Efficient Steiner Tree Construction Based on Spanning Graphs

Hai Zhou
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208

ABSTRACT

Steiner Minimal Tree (SMT) problem is a very important problem in VLSI CAD. Given n points on a plane, a Steiner minimal tree connects these points through some extra points (called Steiner points) to achieve a minimal total length. Even though there exist many heuristic algorithms for this problem, they have either poor performances or expensive running times. This paper records an implementation of an efficient Steiner minimal tree algorithm that has a worst case running time of $O(n \log n)$ and a similar performance as the Iterated 1-Steiner algorithm. The algorithm efficiently combines Borah et al.'s edge substitute concept with Zhou et al.'s spanning graph. Extensive experimental studies are conducted to compare it with other programs.

Categories & subject descriptors

B.7.2 Design Aids: Placement and routing

General terms

Algorithms, Design, Experimentation

Keywords

Steiner tree, Minimal spanning tree, Routing

1. INTRODUCTION

Steiner Minimal Tree (SMT) problem has wide applications in VLSI CAD. Given n points on a plane, a Steiner minimal tree connects these points through some extra points (called Steiner points) to achieve a minimal total length. A SMT is generally used in initial net topology creation for global router and incremental net tree topology creation in physical synthesis. Therefore, it is often used as an accurate estimation for congestion and wire length during floorplan and placement. Since it is a problem that will be computed hundreds of thousands times and many of them will have very large input sizes, the Steiner minimal tree problem definitely deserves good performances and highly efficient solutions.

Because of its importance, there is much previous work to solve the SMT problem. These algorithms can be grouped into two classes: exact algorithms and heuristic algorithms. Since SMT is NP-hard, any exact algorithm is expected to have an exponential worst-case running time. However, two prominent achievements must be noted in this direction. One is the `GeoSteiner` algorithm and implementation by Warme, Winter, and Zacharisen [13, 12], which is the current fastest exact solution to the problem. The other is a Polynomial Time Approximation Scheme (PTAS) by Arora [1], which is mainly of theoretical importance. Since exact algorithms have long running time, especially on large input sizes, much more previous efforts were put on heuristic algorithms. Many of them generate a Steiner tree by improving on a minimal spanning tree topology [6], since it was proved that a minimal spanning tree is a 3/2 approximation of a SMT [7]. However, since the backbones are restricted to the minimal spanning tree topology in these approaches, there is a reported limit on the improvement ratios over the minimal spanning trees. The iterated 1-Steiner algorithm by Kahng and Robins [9] is an early approach to deviate from that restriction and an improved implementation [4] is a champion among such programs in public domain. However, the implementation in [9] has a running time of $O(n^4 \log n)$ and the implementation in [4] has a running time of $O(n^3)$. A much more efficient approach was later proposed by Borah et al. [2]. In their approach, a spanning tree is iteratively improved by connecting a point to an edge and deleting the longest edge on the created circuit. Their algorithm and implementation had a worst-case running time of $\Theta(n^2)$, even though an alternative $O(n \log n)$ implementation was also proposed. Since the backbone is no longer restricted to the minimal spanning tree topology, its performance was reported to be similar to the iterated 1-Steiner algorithm [2]. A recent effort in this direction is a new heuristic by Mandoiu et al. [10] which is based on a 3/2 approximation algorithm of the metric Steiner tree problem on quasi-bipartite graphs [11]. It performs slightly better than the iterated 1-Steiner algorithm, but its running times is also slightly longer than the iterated 1-Steiner algorithm (with the empty rectangle test [10] used).

Our objective in this paper is to implement a Steiner minimal tree heuristic that is much faster than the iterated 1-Steiner algorithm and has similar performance. To achieve that goal, we select the edge substitution approach of Bo-

ISPD'03, April 6–9, 2003, Monterey, California, USA.

rah et al. [2] as the basis, and enhance it with the spanning graph of Zhou et al. [15] and other improvements. The implemented algorithm runs in $O(n \log n)$ time and takes $O(n)$ storage, without large hidden constant. Another advantage of the algorithm is that it is easy to be implemented. Extensive experimental studies are conducted to compare it with other public available SMT programs and its advantages are demonstrated. Even though we only focus on rectilinear Steiner minimal tree (where distances are measured with rectilinear metric), the same idea can also be used in Euclidean Steiner tree.

