element \( v \) from the heap, and let \( \mathcal{E}(v) = (v, w) \) be its associated edge. If \( \text{base}(v) \) and \( \text{base}(w) \) both belong to the same component, then \((v, w)\) is not actually a frontier edge (i.e., the value of \( \mathcal{E}(v) \) is outdated). In that case, search the neighborhood of \( v \) to determine the correct value of \( \mathcal{E}(v) \) and reinsert \( v \) into the heap (if \( \mathcal{E}(v) \) is not defined, disregard it). Proceed until a vertex associated with an actual frontier edge is removed. This vertex determines the closest pair of components (or, more accurately, it is associated with the frontier edge that represents the shortest path between the closest pair of components).

2. **Add to the partial solution the path joining the closest pair.** As in Borůvka’s heuristic, use the Voronoi diagram to obtain the path from the frontier edge.

3. **Update the Voronoi diagram.** Run Dijkstra’s algorithm having as sources all vertices newly inserted into the solution (also as in Borůvka’s heuristic).

4. **Update the heap of candidates.** Each vertex \( v \) in the heap is associated with an edge \( \mathcal{E}(v) = (v, v') \) and a value, the length of the path \( \mathcal{E}(v) \) represents, given by \( d(v) + c(v, v') + d(v') \). The information associated with \( v \) may become inaccurate (outdated) only if one or more of the following situations occur:
   (a) \( d(v) \) is reduced;
   (b) \( d(v'') \) is reduced, \( v'' \) being a neighbor of \( v \) (possibly \( v' \) itself);
   (c) \((v, v')\) ceases to be a frontier edge, i.e., \( \text{base}(v) \) and \( \text{base}(v') \) become part of the same component in the solution.

Note that the only vertices that may have their \( d(\cdot) \) values reduced are those reached during the update of the Voronoi diagram (step 3). Therefore, it suffices to examine the neighborhood of those vertices to update the values associated to them (case (a)) and to their neighbors (case (b)). Although case (c) may involve other vertices, there is no need to address this case directly, since it can only cause an increase in the value associated to \( v \). If we leave it unchanged, \( v \) will become the first element of the heap before it should. This is perfectly acceptable, as long as we are able to discover that \((v, v')\) is not actually a frontier edge, and this is precisely what step 1 above does. Note that cases (a) and (b) must be addressed as soon as they occur, since they potentially increase the priority of the vertices involved.

---

4 Step 4 explains how the value may become outdated.

5 There is also the possibility that \( v \) falls into cases (a) and (b) before it actually makes it to the top of the heap; it will be updated then.
Step 1 removes up to $O(|V|)$ elements from the heap, each in $O(\log |V|)$ time. Steps 1 and 4 may need to check the neighborhoods of all vertices, which requires $O(|E|)$ total time. Step 2 runs in time proportional to the number of vertices in the path ($O(|V|)$ in the worst case, considering the whole algorithm). Finally, updating the Voronoi diagram in step 3 requires $O(|E|+|V|\log |V|)$ time. Therefore, all $|T|-1$ iterations require $O(|T|(|E|+|V|\log |V|))$ time. This means Kruskal-I has the same worst-case complexity as Kruskal-B. However, Section 6 shows that Kruskal-I is much faster. This happens because the Voronoi diagram and the heap of candidates allow Kruskal-I to avoid useless, repeated operations. Updating these structures is on average much faster than the worst-case analysis suggests, since modifications tend to affect small portions of the graph.

5 Prim

Proposed by Takahashi and Matsuyama [11], the constructive heuristic for the SPG based on Prim’s algorithm for the MST problem is probably the most commonly used in the literature [2, 4, 8, 10]. The classic algorithm for the MST problem “grows” the solution from a root vertex. In each step, the vertex that is closer to the current partial solution is added to it, alongside with the connecting edge. After $|V|-1$ steps, all vertices are spanned. The corresponding heuristic for the SPG grows a tree starting from a vertex (usually a terminal). In each iteration, we add to the solution the closest terminal not yet inserted (alongside with the vertices in the path from the solution to that terminal). In $|T|-1$ iterations (or $|T|$, if the root is not a terminal), a valid Steiner tree is created.

