

# A Heuristic for Steiner Tree Construction

Jin Hu

Northwestern University  
Jin@northwestern.edu

**Abstract.** The problem of constructing an optimal Steiner tree is NP-complete. Mapping the problem to standard Cartesian coordinates, the cost of an optimal Steiner tree is defined as the minimal rectilinear distance of all the edges. The heuristic proposed in this paper then finds a close approximation in fast (polynomial) time. The basis of exploits the idea of triangular distance, that is, using distance inequality to find the shortest path between three points. This is then further propagated across the coordinate plane. Thus, this solution makes an in order sweep based on the x-coordinates. The proposed solution has a varying running time, depending upon the distribution of the points. The worst case performance is given to be  $O(n^3)$ .

**Keywords.** Steiner tree, NP-Complete, Routing, Minimal Spanning Tree, Rectilinear Distance

## 1. Introduction

The problem of routing is one of the most time-consuming phases in VLSI chip design. Not only the connection distance between I/O pins need to minimized, the process must be done quickly. Though complex, this problem can be mapped to a well-known graph-theory problem, namely, creating an optimal Steiner tree based on rectilinear distance. This problem, further explained in this paper, is proven to be NP-complete [2]. Currently, there are two approaches to finding a close approximation to this problem – those based on Minimal Spanning Trees (MST) and those that are not; the heuristic presented in this paper falls into the latter category.

The rest of the paper will be organized in the following manner: Section 2 will define clearly the terminology and definitions used throughout the paper. Section 3 will then briefly describe techniques that have been explored previously. Section 4, the main focus, will thoroughly explain the newly proposed heuristic and analyze the running time. Finally, future work is proposed in Section 5 followed by closing remarks in Section 6.

## 2. Definitions and Terminology

The Steiner tree is a special case of a general graph  $G(V,E)$  where  $V$  is the set of points and  $E$  and is set of edges of which the points are connected. The problem of wire routing is modeled by the Steiner tree problem with the following mappings to graph theory terminology. Given a net, let:

$$\begin{aligned} V &= \text{Connection Points} \\ |V| = n &= \text{Total Number of Points or Nodes} \\ E &= \text{Wires Between Connection Points} \\ |E| &= \text{Total Distance of All Edges} \end{aligned}$$

The distance between two points  $p_1(x_1, y_1)$  and  $p_2(x_2, y_2)$  found in  $V$ , is defined as rectilinear (as opposed to Euclidean) with the following equation:

$$d = |x_1 - x_2| + |y_1 - y_2|$$

Lastly, the concept of the Steiner point is introduced. The Steiner point is a point not found in  $V$  but connects edges within  $G$ . With regards to this problem, any Steiner point is removable from the set of all points, must lie within the bounding box  $R$ , and has a degree of 3 or more. The presence of these points is the inherent difference between a Steiner tree and a minimal spanning tree (MST). The purpose of these points is to minimize edge length, as depicted in *Figure 1*.

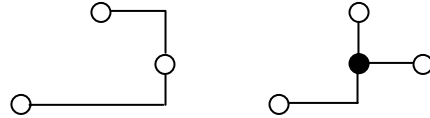


Figure 1: MST (left) and MRST (right) of three points

Particular to this paper, the following definitions are used in the heuristic. Given a set of points  $P \geq 2$  on the standard x-y coordinate system, let  $R$  be defined as the smallest bounding box that encloses all given points. Define  $R_L$  to be the distance long of  $R$  and  $R_H$  to the distance tall, shown in *Figure 2*. Hanan's Theorem states that within this area are all possible Steiner points and thus no single y (x) edge length can be greater than  $R_H$  ( $R_L$ ). Lastly, define the degree of a point, given or Steiner, to be the number of edges connected to the point. Note that since this problem only deals with rectilinear distance, the maximum degree a point can have is 4.

Given this terminology, the Minimum Rectilinear Steiner Tree (MRST) problem can be defined as follows: given a set of points  $V$ , construct a tree such that the total distance  $|E|$ , also known as cost, between the points in  $V$  and any number of Steiner points is minimized. Note that this definition allows the inclusion of zero Steiner points. This special case is the MST, where the total distance between only the points in  $V$  is minimized.