The rest of the paper is organized as follows. In Section 2, the edge substitution approach of Borah et al. [2] and the spanning graph of Zhou et al. [15] will be reviewed as the basis of our program. Then in Section 3, we will show how the spanning graph can be used both to generate the initial spanning tree and to find the point-edge pairs for edge substitution. Section 4 will discuss a problem in Borah et al.'s approach and prove the correctness of our algorithm. Section 5 will give experimental results.

2. BACKGROUNDS

2.1 Borah et al.'s edge substitution approach

As illustrated by the example in Figure 1, Borah et al.'s algorithm [2] for the rectilinear Steiner minimal tree works as follows. It starts with a minimal spanning tree and then iteratively considers connecting a point (for example p in Figure 1) to a nearby edge (for example (a, b)) and deleting the longest edge ((b, c)) on the circuit thus formed.

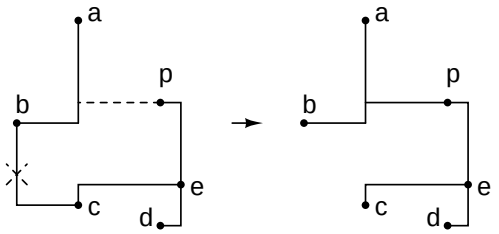

Figure 1: Edge substitution by Borah et al.

A straight-forward implementation by Borah et al. [2] used Prim's algorithm [3] to generate the initial minimal spanning tree in $O(n^2)$ time, and considered all possible point-edge pairs in the given tree, whose number is n^2. To find the longest edge on the circuit formed by connecting each point-edge pair, a depth-first search was conducted starting from every edge. The longest edge on the path from the starting edge to the current point is thus the longest edge on the circuit formed by connecting the current point to the starting edge. Since each depth-first search from one edge takes $\Theta(n)$ time, the total time for all edges is $\Theta(n^2)$. To keep the running time within $O(n^2)$, only one point-edge pair with the maximal gain was kept for each edge, and the $O(n)$ pairs were sorted according to non-increasing gains. Each point-edge pair was then connected (with proper deletion of the longest edge in the circuit) if the two involved edges had not been changed. As will be discussed in Section 4, when there are edges of equal length, errors may exist in the algorithm.

Besides the above implementation, they also discussed a possible way to make an algorithm of $O(n \log n)$ running time. This was based on an observation that not every point needs to be considered with every edge. For example, in Figure 1, point d does not need to be considered with edge (a, b) since they are blocked by edge (c, e). Using the geometrical blockage information, a point needs only to be considered with visible edges from its position. The number of point-edge pairs is thus reduced to $O(n)$. Unfortunately, this requires a geometrical sweepline algorithm to generate visible point-edge pairs. Tarjan's off-line least common ancestor algorithm [3] must be used to compute the longest edges on created circuits by point-edge connections. To keep the initial minimal spanning tree generation within $O(n \log n)$ time, Hwang's algorithm for minimal spanning tree construction [8] was suggested. As we can see, since each stage of this suggested approach involves a separate algorithm and some of them are very complicated, it is much more complicated than the straight-forward implementation and was never implemented.

2.2 Zhou et al.'s spanning graph

Zhou et al. [15] introduced the spanning graph as an intermediate step in minimal spanning tree construction. Given a set of points on the plane, a spanning graph is a graph over the points that contains a minimal spanning tree. The number of edges in the graph is called the cardinality of the graph and they presented an efficient $O(n \log n)$ algorithm to construct a spanning graph of cardinality $O(n)$.

