An *O*(*n*log*n*) Edge-Based Algorithm for Obstacle-Avoiding Rectilinear Steiner Tree Construction

Jieyi Long, Hai Zhou, and Seda Ogrenci Memik

Department of Electrical Engineering and Computer Science

Northwestern University, Evanston, IL 60208

{jlo198, haizhou, seda} @ ece.northwestern.edu

ABSTRACT

Obstacle-avoiding Steiner tree construction is a fundamental problem in VLSI physical design. In this paper, we provide a new approach for rectilinear Steiner tree construction in the presence of obstacles. We propose a novel algorithm, which generates sparse obstacle-avoiding spanning graphs efficiently. We design a fast algorithm for the minimum terminal spanning tree construction, which is the bottleneck step of several existing approaches in terms of running time. We adopt an edge-based heuristic, which enables us to perform both local and global refinement, leading to Steiner trees with small lengths. The time complexity of our algorithm is $O(n\log n)$. Hence, our technique is the most efficient one to the best of our knowledge. Experimental results on various benchmarks show that our algorithm achieves 25.8 times speedup on average, while the average length of the resulting obstacle-avoiding rectilinear Steiner trees is only 1.58% larger than the best existing solution.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids [Placement and Routing]

General Terms

Algorithms, Design, Performance, Theory.

Keywords

Physical Design, Routing, Spanning Graph, Minimum Terminal Spanning Tree, Steiner Tree.

1. Introduction

Steiner routing is considered to be a fundamental problem and has been well studied over the years [1-4]. Most of the existing works on this problem assume an obstacle-free routing plane. However, modern integrated circuits often contain many obstacles such as IP cores, macro blocks, and pre-routed nets within the routing region. Consequentially, Obstacle-Avoiding Rectilinear Steiner Minimal Tree (OARSMT) construction arises as a more practical problem and has attracted increasing attention among VLSI physical design community recently [5-9].

ISPD'08, April 13-16, 2008, Portland, Oregon, USA.

Given a set of pins and a set of rectilinear obstacles, an OARSMT is a rectilinear tree connecting all the pins through a set of additional points (Steiner points) without running over the obstacles, while achieving the minimal possible total wire length. As a special case, the RSMT problem on an obstacle-free plane has been proven to be NP-complete [4]. Therefore, any exact algorithm for OARSMT construction is expected to have exponential worst case running time. On the other hand, the Steiner tree algorithm will be invoked millions of times during the floorplanning and placement phases [3, 10]. Hence, an efficient heuristic with good solution quality is highly desired.

In this paper, we provide a novel algorithm, which produces Obstacle-Avoiding Rectilinear Steiner Trees (OARST, not necessarily Steiner minimal trees) with short wire lengths. The time complexity is bounded by $O(n\log n)$. To the best of our knowledge, it is the most efficient existing algorithm in terms of asymptotic running time.

Several OARSMT heuristics have been proposed in the literature. They mainly fall into two categories [5-9]. The first class of OARST algorithms initially generate the Steiner tree without considering the obstacles and then "legalize" the edges that intersect with the obstacles. Yang et al. proposed a four-step algorithm for overlapping edge removal [6]. This kind of approach fails to exploit global blockage information, thus may produce low quality solutions as long routing detours may be introduced in overlapping edge removal step.

The second class of algorithms would first generate a connection graph that captures the global blockage information. Then, the Steiner tree construction is performed on this graph. The connection graph itself has the property of obstacle-avoidance. Hence, the later generated Steiner tree will naturally inherit the obstacle-avoidance feature. Since the connection graph usually carries the global geometrical information that can be exploited in the Steiner tree construction step, heuristics following this framework usually produce Steiner trees with shorter wire lengths. Early work adopting this strategy includes the escape-graph based heuristic proposed by Ganley et al. [5]. Escape-graph is conceptually similar to the Hannan grid [2]. Ganley et al. proved that there is at least one OARSMT embedded in the escape-graph. Thus, the computational geometry problem can be transformed into a graph-theoretical problem. They proposed an exact solution for three and four pin nets and heuristics for the nets with more pins.