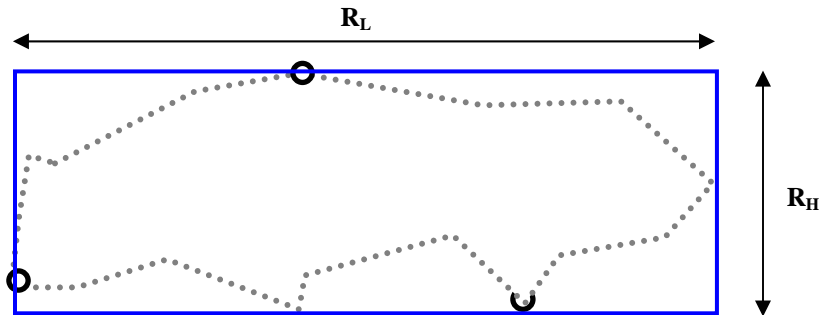


Figure 2: Bounding Box  $R$  enclosing a set of points

### 3. Previous Work

There are two general approaches that past heuristics have been based off of – those that use the MST as a basis and those that do not. It has been proven that the cost of the MST is at most 1.5 times that of the minimum rectilinear Steiner tree [4]. Thus, it is reasonable to start with a MST and improve upon that, as any future tree produced will be costlier than this bound. The base cost of constructing a MST has been shown by Hwang to be only  $O(n \lg n)$  [3]. Several approaches have already been proposed. Borah et al. [1] proposed an edge-based heuristic that incrementally improves upon a given MST. Another method based on recursively dividing MSTs until trees of sufficiently smaller size are reached and then solved exactly was proposed by Sarrafzadeh and Wong [7].

The other approach, for which the proposed algorithm is under, does not start with a MST. Kahng and Robins [6] have shown that the 1.5 bound is often tight for many MST-based heuristics and thus seem reasonable to explore other venues. Kahng and Robins have also developed their own heuristic not based on a MST. Their problem was slightly different, namely, their algorithm 1-Steiner iteratively finds the optimal Steiner point addition if only one additional point is allowed [5].

#### 4. The Heuristic – Algorithm and Analysis

The heuristic has two main components: construction and optimization. The construction portion, which makes creates new edges and Steiner points, uses the base idea of relative point placement and triangular distance. Sweeping from left to right, the heuristic builds optimal Steiner trees along the way and does not revisit any region it has already visited before. The construction portion when building local Steiner trees, can potentially build cycles in the global tree. The optimization portion, then, deletes any redundant and cycle-forming edges and Steiner points. Both portions are explained in detail as part of the algorithm given in *Figure 3*.

```

Algorithm STree()
{
    In:    Set of Points P given in (x,y) coordinates (distinct)
    Out:   Steiner Tree
        1) Set of Added Steiner Points S
        2) Set of edges that which connect all points in P and S for which
           distance is minimized

    1.    Begin
    2.    Sort all points P in ascending x-value order;
    3.    For all repeating x-values, sort in descending y-value;
    4.    TreeE =  $\emptyset$ ;
    5.    TreeP =  $\emptyset$ ;
    6.    ArrayLen =  $\emptyset$ ;
    7.    x-pointer = smallest x value;
    8.    p = first element of P;
    9.    while (x-pointer =< X_MAX)
        {
    10.       Create vertical edges between all points of same x-value;
    11.       Optimize();

    12.       while (p.x == current x-pointer)
            {
    13.           if (only two distinct x-values left)
    14.               Create right-down edge between two remaining
                    sets;
    15.               Optimize();
            else if (only three distinct x-values left)
    16.               MakeMiniSTree() with 3 distinct x-value sets;
    17.               p.next();
            }
    18.       x-pointer.next();
        }
    19.    End
}

```

*Figure 3 – Algorithm for Steiner Tree Construction*

The algorithm begins by sorting the points in ascending order and places the starting pointer at the beginning of the list, which is done in  $O(n \lg n)$  time. In order to keep track of the factors mentioned above, the heuristic employs one main tree: a tree of given points  $P$  and Steiner points  $S$ , an edge tree. Steiner points have already been defined in Section 2 so only edges will be further defined. Edges are kept track of their “start”, “middle”, and “end” nodes. The ordering is not important but for consistency sake, the order has been determined as the following: the “start” will have the smallest  $x$  value then smallest  $y$  value. Due to the bounding box  $R$ , there are a finite set of edges and additional nodes. However, the space that  $R$  occupies potentially can be big and thus to minimize space, dynamic data structures are used.