From each point p, the plane can be divided into eight octal regions as shown in Figure 2. It can be proved that if rectilinear distance (that is, the distance between two points (x_1, y_1) and (x_2, y_2) is given by $|x_1 - x_2| + |y_1 - y_2|$) is used then the distance between any two points in one region is always smaller than the maximal distance from them to p. Based on the cycle property of a minimal spanning tree, that is, the longest edge on any circuit should not be included in any minimal spanning tree, this means that only the closest point to p in each region needs to be connected to p. Considering all given points, the connections will form a spanning graph of cardinality $O(n)$. Similar idea was developed by Yao [14] and further improved by Guibas and Stolfi [5].

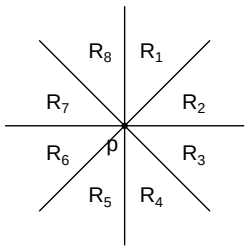

Figure 2: Point p needs to be connected to at most one point in each region.

A sweepline algorithm was designed by Zhou et al. [15] to efficiently construct the above spanning graph. It has a worst-case running time of $O(n \log n)$ and is much simpler than the divide-and-conquer algorithm of Guibas and Stolfi [5]. Its pseudo-code is presented in Figure 3 and works as follows. In order to find the closest points in regions R_1 and R_2 for all points, the points will be sorted in non-decreasing order of $x + y$. In this way, after each point p is swept, the first point seen in its R_1 or R_2 region will be the

```
Algorithm Rectilinear Spanning Graph (RSG)
  for (i = 0; i < 2; i++) {
    if (i == 0) sort points according to x + y;
    else sort points according to x − y;
    A[1] = A[2] = ∅;
    for each point p in the order {
      find points in A[1], A[2] such that p is in their
        R_{2i+1} and R_{2i+2} regions, respectively;
      connect p with points in each subset;
      delete the subsets from A[1], A[2], respectively;
      add p to A[1], A[2];
    }
  }
```

Figure 3: The rectilinear spanning graph algorithm

closest point in that region. To keep the swept points waiting for closest points in R_1 and R_2 regions, two active sets A_1 and A_2 are used. When a new point is swept, we need to search A_1 to find points with the new point in their R_1 region, and to search A_2 to find points with the new point in their R_2 region. Then edges are added from the new point to these points. After the found points are deleted from A_1 and A_2, the new point is added to them, since now the new point is swept and is waiting for the closest points. The connections for all points to their closest points in R_3 and R_4 regions are computed in the exact same fashion, except that now points are swept in non-decreasing order of $x-y$. There is no need to consider connections in regions R_5 through R_8 since they have been implied by connections in regions R_1 through R_4.

To achieve $O(n \log n)$ running time, the active sets A_1 and A_2 must be efficiently maintained so that searching, deletion, and insertion each can be done in $O(\log n)$ time. It can be shown that when an active set is used for region R_i, it cannot have two points such that one is in the other's R_i region. This property guarantees that the points in each active set be linearly ordered. Therefore, a balanced search tree could be used to efficiently implement an active set.

3. STEINER TREE ALGORITHM BASED ON SPANNING GRAPH

Since the edge substitution of Borah et al. [2] is a simple yet effective approach for Steiner minimal tree construction, our algorithm is based on it. Compared with Hwang's $O(n \log n)$ time algorithm [8] they suggested for the initial minimal spanning tree, Zhou et al.'s minimal spanning tree algorithm [15] based on spanning graph is much more efficient and simpler to implement. Furthermore, in their suggestion, a computational geometry algorithm needs to be used to generated the visible relations between points and edges in the tree. However, we find that if a spanning graph is generated, then the geometrical proximity information between points and edges is already embedded in the spanning graph. Therefore, no sweepline routine is needed to compute the edge blockage for point-edge pair generation if the spanning graph is leveraged. This makes the spanning graph a backbone of the whole algorithm: it is first used to generate the initial minimal spanning tree, and then to generate point-edge pairs for tree improvements. This kind of unification happens also in the spanning tree computation and the longest edge computation for each point-edge pair: using Kruskal's algorithm with disjoint set operations (instead of Prim's algorithm) [3] will unifies these two computations.