Three algorithms proposed recently also fall into this category [7-9]. The connection graph used by Feng et al. [7] is the so called obstacle-avoiding constrained Delaunay triangulation,

while in Shen et al.'s [8] and Lin et al.'s [9] approaches, spanning graphs are used. The later steps are common to all: a minimum terminal spanning tree over this connection graph is generated and then refined to become a Steiner tree using heuristics. Feng et al.'s algorithm has $O(n\log n)$ worst case running time. However, sometimes the Steiner tree produced by their algorithm can have a large wire length, especially when the ratio between the number of obstacles and the number of pins is large. Shen et al.'s and Lin et al.'s algorithms can produce Steiner trees with better quality, but their algorithms are more expensive. Analysis shows that the worst case time complexity of Shen et al.'s and Lin et al.'s algorithms are $O(n^2\log n)$ and $O(n^3)$, respectively.

Our OARST construction algorithm shares Shen et al.'s and Lin et al.'s common structure. Despite the similarity of the frameworks, our algorithm is different from theirs in three aspects: First, we propose a novel algorithm which generates a sparse obstacle-avoiding spanning graph in $O(n\log n)$ time. Secondly, we designed an $O(n\log n)$ algorithm for minimum terminal spanning tree construction, which dominates the running time in their approaches. Finally, our edge-based heuristic employed for Steiner tree refinement can handle both global and local refinements, while to the best of our knowledge, all the existing OARST construction techniques make local refinements only. The time complexity of the refinement step is also $O(n\log n)$. Experimental results indicate the efficiency and effectiveness of our algorithm. Compared to Lin et al.'s heuristic, our algorithm achieves 33.1 times speedup on average, while the lengths of the resulting OARSTs are only 0.61% larger on average.

The rest of the paper is organized as follows. In Section 2, the formal formulation of the problem is presented, followed by the detailed discussion on the three-step-algorithm for OARST construction in Section 3. Our experimental results are provided in Section 4. We conclude with a summary of our contributions and findings in Section 5.

2. PROBLEM FORMULATION

The input to our algorithm consists of a set of *pin vertices* and a set of *rectilinear obstacles*. A rectilinear obstacle is an obstacle whose boundaries are either vertical or horizontal. A pin cannot reside inside any obstacle, but it could be located on the boundary of an obstacle. In addition, the obstacles are not allowed to overlap with each other. Nonetheless, they can be line-touched with one another. Notice that a *rectilinear obstacle* can be dissected into several rectangular blocks, as depicted in Figure 1. Hence, without loss of generality, we assume all obstacles are rectangular. A rectangular obstacle can be represented by its four *corner vertices*. Assuming that there are m pin vertices and k rectangular obstacles, the actual input to the algorithm are $n = m + 4k$ vertices. In the rest of the paper, we will use this number as the estimation of the algorithm input size.

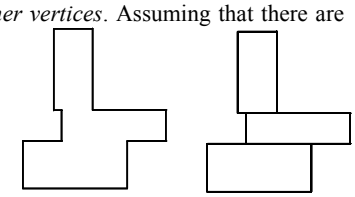

Figure 1. A rectilinear obstacle and its dissection.

The output of our algorithm contains an OARST connecting all the pin vertices. Some additional vertices, namely, *Steiner points*, may be added to the tree as internal nodes. A tree edge is not allowed to intersect with any obstacle. However, it can be point-touched at the corner or line-touched on the boundary with an obstacle. The *length* of the tree refers to the total length of all the edges of the tree. We formulate the OARSMT construction problem as follows:

Problem 1 (OARSMT): Given a set of pin vertices and a set of rectangular obstacles, construct an obstacle-avoiding rectilinear Steiner tree such that the length of tree is minimized.

3. Obstacle-Avoiding Rectilinear Steiner Tree Construction

In this section, we will present our heuristic for OARST construction consisting of the following three steps:

1. Obstacle-Avoiding Spanning Graph (OASG, defined in Section 3.1) generation: In this step, an OASG connecting all the pin vertices and all the corner vertices of the rectangular obstacles is generated efficiently.

2. Minimum Terminal Spanning Tree (MTST, defined in Section 3.2) construction: In this step, an MTST connecting all the pin vertices will be constructed by selecting edges from the OASG generated in the previous step.

3. Obstacle-Avoiding Rectilinear Steiner Tree construction: In this step, the MTST generated in the prior step will be used as an initial solution for further refinement. Steiner points will be introduced by an edge-based heuristic.

3.1 OASG Generation

We define the concept of OASG as follows:

Definition 1. Given a set of pin vertices and a set of rectangular obstacles, an undirected graph G connecting all the pin vertices and corner vertices is called an *obstacle-avoiding spanning graph* if none of its edges intersects with the obstacles.