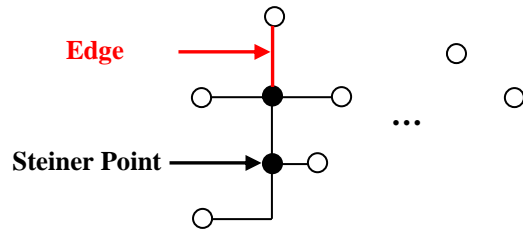


Figure 4 – Examples of a Steiner Point, Keeper Edge, and Potential Cycle Edge

The point tree is, in reality two binary search trees – a tree within a tree. The keys are the distinct  $(x,y)$  coordinates; the outer tree sorts by  $x$  while the inner tree sorts by  $y$ . Thus, a `search()`, `insert()`, and `delete()` operation for any point is done in  $O(\lg n)$  time. Each node stores the following information: type of point and list of edge connections – all nets the point is part of.

The main drawback with using cycles not being able to determine easily when a cycle is formed after an edge insertion. Thus, the heuristic keeps track of all possible mini-nets, that is, from one terminal end to another, along with where each net travel internally, in the current graph. When a new edge is inserted, it is easy to know if a cycle has been formed. This then implies that at any node insertion or deletion operation, each node in the tree will need to be updated of the new net or the deleted net. In the worst case, the update operation will take  $O(n)$  time.

In addition, there will be an array, which uses the net number as an index, that keeps track of the longest edge for each net and its location. Addressing this information is fast –  $O(1)$  time and the array will be updated when the tree is inserted, making it  $O(n)$  time. There will also be a tree with all the edges currently assigned in the tree. This is used to make sure that redundant edges are not added in the real tree. Each query to this tree then will take  $O(\lg n)$  time.

As shown, the bulk of the algorithm’s work is done with the two functions `MakeMiniSTree()` and `Optimize()`. `MakeMiniSTree()` has the power to make new additions to the graph while `Optimize()` has the power to delete. The following figure shows the pseudo-code for `MakeMiniSTree()`. In step 3, the algorithm refers to constructing optimal Steiner trees with three points. This is a sufficiently small set of points and thus can be solved explicitly. The algorithm, when constructing edges, goes as far right first. Thus, given a connection that goes up then right and right then up, the algorithm will choose the latter.

Let the three points within the Steiner tree be denoted  $A(x_A, y_A)$ ,  $B(x_B, y_B)$ , and  $C(x_C, y_C)$ , respectively, where  $(x,y)$  is relative to  $A$ . Without loss of generality, let  $A$  be the leftmost point,

B be the middle point, and C be the rightmost point. Equivalently,  $x_A < x_B < x_C$ . Finally, let the distance  $d_{AB} = d_{BA}$  between two points A and B be the rectilinear distance. The following cases only consider arrangements where B and C are found in A's Quadrants I and IV. Points that lie between I-II and IV-III are considered in the former Quadrants, respectively; those that are found on the I-IV boundary are considered to be in Quadrant I. Due to the nature of the algorithm, it is not possible for B and C to be in A's Quadrants II and III and thus will not be considered in the analysis.

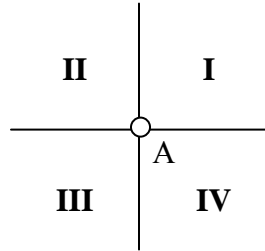


Figure 5 – A's Four Different Quadrants

There are only four possible arrangements for which B and C can be positioned (relative to A). The algorithm is given below and accounts for all four possibilities.

```

MakeMiniSTree()
{
    1.   Begin
    2.   if ((B∈QI  && C∈QI )           ||      // Case 1
    3.       (B∈QI  && C∈QIV))           ||      // Case 2
    4.       Create right-up edge EAB;
    5.   if ((B∈QIV && C∈QIV)           ||      // Case 3
    6.       (B∈QIV && C∈QI )           ||      // Case 4
    7.       Create right-down edge EAB;
    8.   Optimize();
    9.   if ((B∈QI  && C∈QI  && yB > yC) || || // Case 1
    10.      (B∈QIV && C∈QIV && yB < yC) || || // Case 3
    11.      (B∈QI  && C∈QIV)           || || // Case 2
    12.      (B∈QIV && C∈QI )           || || // Case 4
    13.   {
    14.       Construct Steiner Point s at yC and add to TreeP;
    15.       Add EsC to TreeE;
    16.       Split EAB into EAs and EsB;
    17.       Optimize();
    18.   }
    19.   else
    20.   {
    21.       if (B∈QI  && C∈QI )           // Case 1
    22.           Create right-up edge EBC;
    23.       if (B∈QIV && C∈QIV)           // Case 3
    24.           Create right-down edge EBC;
    25.       Optimize();
    26.   }
    27.   End
}

