Highly Scalable Algorithms for Rectilinear and Octilinear Steiner Trees*

Andrew B. Kahng   Ion I. Măndoiu   Alexander Z. Zelikovsky†

Departments of CSE and ECE, University of California at San Diego, La Jolla, California 92093-0114, USA
†Department of Computer Science, Georgia State University, University Plaza, Atlanta, Georgia 30303, USA
E-mail: {abk,mandoiu}@cs.ucsd.edu, alexz@cs.gsu.edu

Abstract—The rectilinear Steiner minimum tree (RSMT) problem, which asks for a minimum-length interconnection of a given set of terminals in the rectilinear plane, is one of the fundamental problems in electronic design automation. Recently there has been renewed interest in this problem due to the need for highly scalable algorithms able to handle nets with tens of thousands of terminals. In this paper we give a practical $\mathbf{O(n\log^2 n)}$ heuristic for computing near-optimal rectilinear Steiner trees based on a batched version of the greedy triple contraction algorithm of Zelikovsky [21]. Experiments conducted on both random and industry testcases show that our heuristic matches or exceeds the quality of best known RSMT heuristics, e.g., on random instances with more than 100 terminals our heuristic improves over the rectilinear minimum spanning tree by an average of 11%. Moreover, our heuristic has very well scaling runtime, e.g., it can route a 34k-terminals net extracted from a real design in less than 25 seconds compared to over 86 minutes needed by the $\mathbf{O(n^2)}$ edge-based heuristic of Borah, Owens, and Irwin [3]. Since our heuristic is graph-based, it can be easily modified to handle practical considerations such as routing obstacles, preferred directions, via costs, and octilinear routing – indeed, experimental results show only a small factor increase in runtime when switching from rectilinear to octilinear routing.

I. INTRODUCTION

The *rectilinear Steiner minimum tree* (RSMT) problem, which asks for a minimum-length interconnection of a given net (i.e., set of terminals) in the rectilinear plane, is one of the fundamental problems in electronic design automation. Although deep-submicron technology has introduced additional routing objectives, minimum length continues to be the primary objective when routing non-critical nets, since the minimum-length interconnection has minimum total capacitance and occupies minimum amount of area.

Of growing interest are practical methods for minimum-length rectilinear routing of nets with up to *tens of thousands of terminals*. Nets of this size, e.g., scan enable, are becoming more common in modern designs due to the increased emphasis on design for test [1]. Such nets are non-critical and tend to consume significant routing resources, so minimizing length is the appropriate optimization objective. Furthermore, very large RSMT instances are created by reductions that model non-zero terminal dimensions, e.g., nets with pre-routes. High-quality routing of such instances requires representing each terminal by a set of electrically equivalent points [15] and this results in RSMT instances with as much as 100,000 points [22].

The main contribution of this paper is a highly scalable heuristic for computing rectilinear Steiner trees. Existing implementations of exact methods [17] and of best-performing heuristics such as Iterated 1-Steiner [10] and Rajagopalan-Vazirani [12] cannot be used on instances with tens of thousands of terminals due to combinatorial explosion and quadratic memory requirements, respectively. Our heuristic requires $O(n)$ memory and $O(n\log^2 n)$ time for n terminals, and routes, e.g., a 34k-terminals net extracted from a real design in less than 25 seconds compared to over 86 minutes needed by the $O(n^2)$ edge-based heuristic of [3]. More importantly, this dramatic reduction in runtime is achieved with no loss in solution quality. On random instances with more than 100 terminals our algorithm improves over the rectilinear minimum spanning tree (MST) by an average of 11%, matching in solution quality the edge-based heuristic of [3]. To the best of our knowledge, this is the first practical sub-quadratic RSMT heuristic with such performance.[1]

Due to considerable potential for further wirelength reductions, recently there has been much interest [11, 16, 18] in *octilinear routing*, which allows 45° diagonal interconnect in addition to the traditional horizontal and vertical orientations.[2] Since our heuristic is graph-based, it can be easily modified to handle octilinear routing – indeed, the results reported in Section V show only a small factor increase in runtime compared

*Work partially supported by Cadence Design Systems, Inc., the MARCO Gigascale Silicon Research Center, NSF Grant CCR-9988331, Award No. MM2-3018 of the Moldovan Research and Development Association (MRDA) and the U.S. Civilian Research & Development Foundation for the Independent States of the Former Soviet Union (CRDF), and by the State of Georgia's Yamacraw Initiative.

[1] The $O(n\log n)$ implementation given in [3] for the edge-based heuristic requires advanced data structures, potentially involving large hidden constants. We are not aware of any implementation demonstrating the practical applicability of this implementation.

[2] The octilinear distance between points (x,y) and (x',y') is equal to $\max\{|x-x'|,|y-y'|\}+(\sqrt{2}-1)\min\{|x-x'|,|y-y'|\}$ and is always smaller than the rectilinear distance, $|x-x'|+|y-y'|$, unless the two points are on the same horizontal or vertical line, in which case the two distances are equal.