Zhou et al. considered the problem of constructing the spanning graph on an obstacle-free plane [3]. Given a vertex u, they defined the *octal partition* of the plane with respect to u as the partition induced by the two rectilinear lines and the two 45 degree lines through u, as shown in Figure 2. They proposed to connect each vertex to its closest neighbor in each octant. They also showed that on an obstacle-free plane, the resulting spanning graph has only $O(n)$ edges and contains the minimum spanning tree for the pin vertices. However, when there are obstacles, it can be proven that this does not guarantee the inclusion of the minimum spanning tree. Lin et al. proposed another technique for spanning graph generation, which contains more "essential" edges and has certain optimal properties. However, Lin's spanning graph may contain up to $O(n^2)$ edges, which increases the time complexity of the later steps to a large extent, as compared to a sparse spanning graph with $O(n)$ edges.

Due to the concern on time complexity, we used the sparse spanning graph concept in our algorithm, although it may lead to sacrifice of quality of the initial solution. On the other hand,

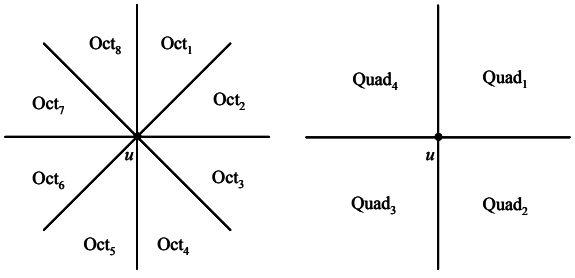

Figure 2. Octal and quadrant partition of the plane with respect to *u*.

```
ALGORITHM OASG-Quad1(P, C)
INPUT:  P  // the set of pin vertices
        C //  the set of corner vertices
OUTPUT: OASG-Quad1 // connection of the obstacle-
                   //avoiding spanning graph in Quad1
BEGIN
  Aev = Aeh = Av = Φ;
  Sort all the vertices in P∪C according to x + y;
  FOR EACH vertex v in the order BEGIN
    FOR EACH vertex u in Av such that v
        is in their Quad1 BEGIN
        IF no obstacle in Aeh or Aev blocks the
           connection between u and v BEGIN
           Add edge (u, v) to OASG;
           Remove u from Av;
        END
    END
    IF v is a corner vertex BEGIN
        Add/Remove the obstacle edges to/from the
          the active edge sets;
    END
    Add v to Av;
  END
END
```

Figure 3. Pseudo code for of the OASG edge connection.

since we employ a powerful edge-based heuristic capable of handling both local and global refinement in the third step, a poor initial solution may not necessarily lead to a Steiner tree with large length. As indicated by the experimental results presented in Section 4, our trade off results in short algorithm running time and good solution quality.

We propose a sweeping line algorithm to construct the OASG in $O(n\log n)$ time. Noticeably, Shen et al. have claimed an $O(n\log n)$ OASG construction algorithm [8]. These two OASG algorithms, though having the same time complexity and similar outcome, do not share a common structure. Moreover, Shen et al. did not give full description or complete complexity analysis for their algorithm. Particularly, the procedure for the 45 degree sweeping is omitted.

Different from Zhou et al's original idea, here we consider *quadrant partition* (depicted in Figure 2) only. Figure 3 provides the pseudo code of the OASG edge connection algorithm for Quad_1. The rest of the quadrants are symmetric so we can easily extend the discussion to handle them.

For Quad_1, we first sort all the vertices (both pins and corners) according to non-decreasing $x + y$. During the sweeping, we maintain an active vertex set A_v. It consists of the vertices whose nearest neighbors in Quad_1 are still to be discovered.

We connect the currently scanned vertex v to a vertex u in A_v that has v in its Quad_1 if the Manhattan connection between v and u does not run through any rectangular obstacle. Obviously, if the Manhattan connection between v and u cannot avoid a rectangular obstacle, the connection must intersect with either the left or lower edge of that obstacle. We thus maintain two active edge sets A_{ev} and A_{eh} to record the blockage information. A_{ev} (A_{eh}) contain the left vertical (lower horizontal) edges of the rectangular obstacles that are intersecting with the current sweeping line. When the lower (left) endpoint of the left (lower) edge e of a rectangular obstacle is scanned, e will be added to the active vertical (horizontal) edge set A_{ev} (A_{eh}). On the other hand, when we encounter the upper (right) endpoint of the left (lower) edge e of a rectangular obstacle, e will be removed from the active vertical (horizontal) edge set A_{ev} (A_{eh}).