In order to reduce the number of point-edge pair candidates from $O(n^2)$ to $O(n)$, Borah et al. suggested to use the visibility of a point from an edge, that is, only a point visible from an edge can be considered to connect to that edge. But this requires a sweepline algorithm to find visibility relations between points and edges. A crucial observation is that if a point is visible to an edge then the point is usually connected to at least one end point of the edge in the spanning graph. An illustrating example is shown in Figure 4. Therefore, in our algorithm, the spanning graph is used to generate point-edge pair candidates. For each edge in the current tree, all points that are neighbors of either of the end points will be considered to form point-edge pairs with the edge. Since the cardinality of the spanning graph is $O(n)$, the number of possible point-edge pairs generated in this way is also $O(n)$.

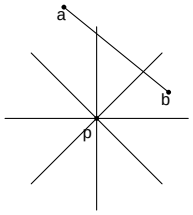

Figure 4: A point visible to an edge is usually connected to one end point in the spanning graph.

When connecting a point to an edge, the longest edge on the formed circuit needs to be deleted. In order to find the corresponding longest edge for each point-edge pair efficiently, we should look at how the spanning tree is formed through Kruskal's algorithm. This algorithm first sorts the edges into non-decreasing lengths and each edge is considered in turn. If the end points of the edge have been connected, then the edge will be excluded from the spanning tree, otherwise, it will be included. The structure of these connecting operations can be represented by a binary tree, where the leaves represent the points and the internal nodes represent the edges. When an edge is included in the spanning tree, a node is created for the edge and has as its two children the trees representing the two components con-

nected by this edge. To illustrate this, a spanning tree with its representing binary tree are shown in Figure 5. As we can see, the longest edge between two points is the least common ancestor of the two points in the binary tree. For example, the longest edge between p and b in Figure 5 is (b, c), which is the least common ancestor of p and b in the binary tree. To find the longest edge on the circuit formed by connecting a point to an edge, we need to find the longest edge between the point and one end point of the edge that are in the same component before connecting the edge. For example, consider the pair p and (a, b), since p and b are in the same component before connecting (a, b), the edge needs to be deleted is the longest between p and b.

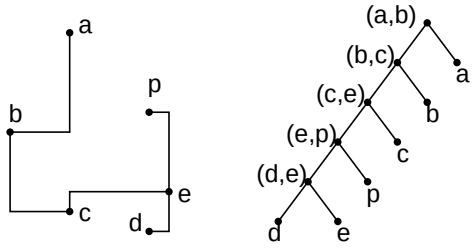

Figure 5: A minimal spanning tree and its merging binary tree.

Based on the above discussion, the pseudo-code of the algorithm can be described in Figure 6. At the beginning of the algorithm, Zhou et al.'s rectilinear spanning graph algorithm [15] is used to generate the spanning graph G for the given set of points. Then Kruskal's algorithm is used on the graph to generate a minimal spanning tree. The data structure of disjoint sets [3] is used to merge components and check whether two points are in the same component (the first **for** loop). During this process, the merging binary tree and the queries for least common ancestors of all point-edge pairs are also generated. Here s, $s1$, and $s2$ represent disjoint sets and each keeps record the root of the component in the merging binary tree. For each edge (u, v) adding to T, each neighbor w of either u or v will be considered to connect to (u, v). The longest edge for this pair is the least common ancestor of w, u or w, v depending on which point is in the same component as w. The procedure `lca_add_query` is used to add this query. Connecting the two components by (u, v) will also be recorded in the merging binary tree by the procedure `lca_tree_edge`. After generating the minimal spanning tree, we also have the corresponding merging binary tree and the least common ancestor queries ready. Using Tarjan's off-line least common ancestor algorithm [3] (represented by `lca_answer_queries`), we can generate all longest edges for the pairs. With the longest edge for each point-edge pair, the gain of connecting the point to the edge can be calculated. Then each of the point to edge connections will be realized in a non-increasing order of their gains. A connection can only be realized if both the connection edge and deletion edge have not been deleted yet.

The running time of the algorithm is dominated by the spanning graph generation and edge sorting, which take $O(n \log n)$ time. Since the number of edges in the spanning graph is $O(n)$, both Kruskal's algorithm and Tarjan's off-line least common ancestor algorithm take $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function, which grows extremely slow.