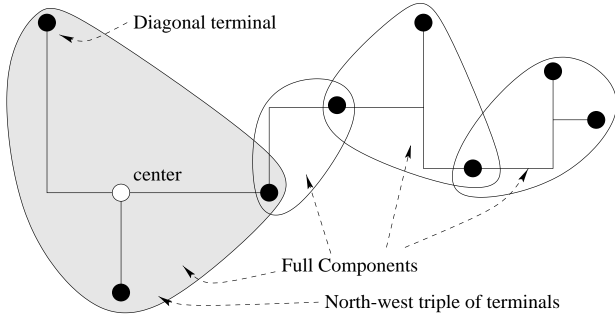

Fig. 1. A 3-restricted rectilinear Steiner tree partitioned into full components. The dark full component is a north-west triple.

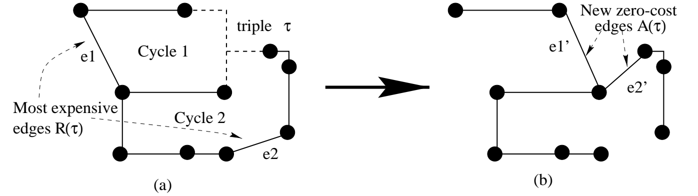

Fig. 2. The MST of G_S (a) before and (b) after contraction of the triple τ. The gain of the dashed triple is the difference between the cost of most expensive edges $R(\tau) = \{e_1, e_2\}$ in each of the two cycles of $T \cup \tau$ and the cost of τ. Two new zero-cost edges $A(\tau) = \{e'_1, e'_2\}$ replace e_1 and e_2 in the updated MST.

to the rectilinear implementation. The heuristic can be similarly extended to handle other practical considerations such as routing obstacles, preferred directions [19], and via costs.

Our highly scalable Steiner tree heuristic, referred to as the *batched greedy algorithm* (BGA) in the following, derives its efficiency from three key ideas:

- Combining the implementation proposed in [5] for the *greedy triple contraction algorithm* (GTCA) of Zelikovsky [21] with the batched method introduced by the Iterated 1-Steiner heuristic of Kahng and Robins [10].
- A new divide-and-conquer method for computing a superset of size $O(n \log n)$ of the set of $O(n)$ triples required by GTCA.
- A new linear size data structure that enables finding a bottleneck (i.e., maximum cost) edge on the tree path between two given nodes in $O(\log n)$ time after $O(n \log n)$ preprocessing, with very low constants hidden under the big O. This data structure allows computing the gain of a triple (see Section II for the definition) in $O(\log n)$ time, leading to an $O(n \log^2 n)$ implementation of BGA.[3]

The rest of the paper is organized as follows. In Section II we briefly review the greedy triple contraction algorithm of [21] and describe our new batched greedy algorithm. In the following two sections we describe in detail two of the key BGA subroutines: in Section III we give the new divide-and-conquer method for computing the set of $O(n \log n)$ triples used by BGA, while in Section IV we describe the new data structure for computing bottleneck edges. Finally, in Section V we give experimental results comparing BGA with previous implementations of rectilinear and octilinear Steiner tree heuristics and exact algorithms on test cases both randomly generated and extracted from recent VLSI designs.

[3]Our data structure may be of interest in other applications that require computing bottleneck edges. For example, Zachariasen incorporated it in the beta version of the GeoSteiner 4.0 code for computing optimum geometric Steiner trees. On large instances, computing bottleneck edges with the new data structure was found to be faster, most likely due to improved memory access locality, than look-up in a precomputed matrix [20].

II. The Greedy Triple Contraction and Batched Greedy Algorithms

We begin this section by introducing the Steiner tree terminology used in the rest of the paper. A Steiner tree for a set of terminals is a tree spanning the terminals and possibly additional points, called *Steiner points*. A Steiner tree is called a *full Steiner tree* if all terminals are leaves (i.e., have degree 1). Any Steiner tree T can be split into edge-disjoint full Steiner trees called the *full Steiner components* of T [6]. A Steiner tree T is called k-restricted if every full component of T has at most k terminals (an example of a 3-restricted rectilinear Steiner tree is shown in Figure 1). The minimum-cost 3-restricted Steiner tree is in general cheaper than the minimum spanning tree (MST) of the terminals (note that the MST is the minimum-cost 2-restricted Steiner tree of the terminals).