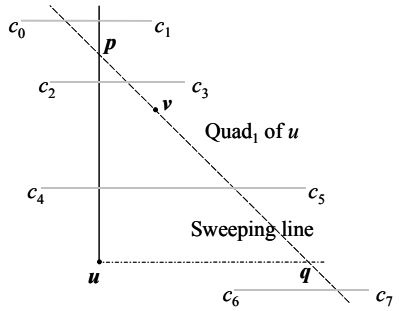

Figure 4. Ilustration of the blockage checking.

To check whether the Manhattan connection between v and u intersects with any edge in the active edge sets, we utilize the following lemma:

Lemma 1. The Manhattan connection between the currently scanned vertex $v(x_v, y_v)$ and an active vertex $u(x_u, y_u)$ with v in its Quad_1 intersects with a horizontal obstacle edge e_h if and only if 1) $y_u \le y_{cl} \le y_v$ where (x_{cl}, y_{cl}) is the coordinate of left end vertex of e_h, and 2) e_h is in the active horizontal edge set A_{eh}.

Proof. Figure 4 portraits the relative positions of the sweeping line, the currently scanned vertex v, a vertex u in the active set A_v, Quad_1 of u, and the horizontal edges in the active set A_{eh}. Notice that since the nearest neighboring vertex of u in its Quad_1 is yet to be discovered, there should not be any vertex (either pin or corner vertex) located within triangle Δupq. Therefore, the left end point of an active horizontal edge e_h (i.e., an edge in A_{eh}) should be on the left of line pu or below line qu. On the other hand, the right end point of an active horizontal edge must be on the right side of the sweeping line. Thus, the Manhattan connection between u and v intersects with e_h if and only if $y_u \le y_{cl} \le y_v$.

We yet need to show that a connection does not intersect with an "inactive" edge. We still use Figure 4 for illustration. Denoting the left and right end point of an inactive horizontal edge e_h by c_l and c_r, clearly, this edge can block the connection between u and v only when $x_{cl} < x_u$, $x_{cr} > x_v$, and $y_u \le y_{cl} \le y_v$. However, this implies c_r is located within Δupq, which is contradicting with our assumption that the closest neighboring vertex of u in Quad_1 is yet to be detected. Therefore, when making a connection, we do not need to check the inactive edges at all. □

We used the balanced binary search tree data structure to store the active horizontal edges with the y_{cl} values as their keys. Hence, at every attempt to connect a spanning graph edge, only $O(\log n)$ query time is needed. The vertical active edges can also be processed in a similar manner.

Now let us consider the data structure for the active vertex set A_v. On an obstacle-free routing plane, it can be shown that no vertex in the active vertex set can be in Quad_1 of another vertex in the same set [3]. This property enables the balanced binary search tree based implementation of the active vertex set, leading to a $O(n\log n)$ spanning graph generation algorithm. When there are obstacles, as depicted in Figure 5(a), the active vertex set may no longer have this property. However, a careful investigation still reveals a special structure of the active vertex set that can be exploited to guarantee $O(n\log n)$ running time. In Figure 5(b), we shade Quad_1 of each active vertex until hitting the sweeping line or the edges of the obstacles. The shaded area will be called the *active area*. We have the following observation:

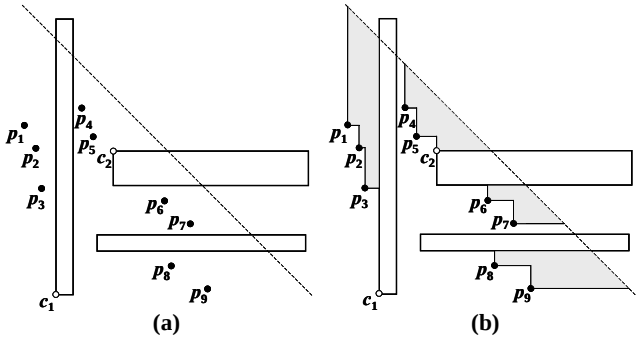

Figure 5. (a) The active vertices p_1~p_9 and c_1~c_2 (b) the active area composed of several disjoint active regions.