4. DISCUSSIONS

As pointed out by one referee, the straight-forward implementation by Borah et al.–as described in [2]–was not totally correct. The problem came from the way they found the longest edge for a point-edge pair. To see how that happened, consider an example shown in Figure 7. As stated in Section 2.1, from each edge in the tree, a depth-first search will be used to find the longest edge on the path to any point. Thus, when starting from edge (a, b) in Figure 7, the longest edge to c could be (d, e) or (f, g). Suppose we pick the closest to the point, it is (f, g). Then when we start from edge (a', b'), similarly, the longest edge to c' could be (d, e) or (f, g). Using the same criteria will give us (d, e). Therefore, when c is connected to (a, b), edge (f, g) will be deleted. Then when c' is connected to (a', b'), edge (d, e) will be deleted. This will give a dangling edge (e, f) and a loop from a to a'.

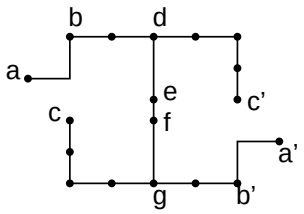

Figure 7: A problem in the straight-forward implementation of Borah et al.

However, we can prove that this problem does not exist in our algorithm.

THEOREM 1. *The* `RST` *algorithm only generates a Steiner tree on a given set of points.*

PROOF. We fulfill the proof by showing the following invariant in the last **for** loop in the algorithm:

> For any point-edge pair $(p, (a, b), (c, d))$, if neither (a, b) nor (c, d) is touched (i.e. deleted), then there must be a path between p and (a, b) that goes through (c, d).

This invariant is true at the beginning of the **for** loop. Suppose it is violated, it must be violated by committing a pair (x, e_1, e_2). And the only possible way is that the deletion edge e_2 is on the path between p and (a, b) and is not (a, b) or (c, d). We claim that the circuit formed by connecting x to e_1 cannot include (c, d). Otherwise, both e_2 and (c, d) are on the path from x to e_1. Since we know (c, d) is an ancestor of e_2 in the binary tree, there is no way for e_2 to be the least common ancestor. Based on the claim, committing (x, e_1, e_2) only changes a part of the path between p and (a, b) that does not include (c, d), Therefore, after the merging, the path between p and (a, b) still includes (c, d).

Based on the above invariant, the tree property will be kept by each edge substitution. □

5. EXPERIMENTAL RESULTS

We implemented the Rectilinear Steiner Tree (RST) algorithm in C language, following exactly the pseudo-code in

```
Algorithm Rectilinear Steiner Tree (RST)
   T = ∅;
   Generate the spanning graph G by RSG algorithm;
   for (each edge (u, v) ∈ G in non-decreasing length) {
      s1 = find_set(u); s2 = find_set(v);
      if (s1 != s2) {
         add (u, v) in tree T;
         for (each neighbor w of u, v in G)
            if (s1 == find_set(w))
                lca_add_query(w, u, (u, v));
            else lca_add_query(w, v, (u, v));
         lca_tree_edge((u, v), s1.edge);
         lca_tree_edge((u, v), s2.edge);
         s = union_set(s1, s2); s.edge = (u, v);
      }
   }
   generate point-edge pairs by lca_answer_queries;
   for (each pair (p, (a, b), (c, d)) in non-increasing positive gains)
      if ((a, b), (c, d) has not been deleted from T) {
         connect p to (a, b) by adding three edges to T;
         delete (a, b), (c, d) from T;
      }
```

Figure 6: The rectilinear Steiner tree algorithm.

Figure 6, with the exception that the program starting from the first **for** loop is repeated on the Steiner tree if there are improvements in the previous iteration.

We compared our program (denoted as `RST`) with other public available programs: the exact algorithm `GeoSteiner` (version 3.1) by Warme, Winter, and Zacharisen [12]; the Batched Iterated 1-Steiner (`BI1S`) by Robins; and the Borah et al.'s algorithm implemented by Madden (`BOI`)[1]. All the programs can be found at GSRC's Achievable Design Bookshelf. We plan to put our RST program there when it is ready.