The *greedy triple contraction algorithm* (GTCA) in [21] finds an approximate minimum-cost 3-restricted Steiner tree by greedily choosing 3-restricted full components which reduce the cost of the MST. In order to describe GTCA we need to introduce a few more notations. Let G_S be the complete graph on a given set S of terminals, and let $MST(S)$ be the MST of G_S. A *triple* τ is an optimal Steiner tree for a set of three terminals.[4] We denote by $center(\tau)$ the single Steiner point of τ and by $cost(\tau)$ the cost of τ. In the graph $MST(S) \cup \tau$, there are two cycles (see Figure 2(a)). To obtain an MST of this graph we should remove the most expensive edge from each cycle. Let e_1 and e_2 be the two edges that must be removed and let $R(\tau) = \{e_1, e_2\}$. The *gain* of τ is $gain(\tau) = cost(R(\tau)) - cost(\tau)$.

GTCA (see Figure 3) repeatedly adds a triple τ with the largest gain and *contracts* it, i.e., collapses the three terminals of τ into a single new terminal. Contraction of a triple is conveniently implemented by adding two new zero-cost edges $A(\tau) = \{e'_1, e'_2\}$ between the three terminals of τ (see Figure 2(b)). It is easy to see that addition of $A(\tau)$ changes the MST of G_S – in the updated MST the two edges in $A(\tau)$ replace the two edges in $R(\tau)$. Finally, GTCA adds all chosen triples to the original MST of G_S and outputs the MST of this union.

Theorem 1 [2] *The cost of the rectilinear Steiner tree constructed by GTCA is at most 1.3125 times more than the opti-*

[4]The optimum Steiner tree of 3 given terminals can be computed in constant time under the common geometric metrics, including rectilinear [8] and octilinear [11] metrics.

Input: Set S of terminals
Output: 3-restricted Steiner tree T spanning S

---

1. $T \leftarrow MST(G_S)$
2. $F \leftarrow \emptyset$
3. Repeat forever
  Find a triple τ with maximum gain
  If $gain(\tau) \leq 0$, then go to Step 4
  $F \leftarrow F \cup \{\tau\}$ // Add τ
  $T \leftarrow T - R(\tau) + A(\tau)$ // Contract τ
4. Output $MST(F \cup MST(G_S))$

Fig. 3. The greedy triple contraction algorithm.

mal Steiner tree.

Fößmeier, Kaufmann and Zelikovsky [5] prove that in order to achieve an approximation ratio of 1.3125 in Theorem 1 it is sufficient to consider only *empty tree triples* of terminals. A triple τ is *empty* if the minimum rectangle bounding the triple does not contain any other terminals and is a *tree* triple if the center c of τ is adjacent to all terminals of the triple in $MST(S + c)$ (or, equivalently, if $gain(\tau) > 0$). As shown in [5], there are at most $36n$ empty tree triples. Even so, finding the best triple in Step 3 of GTCA is very time consuming. An efficient $O(n^2 \log n)$ time implementation of GTCA should maintain dynamic minimum spanning trees for which, to date, there is no data structure able to handle instances with tens of thousands of nodes in practical running time. Existing data structures are difficult to implement and involve big asymptotic constants, see Cattaneo et al. [4] for a recent empirical study.

Our new heuristic, the *batched greedy algorithm* (BGA) (see Figure 4) adopts the batched method from [10], substantially reducing running time by relaxing the greedy rule used to select triples in GTCA. After contracting a triple we continue by picking the best triple among those with unchanged gain; in general this may not be the best triple overall. Note that a triple τ can change its gain only if one of the edges in $R(\tau)$ is removed when contracting other triple – if none of the contracted triples removes edges from $R(\tau)$ then the gain of τ is unchanged. When done with one such *batched phase* (the body of the while loop in Step 4) it is still possible to have positive gain triples. Therefore, we recompute triple gains and repeat the batched phase selection until no positive gain triples are left. To enable further improvements, we add the centers of triples selected in Step 4 to the terminal set then iterate Steps 2–5 (which constitute a *round* of the algorithm) until no more centers are added to the tree.

In next section we show how to compute in $O(n \log n)$ time a set of $O(n \log n)$ triples containing all empty tree triples (see Theorem 3). Then, in Section IV we describe a data structure which enables computing a bottleneck edge on the tree path between any two given nodes in $O(\log n)$ time after $O(n \log n)$ time preprocessing. Since computing the gain of a triple amounts to 3 bottleneck edge computations, this leads

Input: Set S of terminals
Output: Steiner tree T spanning S

---

1. Compute the minimum spanning tree of S, $MST(S)$
2. Compute a set $Triples$, of size $O(n \log n)$, containing all empty tree triples
3. $SP \leftarrow \emptyset$
4. While $Triples \neq \emptyset$ do
  For each $\tau \in Triples$ compute $R(\tau)$, $A(\tau)$, and $gain(\tau)$, discarding triples with non-positive gain
  Sort $Triples$ in descending order of gain
  Unmark all edges of $MST(S)$
  For each $\tau \in Triples$ do
    If both edges in $R(\tau)$ are unmarked, then mark them and replace them in the MST with the two edges in $A(\tau)$, i.e.,
    $MST(S) \leftarrow MST(S) - R(\tau) + A(\tau)$
    $SP \leftarrow SP + center(\tau)$
5. If $SP = \emptyset$ then return the minimum spanning tree of S, else
  $S \leftarrow S + SP$
  Compute the minimum spanning tree of S and discard all Steiner points with degree 1 or 2
  Go to Step 2