5.1 Basic Implementation (Prim-B)

The basic implementation of the heuristic (which we call Prim-B) is discussed in [3, 8], among others. It divides the algorithm into two phases. The first phase consists of building a $|V| \times |T|$ table of distances, which describes the shortest paths between each vertex and every terminal. In can be computed in $O(|T||E|+|V|\log |V|)$ time, by executing the Dijkstra’s algorithm once from each terminal. We also maintain an auxiliary structure, called closest, which associates each terminal not yet inserted to the closest vertex in the partial solution. This structure can be initialized in $O(|T||V|)$ time: just scan the table of distances. The second phase is when the solution is actually built. In each of its $O(|T|)$ iterations, we scan closest to determine which terminal to add to the tree. The corresponding path (described in the table of distances) is then added to the solution. Finally, we update closest: for each vertex added to the solution, we check if its distance to any terminal not in $S$ is less than what closest reports.

The bottleneck of the algorithm is building the table of distances. In [8], Polzin and Daneshmand suggest a Voronoi-based acceleration scheme that tries to expedite the process by limiting the extent of each execution of Dijkstra’s algorithm. Our version of Prim-B tested in Section 6 includes such an acceleration. Although it does make the algorithm run faster for certain instances, it does not make it asymptotically more efficient on average.
5.2 Improved Implementation (Prim-I)

We now suggest an improved implementation (Prim-I) that works in a single phase. It can be seen as a modified version of Dijkstra's algorithm.

It starts with a partial solution $S$ containing just the root $r$. To each vertex $v_i$ we associate a variable $\pi_i$ representing an upper bound to the distance from $v_i$ to $S$ in $G$. Initially, $\pi_r = 0$ ($r$ is already in $S$) and $\pi_i = \infty$ for all vertices $v_i$ not in $S$. In each iteration, the algorithm determines the closest terminal $t$ and adds it to the partial solution, alongside with the non-terminals in the corresponding shortest path. The process of determining $t$ is similar to a heap-based implementation of Dijkstra's algorithm. Remove $v_\sigma$, the element with the highest priority (smallest $\pi_i$), from the heap. If $v_i$ is not a terminal or already belongs to the solution, then for each neighbor $v_j$ of $v_i$, check whether $\pi_j < \pi_i + c(v_i, v_j)$. If that is the case, set $\pi_j \leftarrow \pi_i + c(v_i, v_j)$ and update the heap accordingly. Keep removing elements from the heap until a terminal not yet spanned is found. When this happens, add the corresponding new path to $S$.

The distance from any given vertex to the partial solution can only decrease in time. Hence, upper bounds found in any iteration remain valid until the end of the algorithm. To guarantee that the best vertices will be selected, all we have to do is reinsert every vertex $v_i$ into the heap (with $\pi_i = 0$) as soon as it becomes part of the solution. This ensures that, in subsequent iterations, vertices that get closer to the solution will be visited if necessary.

The algorithm consists of $O(|T|)$ iterations ($|T|$ if the root is non-terminal, $|T|-1$ if it is not). In each of these iterations, as many as $O(|V|)$ vertices may be removed from the heap, which takes $O(|V| \log |V|)$ time in the worst case. Each of them may have its neighborhood searched, and the values associated with the neighbors may need to be updated; this can be done in $O(|E|)$ time per iteration. Hence, the overall worst-case running time of Prim-I is $O(|T|(|E| + |V| \log |V|))$.

Note that some vertices may be inserted several times into the heap. However, this happens only if this is really necessary, that is, if their distances to the partial solution actually decrease. In fact, the experimental results in Section 6 suggest that, on average, a vertex is visited very few times during the execution of the algorithm. To understand why this happens, first note that each vertex is visited at most once per iteration, which means that no vertex can be visited more than $O(|T|)$ times. Furthermore, as the number of terminals increases, the number of vertices visited in each iteration tends to decrease. After all, with more terminals, there are on average fewer edges between the partial solution and the closest (non-inserted) terminal. It turns out that in practice these two factors (number of iterations and time for each iteration) balance each other, thus making the running time virtually independent of $|T|$.