Lemma 2. The active area is composed of several disjoint regions, each having a segment on the sweeping line. These on-sweeping-line segments do not overlap with each other.

The disjoint regions will be called the *active regions*. We group the active vertices into *active groups*. Two active vertices are allocated in the same active groups if they are located in the same active region. Lemma 2 implies that the active regions, and thereby the active groups, have an order that is kept on only one dimension. On the other hand, similar to the situation on an obstacle-free plane, no vertex can be in $Quad_1$ of another vertex in the same active group, as there should not be any obstacle within each active region. Therefore, we can implement the active vertex set A_v based on a *hierarchical* balanced binary search tree, i.e., the active regions can be maintained by a balanced binary search tree while the active group in each region is maintained also by one balanced binary search tree, linked from that region. This data structure will guarantee $O(\log n)$ insertion, deletion, and query time. As the number of attempts to connect OASG edges is bounded by $O(n)$, the time complexity of OASG generation will be $O(n\log n)$.

3.2 MTST Construction

After generating the OASG, the next task is to obtain the minimum terminal spanning tree connecting all the pin vertices. Note that the spanning graph generated in the first step does not intersect with the obstacles, thus, the minimum terminal spanning tree over this graph will naturally inherit the obstacle-avoidance feature. The problem of finding the minimum terminal spanning tree over an OASG can be generalized after introducing the following concepts:

Definition 2. Given a non-negative weighted graph G with a subset of its vertices identified as *terminal vertices*, we call a loop-free path on G a *terminal path* if 1) its two end vertices are both terminals and 2) its does not contain other terminals except for the two end vertices.

Definition 3. Given a non-negative weighted graph G with a subset of its vertices identified as terminals, a graph G' composed of some terminal paths is called a *minimum terminal spanning tree* of G if 1) it connects all the terminals, and 2) it has the smallest possible length, where the length of G' is defined as the sum of the lengths of all the terminal paths on G'. The terminal paths consisting G' will be referred to as the *MTST paths*.

Note that some edges of G may be included in the MTST more than once. For instance, in Figure 6 (b), edge *ad* is included in the MTST twice. When we calculate the length of the MTST, we should count the length of *ad* twice. Also note that when the

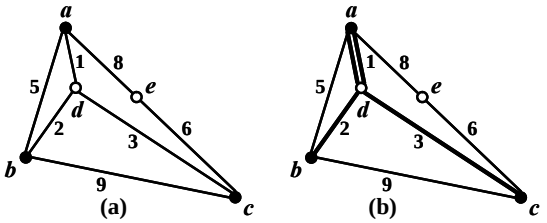

Figure 6. (a) A non-negative weighted graph G where the terminals are represented by black dots (b) the MTST of G is shown in bold lines. Notice that edge *ad* is included twice in the MTST.

vertices of a graph are all terminals, the MTST will be identical to the minimum spanning tree of this graph.

Problem 2 (MTST). Given a non-negative weighted graph G, construct the minimum terminal spanning tree of G.

Obviously, finding the MTST for an OASG is a special case of Problem 2, as the pin vertices can be viewed as terminal vertices. On the other hand, since G contains non-terminal vertices that may or may not be present on the minimum terminal spanning tree, the traditional algorithms for minimum spanning tree construction such as Kruskal's or Prim's algorithm cannot be applied. Lin et al. and Shen et al. both used a direct approach to construct the MTST for a given OASG. They first construct a complete graph for all the pin vertices, where the edge weight is equal to the shortest path length of its two end vertices on the OASG. The shortest path lengths for the pin pairs can be computed by Dijkstra's or Floyd-Washall algorithm. Then, they may either apply Kruskal's or Prim's algorithm to obtain the minimum spanning tree on the complete graph. At last, they map this minimum spanning tree back to the OASG to get the MTST. Although this approach can compute the desired MTST, it is expensive. Especially in Lin's algorithm, since the OASG may contain $O(n^2)$ edges, MTST generation takes $O(n^3)$ time in the worst case. In fact, this step is the bottleneck in Lin et al.'s and Shen et al.'s algorithms in terms of running time.

In this section, we propose a novel algorithm for solving Problem 2. The running time of this algorithm is $O(n\log n)$.