Fig. 4. The batched greedy algorithm.

to an $O(n \log^2 n)$ time implementation of the batched phase algorithm. This gives the following:

Theorem 2 *The running time of the batched greedy algorithm is $O(Pn \log^2 n)$, where P is the total number of batched phases and n is the number of terminals.*

In practice the total number of phases P is small and can be bounded by a constant. Thus, the runtime of BGA is $O(n \log^2 n)$.

III. Generation of Triples

In this section we show how to compute in $O(n \log n)$ time a set of $O(n \log n)$ triples containing all empty tree triples. For simplicity we assume that terminals are in general position, i.e., no two of them share the same x- or y-coordinate. This assumption is not restrictive since we can always break ties, e.g., according to terminal IDs.

In a triple, the terminal which does not share x- and y-coordinates with the center is called *diagonal*. There are 4 types of triples depending on where the diagonal terminal lies with respect to the center: a triple is called *north-west* if the diagonal terminal is in the north-west quadrant of the center (see Figure 1); north-east, south-west, and south-east triples are defined similarly. We will use the divide and conquer method to find $O(n \log n)$ north-west triples containing all north-west empty tree triples. Triples of the other types are obtained by reflection and application of the same algorithm.

For finding north-west triples we recursively partition the terminals into (almost) equal halves with a bisector line parallel to line $y = -x$. Let LB (left-bottom) and TR (top-right) be

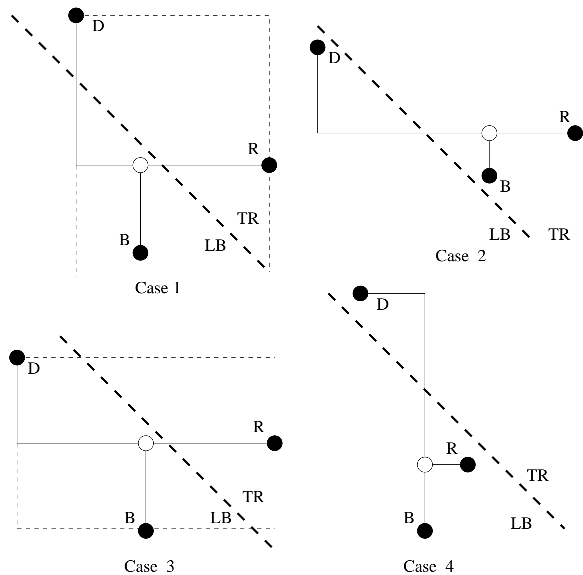

Fig. 5. Four cases of partitioning of a north-west triple.

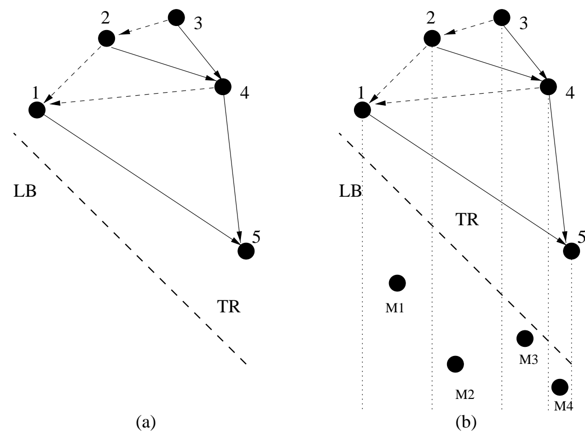

Fig. 6. Case 1: (a) Finding the right terminal for each diagonal terminal, e.g., if $D = 1$, then $R = 5$, if $D = 3$, then $R = 4$, etc. (b) Finding highest terminals in the strips corresponding to consecutive terminals.

the half-planes defined by the bisector line, and let D, R, and B be the diagonal, right, and bottom terminals of a north-west triple that is intersected by the bisector line (see Figure 5). We distinguish the following 4 cases:

Case 1: $D, R \in TR$ **and** $B \in LB$. Figure 6 (a) illustrates how to compute for each diagonal terminal D the unique terminal R that can serve as a right terminal in an empty north-west triple with D as the diagonal terminal. All terminals in TR are processed in x-ascending order as follows: (1) if the next terminal has y larger then the current terminal, then a dashed pointer is set from the next to the current terminal, and then the current terminal is advanced to the next terminal; (2) otherwise, a solid pointer is set from the current terminal to the next one, and the current terminal is moved back along the dashed pointer (if it exists, otherwise the current terminal is advanced to the next). Clearly this procedure is linear since the runtime is proportional to the number of pointers established and each terminal has at most two pointers (one solid and one dashed). When processing of the points in TR is finished, each solid arc connects a terminal D with the leftmost terminal in TR lower than and to the right of D, i.e., with the unique terminal R that can serve as a right terminal in an empty north-west triple with D as the diagonal terminal.