This explains why Prim-I is much faster than Prim-B on average, although both have the same worst-case complexity. In fact, relying on a table of distances severely limits the applicability of the basic implementation. Merely initializing the table may be (and often is) much more expensive than running Prim-I in

---

6 It could be implemented without a heap, but it would not be as efficient in practice.
its entirety. But the main weakness of Prim-B is its quadratic memory usage; in practice, it cannot handle instances that are easily processed by Prim-I.

6 Empirical Analysis

This section presents an empirical comparison of the implementations discussed in this paper, with an emphasis on running times. Although we do compare the heuristics in terms of solution quality, we do it superficially. The reader is referred to [12] for a more thorough analysis, including not only most of the constructive heuristics discussed here (in their basic implementations), but also several other methods not directly based on algorithms for MST problems.

Test Problems. Our goal is to assess how the algorithms behave “on average”. However, it is not clear how to do this strictly, since there are too many variables involved in a single instance of the problem (number of vertices, number of edges, graph topology, number of terminals, terminal distribution, and edge weights, to mention a few). We decided to use instances already available on the literature, assuming they represent a fair sample of what could be considered “typical” instances. The algorithms were tested on all 994 valid instances\(^7\) available at the SteinLib repository [5]. To simplify the analysis, we grouped the original 36 series into six classes, as shown in Table 1.\(^8\) These instances vary greatly in size. On average, they have 1,252 vertices, 9,800 edges, and 186 terminals; these dimensions reach values up to 38,418, 221,445, and 11,849, respectively.

Table 1. Instances available at the SteinLib

<table>
<thead>
<tr>
<th>class</th>
<th>series</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>b, c, d, e, mc, p6z</td>
<td>random graphs, random weights</td>
</tr>
<tr>
<td>lst</td>
<td>es*fst, tspfst</td>
<td>rectilinear graphs, L1 weights</td>
</tr>
<tr>
<td>vlsi</td>
<td>alue, alut, dmxa, dw, gap, lin, ms, taq</td>
<td>grid graphs with holes</td>
</tr>
<tr>
<td>incidence</td>
<td>i080, i160, i320, i640</td>
<td>random graphs, incidence weights</td>
</tr>
<tr>
<td>euclidean</td>
<td>x, pde</td>
<td>graphs with euclidean weights</td>
</tr>
<tr>
<td>hard</td>
<td>sp, puc</td>
<td>artificial, “hard” instances</td>
</tr>
</tbody>
</table>

Methodology. All algorithms were implemented in C++ and compiled under Linux with gcc 2.96 with the -04 flag (full optimization). CPU times were obtained on a 1.7 GHz Pentium 4 with 256 MB of RAM. Because the timing function (getrusage) has low precision (1/60th of a second), running times could not be measured directly. Instead, each algorithm was repeatedly run on each instance until five seconds have passed, and the average time was considered.

\(^7\) At the time of writing, September 1, 2001.

\(^8\) In the table, the notion of “incidence weights” by that edges incident to terminals have larger weights than those incident to non-terminals (see [4]).
All algorithms tested share the same implementation of basic data structures (graphs, heaps, stacks). We opted for using binary heaps to implement priority queues. Although they are asymptotically less efficient, their performance is competitive in practice with that of Fibonacci heaps [7]. With binary heaps, the worst cases of our algorithms are slightly different from those mentioned in previous sections. For DNH-Prim, DNH-Borůvka, and DNHζ, the worst case is $O(|E| \log |V|)$; for Borůvka, $O(|E| \log |T| \log |V|)$; and for all versions of Prim and Kruskal, $O(|T||E| \log |V|)$. Note that these algorithms differ only in the dependency on $|T|$.