Definition 4. Given a non-negative weighted graph G, a directed sub-graph of G is called a *terminal forest* on G if 1) each tree in the forest contains exactly one terminal vertex and is rooted at this terminal, and 2) each vertex (can be either terminal or non-terminal vertex) belongs to one tree. A tree in the forest is called a *terminal tree*. The *root terminal* of a vertex v refers to the root of the terminal tree that v belongs to.

Definition 5. Given a non-negative weighted graph G and a terminal forest F on it, F is called a *shortest path terminal forest*

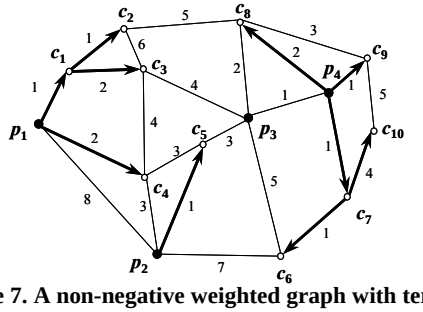

Figure 7. A non-negative weighted graph with terminal vertices p_1~p_4 and non-terminal vertices c_1~c_{10}. Its shortest path pin forest, which consists of four pin trees, is shown by the bold directed lines.

if 1) each tree in F is a shortest path tree, and 2) for any vertex v, its root terminal is the nearest one among all the terminals on G.

Figure 7 gives an example of a non-negative weighted graph, where the black dots p_1~p_4 are terminal vertices and hollow dots c_1~c_{10} are non-terminal vertices. A terminal forest on the graph is shown by the directed bold lines. Notice that, this terminal forest is also the shortest path terminal forest.

Definition 6. Given a non-negative weighted G and a terminal forest F on it, an edge $e(u, v)$ is called a *bridge edge* if its two end vertices belong to different terminal trees. Also, we call an edge $e(u, v)$ an *on-forest edge* if $e(u, v)$ belongs to one of the terminal trees. For an edge whose two end vertices belong to the same tree but not on the tree, we will call it an *intra-tree edge*.

In Figure 7, edges (c_4, c_5), (c_8, p_3) and (p_3, p_4) are examples of bridge edges. Edges (p_1, c_1) and (c_7, c_{10}) are examples of on-forest edges. Edges (c_2, c_3) and (c_8, c_9) are examples of intra-tree edges.

The following lemma indicates that to construct the minimum terminal spanning tree, we only need to consider the bridge edges and the on-forest edges.

Lemma 3. Given a non-negative weighted graph G, there is at least one MTST containing only bridge edges and on-forest edges.

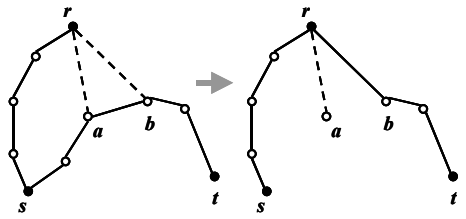

Figure 8. Illustration of the proof of Lemma 3.

Proof. Suppose intra-tree edge $e(a, b)$ in Figure 8 is part of the MTST path $path_{MTST}(s, t)$. Since $e(a, b)$ is an intra-tree edge, a and b should have a common root terminal r. We first remove path (s, b) from the MTST, and the MTST are divided into two components. Without lost of generality, we assume r is in the same component as s. We then add the shortest path between r and b (which consists of only on-forest edges) to the MTST. By definition, r is the closest terminal to b among all the terminals on G. Therefore, the length of the MTST does not increase. Notice this operation eliminates intra-tree edge (a, b) without introducing any new intra-tree edge into the MTST. Therefore, starting from any MTST, we can repeat the above process to obtain an MTST consisting of only bridge edges and on-forest edges. □

Given a non-negative weighted graph G and its sub-graph G_{fb} that consists of all the on-forest edges and bridge edges, we have the following extended cycle-property:

Lemma 4 (Extended Cycle-Property). If a terminal path on G_{fb} is the longest terminal path on a cycle on G_{fb}, then there is at least one minimum terminal spanning tree of G that does not contain this terminal path.