In order to find all Case 1 north-west triples, we must find for each solid arc (D, R) in TR the node B in LB which can complete the triple, i.e., the node B with maximum y-coordinate in the vertical strip defined by D and R. This is done in linear time by one traversal of the terminals in LB in x-ascending order (i.e., strip by strip) while computing the highest point in each strip.

Case 2: $B, R \in TR$ **and** $D \in LB$. For each terminal R, the unique terminal in TR that can serve as the bottom terminal in an empty north-west triple with R as the right terminal (i.e., the highest terminal in TR lower and to the left of R) can be found by a procedure similar to the one in Step 1. Cf. [5], an arc (R, B) in TR is completed into a tree north-west triple only when the diagonal node D is the closest to R (and therefore to B) in the octant of LB containing points higher than R. To find the diagonal points D for each arc (R, B) in TR, we simultaneously traverse terminals in TR in y-ascending order and terminals in LB in $(x - y)$-ascending order as follows: While there are unprocessed terminals in both TR and LB, do

- Advance in TR until we reach a terminal R which has an arc to the associated B
- Advance in LB until we reach a terminal D higher than R
- Assign D to R

Note that triples found bye the above procedure are not necessarily empty. With a more careful implementation it is possible to avoid generating non-empty triples, however this would not change the asymptotic number of triples generated or the worst-case running time of the algorithm.

Case 3: $R \in TR$ **and** $D, B \in LB$. It is equivalent to the case 1 after reflection over bisector.

Case 4: $D \in TR$ **and** $B, R \in LB$. It is equivalent to the case 2 after reflection over bisector.

Theorem 3 *A set of size $O(n \log n)$ containing all empty tree triples can be computed in $O(n \log n)$ time.*

Proof. Each north-west empty tree triple crossing the dividing diagonal must fall in one of the 4 cases considered and finding all crossing triples takes linear time. Thus, the running time is given by the recurrence $T(n) = 2T(n/2) + O(n)$, i.e., $T(n) = O(n \log n)$. The number of triples generated by the divide-and-conquer algorithm is also $O(n \log n)$ by the same recurrence; notice that each recursive step generates a linear number of triples. The same argument applies to the other 3 triple types. □

Input: Weighted tree $T = (V, E, cost)$ with $V = \{1, 2, \ldots, n\}$
Output: Arrays $parent(i)$ and $edge(i)$, $i = 1, \ldots, 2n-1$

---

1. Sort tree edges $e_1, \ldots, e_{n-1}$ in ascending order of cost
2. Initialization:

$next \leftarrow n$
For each $i = 1, 2, \ldots, 2n-1$ do
  $parent(i) \leftarrow NIL$
  $edge(i) \leftarrow NIL$

3. For each edge $e_i = (u, v)$, $i = 1, \ldots, n-1$, do

While $u \neq v$ and $parent(u) \neq NIL$ and
 $parent(v) \neq NIL$ do
  $u \leftarrow parent(u)$
  $v \leftarrow parent(v)$
If $parent(u) = parent(v) = NIL$, then
  $next \leftarrow next + 1$
  $parent(u) \leftarrow parent(v) \leftarrow next$
  $edge(u) \leftarrow edge(v) \leftarrow i$
If $parent(u) = NIL$ and $parent(v) \neq NIL$, then
  $parent(u) \leftarrow parent(v)$
  $edge(u) \leftarrow i$
If $parent(u) \neq NIL$ and $parent(v) = NIL$, then
  $parent(v) \leftarrow parent(u)$
  $edge(v) \leftarrow i$

4. Output the arrays $parent(i)$ and $edge(i)$

Fig. 7. The hierarchical greedy preprocessing algorithm.

IV. Computing maximum cost edge on a tree path

It is easy to see that computing the gain of a triple τ and the edges in $R(\tau)$ reduces to finding bottleneck (i.e., most expensive) edges on the tree paths between pairs of terminals in τ. The *hierarchical greedy preprocessing* (HGP) algorithm given in Figure 7 computes for a given tree on n terminals two auxiliary arrays, $parent$ and $edge$, with at most $2n-1$ elements each. Using these arrays, the bottleneck tree edge between any two terminals u and v can be found in $O(\log n)$ using the algorithm in Figure 8.