Although described as part of DNH, MST-prune may improve any valid solution, as noted in [12]. Therefore, unless otherwise noted, running times for all heuristics include this post-optimization phase, and solution qualities refer to results obtained after its application. MST-prune was implemented with Prim’s algorithm (with binary heap), with roots selected at random. Its $O(|E| \log |V|)$ running time does not affect the worst-case complexities of the heuristics.

Comparing the performance of the heuristics with a well-known algorithm may convey a better assessment of the behavior of the methods discussed here. Therefore, our analysis includes an “extra” method, MST: it simply applies Prim’s algorithm to calculate the minimum spanning tree of the entire graph (treating terminals as ordinary vertices). Running times for this method do not include MST-prune.

Results. Table 2 shows the average running times of each algorithm for all classes of instances. Due to its quadratic memory usage, implementation Prim-B could not be tested on 11 instances. Some of the values in the tables below (marked with a “~” symbol) are therefore approximate, since they consider only the remaining instances.

<table>
<thead>
<tr>
<th>method</th>
<th>complexity</th>
<th>euclidean</th>
<th>fst</th>
<th>hard incidence</th>
<th>random</th>
<th>vlsi</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNH-Prim</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
<td>)$</td>
<td>2.71</td>
</tr>
<tr>
<td>DNH-Borůvka</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
<td>)$</td>
<td>3.27</td>
</tr>
<tr>
<td>DNHζ</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
<td>)$</td>
<td>3.30</td>
</tr>
<tr>
<td>Borůvka</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
<td>\log</td>
<td>T</td>
</tr>
<tr>
<td>Kruskal-B</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>Kruskal-I</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>Prim-B</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>Prim-I</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>MST</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
<td>)$</td>
<td>1.36</td>
</tr>
</tbody>
</table>

---

9 Namely: aue7065, aue7080, alut2625, es10000fst01, fli3795fst, fh4461fst, pch3038fst, pla7397fst, rl11849fst, rls915fst, and rls934fst (first three belong to vlsi, others to fst).
It should come as no surprise that DNH-Prim, DNH-Borůvka, and DNHz have average running times comparable to that of MST. They all have the same worst-case complexity, $O(|E| \log |V|)$, and share a very similar structure. Due to the extra $\log |T|$ factor, $O(|E| \log |V| \log |T|)$. Borůvka is a little slower, but still has comparable running times. The difference is much more evident for Kruskal-B and Prim-B, and can be explained by the $|T|$ factor in their complexity. However, the improved implementations of the same heuristics (Kruskal-I and Prim-I) also have the extra $|T|$ factor, but are only slightly slower than DNH-Prim.

But being fast on average is not always enough. Several instances of the SteinLib have hundreds or even thousands of terminals, and it could be the case that the improved implementations perform poorly for these (and that the average is small because of other instances, with few terminals). Table 3 shows that this is not the case. For each instance, we calculated the relative running time of each algorithm, that is, the ratio between its running time and that of MST (the minimum spanning tree of the entire graph). The table shows the best, average, and worst-case ratios obtained when all instances are considered.

<table>
<thead>
<tr>
<th>method</th>
<th>complexity</th>
<th>best</th>
<th>average</th>
<th>worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNH-Prim</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>DNH-Borůvka</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>DNHz</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>Borůvka</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
<tr>
<td>Kruskal-B</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Kruskal-I</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Prim-B</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>Prim-I</td>
<td>$O(</td>
<td>T</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>MST</td>
<td>$O(</td>
<td>E</td>
<td>\log</td>
<td>V</td>
</tr>
</tbody>
</table>

On average, none of the improved versions takes more than three times the time required to calculate a simple minimum spanning tree. In some (rare) cases, they can even be faster, because they do not necessarily have to examine the whole graph. But, most importantly, Table 3 reveals that the worst cases of both Prim-I and Kruskal-I are remarkably tolerable, especially when we consider that more naïve implementations of the same algorithms (Prim-B and Kruskal-B) are up to hundreds of times slower than MST. Prim-I and Kruskal-I are never more than ten times slower than MST. That is only twice as much as the fastest heuristic tested, DNH-Prim, which has the same worst-case complexity as a simple minimum spanning tree computation.