Lemma 4 indicates that after obtaining the shortest path terminal forest, Kruskal's algorithm could be extended to construct the minimum terminal spanning tree. On the other hand, the similarity between the shortest path terminal forest problem and the single source shortest paths problem inspired us to generalize Dijkstra's algorithm to solve it.

```
ALGORITHM Extended-Dijkstra(G)
INPUT:  G  // a non-negative weighted graph
OUTPUT: SPTF // the shortest path terminal forest
BEGIN
  // Initialization
  Heap H_v = Φ;
  FOR EACH vertex u of G BEGIN
    Set u.dist to 0 if u is a terminal vertex, +∞ otherwise;
    H_v.insert(u, u.dist); // use u.dist as the key
    u.parent = u;
    Make-Set(u);
  END

  // Shortest path terminal forest construction
  WHILE H_v is not empty BEGIN
    u = H_v.extractMin();
    Set-Union(u, u.parent);
    FOR EACH edge e(u, v) of G BEGIN
      IF v.dist > u.dist + e.length BEGIN
        v.dist = u.dist + e.length;
        v.parent = u;
        H_v.decreaseKey(v);
      END
    END
  END
  FOR EACH vertex u of G BEGIN
    u.root = Find-Set(u);
  END
END
```

Figure 9. Pseudo code of the extended-Dijkstra algorithm.

Figure 9 provides the pseudo code of the extended-Dijkstra's algorithm. It is similar to Dijkstra's algorithm with one exception: in the initialization step, we set the *dist* parameter of a vertex u to 0 if it is a terminal vertex; otherwise, we set it to $+\infty$. Using the concept of Dijkstra's algorithm, we essentially view the terminal vertices as multiple sources. During the shortest path terminal forest construction, the disjoint set data structure is utilized to record the root of each pin tree.

Lemma 5. The extended-Dijkstra algorithm generates the shortest path terminal forest for any non-negative weighted graph.

Now we can present the extended-Kruskal's algorithm for minimum terminal spanning tree construction. The pseudo code is given in Figure 10. It works the same way as the original Kruskal's algorithm. Exploiting the fact that there is a one-to-one correspondence between the bridge edges and the terminal paths of G_{fb}, we operate with the bridge edges instead of handling the terminal paths directly. To examine whether an edge is a bridge edge, we can simply check whether its two end vertices have different root terminals. Root terminals for the vertices have been computed in the last step of the extended-Dijkstra's algorithm using the Find-Set routine. The bridge edges are sorted according to the lengths of their corresponding terminal paths. For a bridge edge $e(u, v)$, the length of its corresponding terminal path is equal to $u.dist + e.length + v.dist$, where $u.dist$ and $v.dist$ record the distances of u and v to their root terminals, respective, and have been computed previously by the extended-Dijkstra algorithm. Along with the MTST, we also construct its merging tree, which will be used for the Steiner tree refining heuristic later. For the concept of the merging tree, please refer to [3].

Analysis of the running time of the extended-Dijkstra's algorithm is similar to the original Dijkstra's algorithm. As the edge number in the OASG is bounded by $O(n)$, the extend-Dijkstra's algorithm takes $O(n\log n)$ time. The same argument

```
ALGORITHM Extended-Kruskal(G, SPTF)
INPUT: G // a non-negative weighted graph
       SPTF // the shortest path terminal forest
OUTPUT: MTST // the minimum terminal spanning tree
        T_merg // the merging tree of the MTST
BEGIN
  // Initialization
  Heap H_be = Φ;
  Merging Tree T_merg = Φ;
  FOR EACH edge e(u, v) of G BEGIN
    IF u.root ≠ v.root BEGIN
      H_be.insert(e, u.dist + e.length + v.dist);
    END
  END

  // MTST and merging tree construction
  WHILE H_be is not empty BEGIN
    e(u, v) = H_be.extractMin();
    s1 = Find-Set(u);
    s2 = Find-Set(v);
    IF s1 ≠ s2 BEGIN
      Connect MTST edge e_MTST(u.root, v.root);
      s = Set-Union(s1, s2);
      s.edge = e_MTST;
      T_merg.merge(s, s1, s2);
    END
  END
END
```

Figure 10. Pseudo code of the extended-Kruskal algorithm.

applies to the extended-Kruskal's algorithm. Therefore, the time complexity of MTST generation is $O(n\log n)$.

Theorem 1. The extended-Dijkstra-Kruskal algorithm solves the MTST problem in $O(n\log n)$ time.

Note that when the vertices of the given non-negative weighted graph are all terminals, the extended-Dijkstra-Kruskal algorithm degenerates to Kruskal's algorithm. Therefore, when solving this special case, no extra work is actually needed.