Assuming that edges are sorted in ascending order of cost, HGP is equivalent to the following recursive procedure. First, for each node u, direct the cheapest edge incident to u, away from u, and save its index in $edge(u)$. As a result some edges remain undirected, some become unidirected, and some become bidirected. In the subgraph induced by the (uni- and bi-) directed edges, each connected component consists of a bidirected edge with two (possibly empty) arborescences attached to its ends. HGP collapses each such connected component K into a single node q, then sets $parent(u)$ to q for every $u \in K$. Since each connected component contains at least one bidirected edge, no more than $n/2$ collapsed component nodes are created. The procedure is repeated on the tree induced by collapsed components until there is a single node left. The total runtime of HGP is $O(n \log n)$ because of the edge sorting in Step 1, remaining HGP steps require $O(n)$ time.

Clearly, edge costs decrease along any directed path of a connected component K. Therefore, if u and v are two vertices of K, then the index of the maximum cost edge on the tree path between u and v is $\max\{edge(u), edge(v)\}$. If u and v are in different components K and K', we need to compute the maximum between $edge(u)$, $edge(v)$, and the maximum index of the most expensive edge on the path between K and K' in the tree T with collapsed connected components. The algorithm in Figure 8 is an iterative implementation of this recursive definition. Since the hierarchy of collapsed connected components has a depth of at most $\log n$, we get:

Input: Tree edges $e_1, \ldots, e_{n-1}$ in ascending order of cost, arrays $parent(i)$ and $edge(i)$, $i = 1, \ldots, 2n-1$, and nodes $u, v \in V$
Output: Maximum cost edge on the tree path between u and v

---

1. $index \leftarrow -\infty$,
2. While $u \neq v$ do
  $index \leftarrow \max\{index, edge(u), edge(v)\}$
  $u \leftarrow parent(u)$
  $v \leftarrow parent(v)$
3. Return e_{index}

Fig. 8. Subroutine for computing the maximum cost edge on the tree path between nodes u and v.

Theorem 4 *The algorithm in Figure 8 finds the maximum cost edge on the tree path connecting two given nodes in $O(\log n)$ time after $O(n \log n)$ time for hierarchical greedy preprocessing.*

V. Experimental Results

Comprehensive experimental evaluation indicates that the Iterated 1-Steiner heuristic of Kahng and Robins [10] significantly outperforms in solution quality the RSMT heuristics proposed prior to 1992 [9]. Since then, the edge-based heuristic of Borah, Owens, and Irwin [3], and the IRV heuristic [12] have been reported to match or slightly exceed Iterated 1-Steiner in solution quality. However, among these best-performing heuristics only the edge-based heuristic can be applied to instances with tens of thousands of terminals, since current implementations of Iterated 1-Steiner and IRV require quadratic memory. Besides Borah's $O(n^2)$ implementation of the edge-based heuristic, we compared our $O(n \log^2 n)$ batched greedy algorithm to the recent $O(n \log^2 n)$ Prim-based heuristic of Rohe [14]. For comparison purposes, we also include results from our implementation of the Guibas-Stolfi $O(n \log n)$ rectilinear MST algorithm [7], and, whenever possible, the optimum RSMTs computed using the beta version of the GeoSteiner 4.0 algorithm recently announced in [13].

All heuristics and MST algorithms were run on a dual 1.4 GHz Pentium III Xeon server with 2GB of memory running Red Hat Linux 7.1. The GeoSteiner code using the CPLEX 6.6 linear programming solver was run on a 360 MHz SUN Ultra 60 workstation with 2 GB of memory under SunOS 5.7. The test bed for our experiments consisted of two categories

of instances: instances drawn uniformly at random from a $1,000,000 \times 1,000,000$ grid, ranging in size between 100 and 500,000 terminals, and a set of 8 testcases extracted from recent industrial designs, ranging in size between 330 and 34,000 terminals.

Table I gives the percent improvement over the rectilinear MST and running time (in CPU seconds) for experiments on rectilinear instances. On random instances, the batched greedy heuristic matches or slightly exceeds in average solution quality the edge-based heuristic of [3]. Both batched greedy and the edge-based heuristic improve the rectilinear MST by an average of 11% in our experiments. This is roughly 1% more than the average improvement achieved by the Prim-based heuristic of [14], and is within 0.7% of the optimum average improvement for the sizes for which the optimum could be computed using GeoSteiner. Results on VLSI instances show that the relative performance of the heuristics is the same to that observed on random instances. However, the improvement over the rectilinear MST and the gaps between heuristics are smaller in this case.

The results in Table I show that the batched greedy algorithm is highly scalable. Even though batched greedy is not as fast as the MST or the Prim-based heuristic of [14], it can easily handle up to hundreds of thousands of terminals in minutes of CPU time. Compared to Borah's $O(n^2)$ implementation of the edge-based heuristic, batched greedy is two or more orders of magnitude faster as soon as the number of terminals gets into the tens of thousands.