In Table 1, we reported the comparisons between our program and the three programs: `GeoSteiner, BI1S`, and `BOI`. For fair comparisons, we compile and run all programs on the same machine–a Dell PowerEdge 1400SC running Linux operating system. For each input size ranging from 100 to 5000, 30 different test cases are randomly generated through the `rand_points` program in `GeoSteiner`. Each test case is run on the four programs and the improvement ratios of the Steiner tree (St) over the minimal spanning tree (MST)–that is, (MST-St)/MST)–are calculated. For each input size, we report the average improvement ratio (in percentage) and average running time (in seconds) on each of the programs. As we can see, `RST` always gives better improvements than `BOI` with less running times. This confirms our belief that there is no large hidden constant in the running time of `RST`. The fact that `RST` does not show $n \log n$ growth comes from the factor that it is repeated until there is no improvement.

For better understanding of the use of spanning graphs in Steiner tree construction, and the quality of Steiner trees generated by `RST`, we also plotted the spanning graph, the minimal spanning tree, and the Steiner tree, computed by `RST` for a randomly generated 500 points, in Figures 8, 9, and 10.

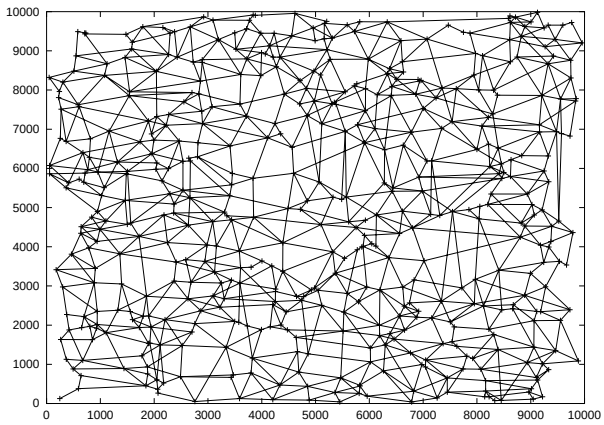

Figure 8: The spanning graph over 500 points

6. CONCLUSIONS

In summary, we developed an efficient Steiner tree algorithm which is based on Borah et al.'s edge substitution approach and Zhou et al.'s spanning graph algorithm. The implementation has a running time of $O(n \log n)$ and a storage requirement of $O(n)$, without large hidden constant. Experimental results showed the efficiency of this algorithm.

Acknowledgments

The author wants to thank Mr. Chuan Lin for help running the experiments.

[1]We did not include a comparison with `BOI` in the primary submission since `BOI` gave us wrong results–even the minimal spanning tree lengths were not correct. Only after digging into the code, we found that three integers were read for each point instead of two that is specified in the documents.

Table 1: Experimental Results

input size	GeoSteiner		BI1S		BOI		RST	
	improve	time	improve	time	improve	time	improve	time
100	11.440	0.4870	10.907	0.6325	9.300	0.0267	10.366	0.0143
200	11.492	3.5567	10.897	4.8103	9.192	0.1287	10.375	0.0697
300	11.492	12.6853	10.931	18.7704	9.253	0.2993	10.361	0.1703
500	11.525	72.1919	-	-	9.274	0.8770	10.363	0.4790
800	11.343	536.1733	-	-	9.284	2.3987	10.428	1.5207
1000	-	-	-	-	9.367	4.0843	10.508	2.9903
2000	-	-	-	-	9.326	31.0980	10.414	26.0257
3000	-	-	-	-	9.390	104.9190	10.567	79.8225
5000	-	-	-	-	9.356	307.9767	10.500	256.9443