```

Figure 6 – Algorithm of MakeMiniSTree()

Each edge creation takes  $O(1)$  time, but each `Optimize()` (explained in a following sector) takes  $O(n)$  time. Thus, each call to `MakeMiniSTree()` takes  $O(n)$  time.

Note that in when B and C are in different quadrants, a Steiner point is required. Cases in which Steiner points are not required degenerates to finding the MST. The next figure depicts Cases 1 through 4 along with some examples of optimally constructed Mini-Steiner trees based on the `MakeMiniSTree()` algorithm.

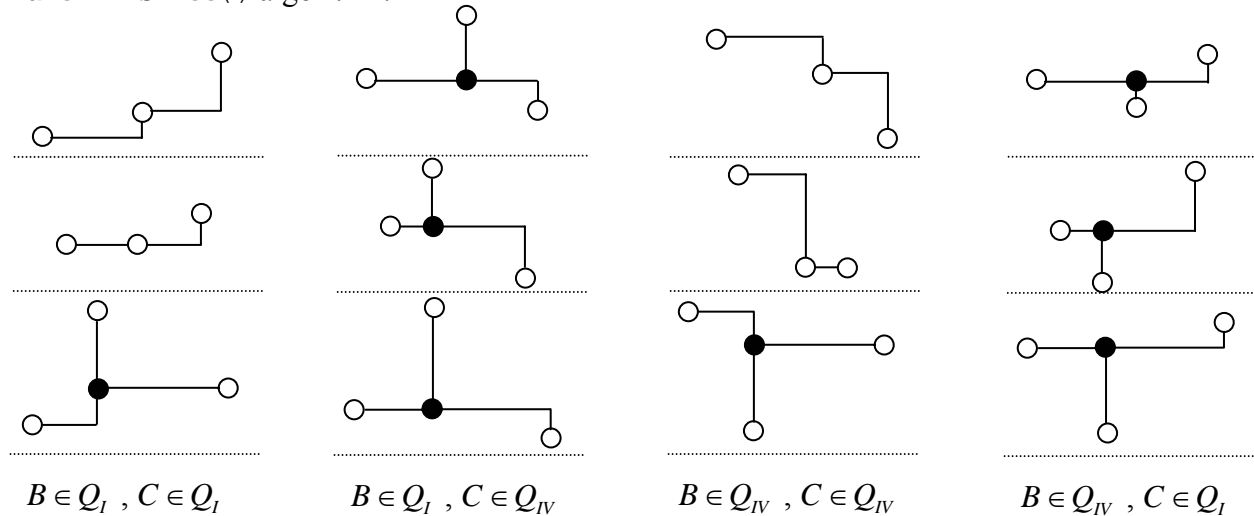


Figure 7 – Examples of Constructed Optimal Steiner Trees

The flaw with the construction portion is that it can and will produce duplicate edges and cycles. This is then rectified by the optimizing stage, where any edges and Steiner points can potentially be deleted. The optimizer uses the net-lists to determine if newly formed edges complete cycle(s). If they do, the algorithm deletes the longest edge found in the cycle. If there are multiple longest edges, the algorithm deletes an arbitrary horizontal edge. If an edge is removed, the optimizer must also check the point statuses to which the deleted edge was connected to. If the point is real, it does nothing. If the point is a Steiner point, the optimizer checks to see if the degree property is violated. Namely, if the point now only has degree 2 or fewer, then the point is removed from the point tree and the corresponding edges updated accordingly. The algorithm for `Optimize()` is shown below in the pseudo-code. Its running time, will run in  $O(n)$  time.

```

Optimize()
{
    1. Begin
    2. Remove redundant edge;
    3. Locate first cycle C;
    4. if (cycles exist)
        {
    5.     remove longest edge E from C;
    6.     update Steiner points affected by E;
    7.     update edges affected by E;
        }
    8. End
}

```

Figure 8 – Algorithm of `Optimize()`

After optimizing, the algorithm moves onto the next set of points with distinct x-values. Note the algorithm does not consider the last set of x-value points. The reasoning is the following: in step 7, all points with the same x-value are connected through vertical lines. After the second to last x-value, the graph has already optimized for all x-values previous and current; it is not possible then to find a shorter path than the already placed vertical lines. The following gives a simple example of what the `Optimize()` function does.