The batched greedy algorithm can be easily adapted to other cost metrics, such as octilinear routing. The only required modifications are in the distance formula (see Footnote 2) and in the procedure for finding the optimum Steiner point of a triple. Table II gives results obtained by the octilinear versions of the Guibas-Stolfi MST, $O(n^2)$ edge-based,[5] batched greedy, and GeoSteiner 4.0 algorithms. Octilinear batched greedy is almost always better than the octilinear edge-based heuristic, and very close to optimum for the instances for which the latter is available. Furthermore, octilinear batched greedy remains highly scalable, with just a small factor increase in runtime compared to the rectilinear version.

VI. Conclusions

Non-critical nets with tens of thousands of terminals are becoming more common in modern designs due to the increased emphasis on design for test. Even a single net of this size can render quadratic Steiner tree algorithms impractical, given the stringent constraints on routing runtime (e.g., designers expect full chip global and detailed routing to be completed overnight). In this paper we have given a high-quality $O(n \log^2 n)$ heuristic that can practically handle these nets without compromising solution quality.

Since our heuristic is graph-based, it can be easily modified to handle other cost metrics, e.g., octilinear routing. We are currently extending the heuristic to handle other practical considerations, such as routing obstacles, preferred directions, and via costs.

[5] We use our own octilinear modification of Borah's code since the implementation in [11] appears to have cubic rather than quadratic runtime.

Acknowledgments

We would like to thank Manjit Borah, Benny Nielsen, André Rohe, and Martin Zachariasen for giving us access to their implementations, and Charles Alpert for providing the VLSI benchmarks.

References

[1] C. Alpert. Personal communication.

[2] P. Berman, U. Fossmeier, M. Kaufmann, M. Karpinski and A. Zelikovsky. Approaching the 5/4-approximation for rectilinear Steiner trees, in K. W. Ng et al. (eds.), *Proc. European Symp. on Algorithms (ESA)*, Springer Verlag Lecture Notes in Computer Science 762, 1994, pp. 533–542.

[3] M. Borah, R.M. Owens, and M.J. Irwin. A fast and simple Steiner routing heuristic, *Discrete Applied Mathematics* 90 (1999), pp. 51–67

[4] G. Cattaneo, P. Faruolo, U.F. Petrillo, and G.F. Italiano. Maintaining dynamic minimum spanning trees: an experimental study, in D.M. Mount, C. Stein (eds.), *Proc. 4th Int. Workshop on Algorithm Engineering and Experiments (ALENEX)*, Springer Verlag Lecture Notes in Computer Science 2409, 2002, pp. 111–125.

[5] U. Fößmeier, M. Kaufmann and A. Zelikovsky. Faster approximation algorithms for the rectilinear Steiner tree problem, *Discrete & Computational Geometry* 18 (1997), pp. 93–109.

[6] E.N. Gilbert, H.O. Pollak. Steiner minimal trees, *SIAM J. Appl. Math.* 32 (1977) pp. 826–834.

[7] L.J. Guibas and J. Stolfi. On computing all north-east nearest neighbors in the L_1 metric *Information Processing Letters* 17 (1983), pp. 219–223.

[8] M. Hanan. On Steiner's problem with rectilinear distance, *SIAM Journal on Applied Mathematics* 14 (1966), 255–265.

[9] F.K. Hwang and D.S. Richards and P. Winter. *The Steiner tree problem*. North-Holland, Annals of Discrete Mathematics 53, 1992.

[10] A.B. Kahng and G. Robins. A new class of iterative Steiner tree heuristics with good performance, *IEEE Trans. on CAD* 11 (1992), pp. 1462–1465.

[11] C.-K. Koh and P.H. Madden, Manhattan or Non-Manhattan? A study of alternative VLSI routing architectures, in *Proc. Great Lakes Symposium on VLSI*, 2000, pp. 47–52.

[12] I.I. Măndoiu, V.V. Vazirani, and J.L. Ganley. A new heuristic for rectilinear Steiner trees. *IEEE Transactions on Computer-Aided Design* 19 (2000) pp. 1129–1139.

[13] B.K. Nielsen, P. Winter, and M. Zachariasen. An exact algorithm for the uniformly-oriented Steiner tree problem, in R. Möhring and R. Raman (eds.), *Proc. European Symp. on Algorithms (ESA)*, Springer Verlag Lecture Notes in Computer Science 2461, 2002, pp. 760–772.

[14] A. Rohe. Sequential and parallel algorithms for local routing, Ph.D. Thesis, Bonn University, Bonn, Germany, December 2001.

[15] L. Scheffer. Rectilinear MST code, available as part of the RMST-Pack at *http://www.gigascale.org/bookshelf/slots/RSMT/RMST/*.

[16] S. Teig, The X architecture: Not your father's diagonal wiring, inn *Proc. ACM/IEEE Workshop on System Level Interconnect Prediction (SLIP)*, 2002, pp. 33–37.