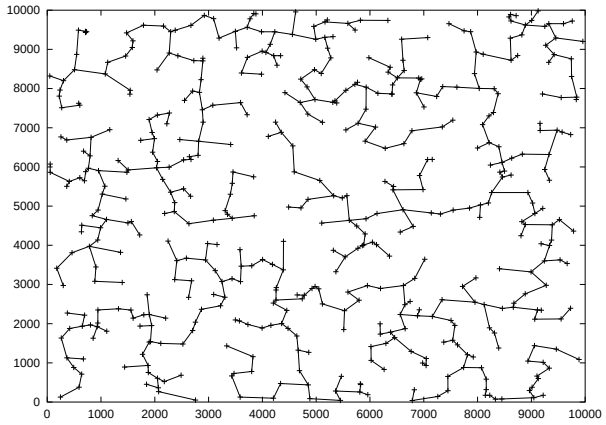

Figure 9: The minimal spanning tree over 500 points

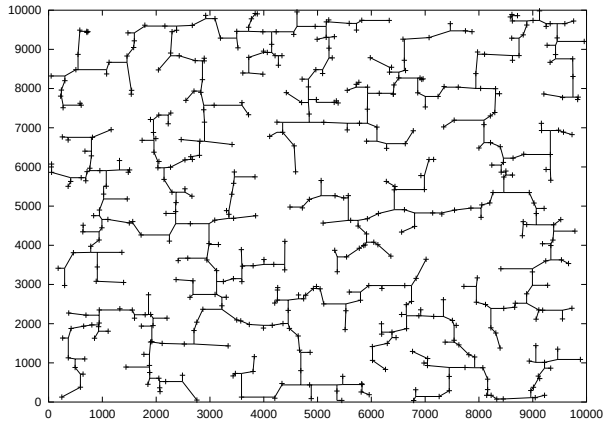

Figure 10: The Steiner tree by RST over 500 points

7. REFERENCES

[1] S. Arora. Polynomial-time approximation schemes for euclidean tsp and other geometric problem. *Journal of the ACM*, 45(5):753–782, 1998.

[2] M. Borah, R. M Owens, and M. J. Irwin. An edge-based heuristic for steiner routing. *IEEE Transactions on Computer Aided Design*, 13:1563–1568, 1994.

[3] T. H. Cormen, C. E. Leiserson, and R. H. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[4] J. Griffith, G. Robins, J. S. Salowe, and T. Zhang. Closing the gap: Near-optimal steiner trees in polynomial time. *IEEE Transactions on Computer Aided Design*, 13(11):1351–1365, November 1994.

[5] Leo J. Guibas and Jorge Stolfi. On computing all north-east nearest neighbors in the L_1 metric. *Information Processing Letters*, 17(4):219–223, 8 November 1983.

[6] J.-M. Ho, G. Vijayan, and C. K. Wong. New algorithms for the rectilinear steiner tree problem. *IEEE Transactions on Computer Aided Design*, 9:185–193, 1990.

[7] F. K. Hwang. On Steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics*, 30:104–114, 1976.

[8] F. K. Hwang. An $O(n \log n)$ algorithm for rectilinear minimal spanning trees. *Journal of the ACM*, 26(2):177–182, April 1979.

[9] A. B. Kahng and G. Robins. A new class of iterative steiner tree heuristics with good performance. *IEEE Transactions on Computer Aided Design*, 11(7):893–902, July 1992.

[10] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley. A new heuristic for rectilinear Steiner trees. In *Proc. Intl. Conf. on Computer-Aided Design*, 1999.

[11] S. Rajagopalan and V. V. Vazirani. On the bidirected cut relaxation for the metric Steiner tree problem. In *10th ACM-SIAM Symposium on Discrete Algrothms*, pages 742–751, 1999.

[12] D. M. Warme, P. Winter, and M. Zacharisen. `GeoSteiner 3.1 package`. available at `ftp.diku.dk/diku/users/martinz/geosteiner-3.1.tar.gz`.

[13] D. M. Warme, P. Winter, and M. Zacharisen. Exact algorithms for plane steiner tree problems: A computational study. Technical Report DIKU-TR-98/11, Dept. of Computer Science, University of Copenhagen, 1998.

[14] Andrew Chi-Chih Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, November 1982.

[15] H. Zhou, N. Shenoy, and W. Nicholls. Efficient spanning tree construction without delaney triangulation. *Information Processing Letter*, 81(5), 2002.