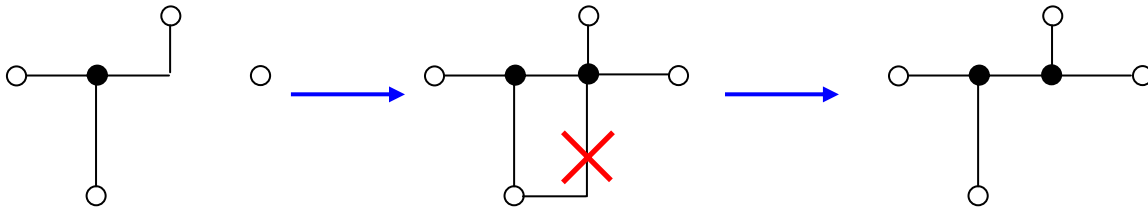


Figure 9 – Stages of `MakeMiniSTree()` [2 left] and `Optimize()` [right]

The last piece of information in regards to timing analysis is to know the number of iterations the main loop runs for. If each point were treated separately and taken in combination with every other point, the running time would result in  $O(n^3)$  iterations. However, due to Step 10 of the algorithm, this will substantially speed things up, for now each point is not considered individually but rather as part of a set. With the addition of this step, the number of iterations can be reduced to  $O(n^2)$ . Each iteration will take  $O(n)$ ; thus the worst case running time of this algorithm will take  $O(n^3)$  time.

## 5. Future Work

The Steiner tree problem can be explored in two possible areas – further optimization of the current problem and dimensional expansion. The main bottleneck of this heuristic is the determination of cycles' existence and the removal of the longest edge. If this problem can be done in  $O(\lg n)$  time, the entire heuristic will see a drastic drop in running time – the overall heuristic can then run in  $O(n^3 \lg n)$ . The problem presented in this paper had no limitations on the placement of the edge. Possible variations of the problem can include placing obstacles along the coordinate plane or minimizing the usage of either the x- or y- directional edges.

## 6. Conclusion

The MRST, proven to be NP-complete, can be best approximated by heuristics. Past approaches have included improving upon a given MST. The heuristic proposed in this paper can be improved upon with better techniques or the usage of more advanced data structures. The heuristic, having a polynomial running time, gives an impeccably close approximation to an optimal Steiner tree.

## 8. References

- [1] M. Borah, R. M. Owens, and M.J. Irwin, “An Edge-Based Heuristic for Steiner Routing”, *IEEE Trans. Computer-Aided Design*, vol. 13, no. 12, pp. 1563 – 1568, Dec. 1994.
- [2] M. Garey and D. S. Johnson, “The Rectilinear Steiner Tree Problem is NP-Complete”, *SIAM J. Appl. Math.*, vol. 13, no. 2, pp. 338 – 355, 1977.
- [3] F.K. Hwang, “An  $O(n \lg n)$  Algorithm for Rectilinear Minimal Spanning Trees”, *J. Assoc. Computing Machinery*, vol. 26, no. 2, pp. 177 – 182, Apr. 1979.
- [4] F. K. Hwang, “On Steiner Minimal Trees with Rectilinear Distance”, *SIAM J. Appl. Math.*, vol. 30, no. 1, pp. 104 – 114, 1976.
- [5] A. Kahng and G. Robins, “A New Class of Iterative Steiner Tree Heuristics with Good Performance”, *IEEE Trans. Computer-Aided Design*, vol. 11, no. 7, pp. 893 – 902, July 2002.
- [6] A. Kahng and G. Robins, “On Performance Bounds for Two Rectilinear Steiner Tree Heuristics in Arbitrary Dimension”, *Tech. Rep. UCLA CSD-TR-900015*, Apr. 1990.
- [7] M. Sarrafzadeh and C.K. Wong, “Hierarchical Steiner Tree Construction in uniform orientations”, *IEEE Trans. Computer-Aided Design*, vol. 11, no. 9, pp. 1095 – 1103, Sept. 1992.