[17] D.M. Warme, P. Winter, and M. Zacharisen. GeoSteiner 3.1 package, available from `http://www.diku.dk/geosteiner/`

[18] `http://www.xinitiative.org`

[19] M.C. Yildiz and P.H. Madden. Preferred direction Steiner trees, in *Proc. Great Lakes Symposium on VLSI*, 2001, pp. 56–61.

[20] M. Zachariasen. Personal communication, August 2002.

[21] A. Zelikovsky. An 11/6-approximation for the network Steiner tree problem, *Algorithmica* 9 (1993), pp. 463–470.

[22] H. Zhou, N. Shenoy, and W. Nicholls. Efficient minimum spanning tree construction without Delaunay triangulation, in *Proc. Asia-Pacific Design Automation Conference (ASP-DAC)*, 2001, pp. 192–197.

TABLE I

PERCENT IMPROVEMENT OVER MST AND CPU TIME OF THE COMPARED RECTILINEAR STEINER TREE ALGORITHMS

#Term.	MST		Prim-Based		Edge-Based		Batched Greedy		GeoSteiner 4.0	
	Len.(μm)	CPU	%Imp.	CPU	%Imp.	CPU	%Imp.	CPU	%Imp.	CPU
Random instances (average results over 10 instances)										
100	85169.9	0.0005	9.78	0.001	10.97	0.006	10.99	0.003	11.66	0.555
500	184209.7	0.0036	10.08	0.007	11.12	0.216	11.17	0.081	11.76	15.205
1000	258926.8	0.0079	10.04	0.014	10.96	0.939	10.99	0.230	11.61	117.916
5000	573178.8	0.0501	10.02	0.082	11.02	56.348	11.05	1.903	—	—
10000	809343.5	0.1268	10.04	0.191	11.01	415.483	11.05	5.192	—	—
50000	1808302.7	1.2330	10.05	1.320	11.01	16943.777	11.06	69.043	—	—
100000	2555821.9	3.1150	10.08	3.143	11.04	61771.928	11.08	195.589	—	—
500000	5710906.8	22.9130	10.07	20.570	—	—	11.08	1706.765	—	—
VLSI instances										
337	247.7	0.0020	5.96	0.000	6.50	0.060	6.43	0.040	6.75	16.040
830	675.6	0.0055	3.10	0.010	3.19	0.320	3.20	0.080	3.26	9.480
1944	452.2	0.0165	6.86	0.040	7.77	3.640	7.85	0.400	8.15	1304.270
2437	578.8	0.0217	7.09	0.040	7.96	5.740	7.96	0.680	8.34	13425.310
2676	887.2	0.0235	8.07	0.040	8.99	5.340	8.93	0.770	9.38	430.800
12052	2652.7	0.1378	7.65	0.180	8.46	540.840	8.45	5.230	—	—
22373	13962.5	0.3419	8.99	0.480	9.83	2263.760	9.85	13.060	—	—
34728	9900.5	0.5455	8.16	0.690	9.01	5163.060	9.05	24.200	—	—

TABLE II

PERCENT IMPROVEMENT OVER MST AND CPU TIME OF THE COMPARED OCTILINEAR STEINER TREE ALGORITHMS

#Term.	MST		Edge-Based		Batched Greedy		GeoSteiner 4.0	
	Len.(μm)	CPU	%Imp.	CPU	%Imp.	CPU	%Imp.	CPU
Random instances (average results over 10 instances)								
100	72375.1	0.0005	4.28	0.530	4.43	0.010	4.75	11.608
500	155611.7	0.0036	4.12	13.410	4.29	0.118	4.60	311.991
1000	219030.8	0.0079	4.12	54.641	4.25	0.296	4.59	1321.382
5000	484650.5	0.0506	4.17	1466.296	4.31	2.820	—	—
10000	684409.5	0.1217	4.13	5946.815	4.28	8.362	—	—
50000	1528687.2	1.1940	4.16	147210.395	4.30	116.419	—	—
100000	2160629.4	3.1060	—	—	4.32	476.307	—	—
500000	4826839.1	23.0610	—	—	4.31	6578.840	—	—
VLSI instances								
337	219.0	0.0020	2.92	5.690	2.99	0.050	3.13	72.960
830	630.4	0.0055	0.93	27.610	0.90	0.120	1.07	195.190
1944	407.2	0.0167	3.33	202.030	3.47	0.750	4.01	5279.870
2437	523.1	0.0218	3.67	345.330	3.77	0.820	4.29	7484.730
2676	780.2	0.0236	3.41	392.340	3.51	1.310	3.89	6080.050
12052	2372.3	0.1417	3.63	7517.680	3.72	10.800	—	—
22373	12069.8	0.3447	3.65	25410.340	3.74	21.380	—	—
34728	8724.9	0.5427	3.64	62971.090	3.74	25.160	—	—