The results obtained so far strongly suggest that the performances of our implementations do not depend as much on the value of $|T|$ as their worst-case expressions indicate. In Figure 1, we show the relative running times of Prim-I (without MST-prune, in this case) with respect to MST for all 994 Steinlib
instances. If both algorithms behaved in average as their worst-case complexities suggest, the ratio should be linearly dependent on \(|T|\). In log scale, this would mean an exponentially-growing curve, but that is clearly not what we obtained.

![Graph](image)

**Fig. 1.** Relative times of Prim-I for all SteinLib instances (without MST-prune)

Once again, this confirms what was said in Section 5: the worst-case analysis is overly pessimistic when it assumes that each vertex is visited (i.e., removed from the heap) once per iteration, or \(\Theta(|T|)\) times during the whole algorithm. For this particular set of instances, we verified that each vertex is visited only 2.17 times on average (recall that the average number of terminals is 186). Even in the worst case (instance \(i640-345\), with 160 terminals), the average vertex was removed from the heap only 4.28 times.

Although all these experiments on SteinLib make it clear that Prim-I does not depend “too much” on \(|T|\), it is not obvious what the dependence is. Because the set of instances is heterogeneous, there are simply too many variables (number of edges, number of vertices, graph topology, and so on) determining the results.

Therefore, we devised a more controlled experiment, where \(|T|\) is the only relevant variable to be considered. While the results for this particular experiment cannot be directly extended to any arbitrary set of instances, we believe it does help understand the behavior of the algorithms. We generated a new series, \textbf{special}, which may be seen as an extension of \textbf{random}. This is a series of random graphs, with integer edge weights uniformly distributed in \([1; 10]\) and random terminal placement. All graphs have exactly 1024 vertices and 32768 edges, and number of terminals ranging from 2 to 1024. For each value of \(|T|\), there are 25
different graphs, generated with different random seeds. Figure 2 shows how the average running times of the algorithms depend on the size of $T$. In this particular test, all algorithms (including DNH) were run without MST-prune, thus allowing for a better assessment of the relative performance of each algorithm's "core".\textsuperscript{10}

![Graph showing running times for various algorithms.](image)

**Fig. 2.** Running times for series special (without MST-prune)

The figure shows that, for graphs in this series, only two methods (the basic implementations of Kruskal's and Prim's heuristics) actually depend linearly on $|T|$, as the fast-growing curves indicate (note that the graph is in log scale). In fact, Kruskal-B takes 2.76 seconds on average for $|T| = 1024$ and Prim-B takes 0.78 seconds for $|T| = 724$ (Prim-B is actually faster than that for $|T| = 1024$ — just 0.09 seconds — because of the acceleration suggested in [8]). Both times are much higher than those of all other algorithms, including the new implementations of Kruskal and Prim. No other method takes more than 0.01 second to find a solution, regardless of the number of terminals.

**Solution quality.** Even with their best implementations, Prim and Kruskal are still a little slower on average than DNH, and, although we have established that the extra $|T|$ factor in their complexities does not tend to be relevant in practice, it does exist. All other things being equal, it seems that it would be faster and safer to use DNH rather than other heuristics. But there is something that is

\textsuperscript{10} For any given $|T|$, the effect of including MST-prune would be a uniform increase in running times for all heuristics. This effect is more noticeable for large values of $|T|$, but the additional running times are never greater than that of running MST.
not equal: solutions provided by Prim’s or Kruskal’s heuristics are much better than those provided by DNH. Table 4 shows, for each class of instances, the average percentual error obtained by each algorithm (only one implementation of each algorithm is shown; others behave similarly to their counterparts). Errors were calculated with respect to the best solutions known on September 1, 2001 (available at the SteinLib).

Table 4. Average percentual error w.r.t. best known solution (with MST-prune)

<table>
<thead>
<tr>
<th>method</th>
<th>euclidean</th>
<th>fast</th>
<th>hard incidence</th>
<th>random</th>
<th>vlsi</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNH-Borůvka</td>
<td>2.53</td>
<td>2.70</td>
<td>24.23</td>
<td>23.31</td>
<td>5.03</td>
</tr>
<tr>
<td>DNHz</td>
<td>2.34</td>
<td>2.23</td>
<td>22.20</td>
<td>21.49</td>
<td>4.09</td>
</tr>
<tr>
<td>Borůvka</td>
<td>2.35</td>
<td>1.90</td>
<td>21.18</td>
<td>20.29</td>
<td>3.56</td>
</tr>
<tr>
<td>Kruskal-I</td>
<td>1.65</td>
<td>1.86</td>
<td>9.55</td>
<td>18.36</td>
<td>2.68</td>
</tr>
<tr>
<td>Prim-I</td>
<td><strong>1.28</strong></td>
<td><strong>1.83</strong></td>
<td><strong>7.87</strong></td>
<td><strong>17.92</strong></td>
<td>2.82</td>
</tr>
</tbody>
</table>

The performance of each algorithm varies significantly among series. Within each series, however, the table clearly shows that the more actively an algorithm looks for “shortcuts” between terminals, the better it tends to behave with respect to other methods. DNH calculates the Voronoi diagram and determines all connections at once; it does not actively use previously inserted non-terminals to try to reduce the cost of the final solution. DNHz does that in a limited fashion, by favoring edges that are incident to vertices in the partial solution. Borůvka takes this strategy one step further: after each iteration, the whole Voronoi diagram is updated, so future links can take further advantage of the new non-terminal vertices in the solution. Kruskal also updates the Voronoi diagram after each iteration, but recall that an iteration in this algorithm corresponds to the addition of a single path, as opposed to multiple paths in Borůvka. This means that each decision made by Kruskal (and Prim) can take into account completely updated information about the partial solution. In general, this leads to solutions that are closer to the optimum.

7 Final Remarks

We have proposed new implementations of two important constructive heuristics for the Steiner Problem in Graphs: those based on Prim’s and Kruskal’s algorithms for the MST problem. Empirical results attest that these implementations are significantly faster and more robust than the usual, basic ones. We have also suggested some minor variations of Melhorn’s implementation of DNH heuristic, shown to be competitive in practice. A Borůvka-based heuristic, which can be seen as an intermediate method between DNH and Kruskal (in terms of both solution quality and worst-case complexity), was also proposed.

From a theoretical point of view, an interesting extension of this work would be a strict average case analysis of the running times of the algorithms. How-
ever, this seems to be a non-trivial task, since there are too many variables to
color: [V], [E], [T], edge weights, graph topology, terminal distribution, etc.
Alternatively, one could identify specific classes of instances for which the new
implementations (especially Kruskal-I and Prim-I) are guaranteed to perform
fewer operations than their worst-case expressions suggest.

In terms of applications, it would be interesting to know to what extent
these new implementations of constructive heuristics can help accelerate more
elaborate exact and approximate algorithms for the SPG, as those described in
[2, 4, 8]. In [10], the fast implementation of Prim’s has already been used and
greatly contributed to keep running times within acceptable limits.

Acknowledgements

We would like to thank Eduardo Uchoa, Diego Nehab, and two anonymous
referees for their helpful comments. MPA was partially supported by CNPq,
process 300475/93-4. RFW was partially supported by CAPES.

References

1. R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, algorithms, and
2. C. Duin and S. Voss. The Pilot method: A strategy for heuristic repetition with
3. F. Hwang, D. Richards, and P. Winter. *The Steiner tree problem*, volume 53 of
5. T. Koch, A. Martin, and S. Voss. SteinLib: An updated library on Steiner tree
7. B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for
   constructing a minimum spanning tree. *DIMACS Monographs in Discrete Mathe-
8. T. Polzin and S. V. Daneshmand. Improved algorithms for the Steiner problem in
    In H. Noltemeier, editor, *Proceedings of the International Workshop on Graph-