The spanning graphs used by Shen et al., Lin et al, and us, though having different definitions, are all instances of the non-negative weighted graph. Thus, regarding the fact that the MTST generation step is the bottleneck of both Shen et al.'s and Lin et al.'s schemes, our extended-Dijkstra-Kruskal algorithm can be incorporated to speed them up.

3.3 OARST Construction

Having generated a MTST as the initial solution, the next step is to transform it into a Steiner tree by adding some Steiner points.

All the existing approaches for OARST construction only make local adjustments to the initial solution, meaning that the backbone of the resulting Steiner tree is restricted to the topology of the minimum terminal spanning tree. Hence, the improvement over the initial solution may be small [3].

Borah et al. proposed an edge-substitution heuristic, a simple yet effective approach for Steiner tree refinement (on a obstacle-free plane) [1]. Zhou et al. observed that the geometrical proximity information embedded in the spanning graph could be leveraged to simplify the heuristic [3]. In their algorithm, for each edge in the initial tree, all vertices that are neighbors of either of the end points on the spanning graph are considered to form vertex-edge pairs with the edge. The gain of each vertex-edge pair would be calculated to determine whether the edge-substitution should be made. In this section, we enhance the Borah-Zhou edge-based refinement to handle the obstacles.

Figure 11 illustrates the enhanced edge-substitution technique. As defined earlier, an MTST path is a terminal path on the

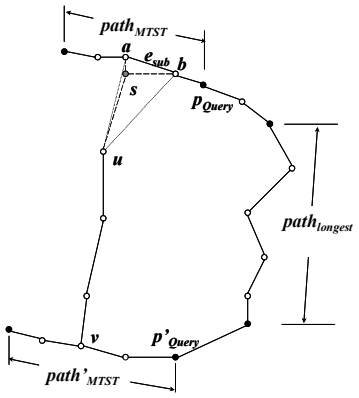

Figure 11. Illustration of the edge-substitution heuristic.

MTST. Furthermore, *sub-edges* of an MTST path refers to the OASG edges on this MTST path. For each sub-edge of each MTST path, we consider the vertex-edge pair formed by the sub-edge and each of its OASG neighboring vertices. An OASG vertex is called a neighboring vertex of a sub-edge if it is connected to either of the end points of the sub-edge. In Figure 11, suppose u is a neighboring vertex of $e_{sub}(a, b)$, we calculate the gain of vertex-edge pair (u, e_{sub}) in the following manner: we first find out the closest on-MTST vertex (can be either corner or pin vertex) of u (vertex v in Figure 11). Suppose e_{sub} and v are parts of MTST paths $path_{MTST}$ and $path'_{MTST}$, respectively. We will next find out the longest MTST path $path_{longest}$ between $path_{MTST}$ and $path'_{MTST}$ ($path_{MTST}$ and $path'_{MTST}$ excluded). Let us denote the Steiner point of vertices a, b and u by s. If we make the edge substitution, i.e., we connect new edges (s, a), (s, b), (s, u) and path (u, v), we will need to delete e_{sub} and $path_{longest}$ to maintain the tree topology. As the length of e_{sub} is equal to the sum of the lengths of (s, a) and (s, b), the gain of the vertex-edge pair can be computed by:

$$gain(u, e_{sub}) = len(path_{longest}) - len(path(u, v)) - len((s, u)).$$

As the following lemma implies, $len((s, u))$ is nothing but the Manhattan distance between the Steiner point s and vertex u.

Lemma 6. Assuming vertex u is a neighboring vertex of an MTST sub-edge $e_{sub}(a, b)$, and s is the Steiner point of a, b, and u, Manhattan connections from s to a, b, and u do not intersect with any obstacle.

Proof: We have two cases: first, if u and b reside in two non-neighboring quadrants (eg. $Quad_1$ and $Quad_3$), the problem becomes trivial since a is overlapping with s; secondly, as shown in Figure 12, if u and b reside in two neighboring quadrants (eg. $Quad_1$ and $Quad_2$) of a, there should not be any obstacle in the shaded area since u and b are the closest neighboring vertices of a in these two quadants. Therefore, Manhattan connection from s to a, b, and u can be made as the dashed bold lines in Figure 12. □

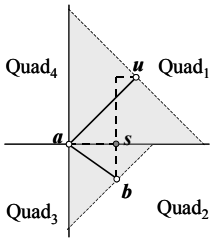

Figure 12. Steiner connection of a vertex-edge pair.

The value $len(path(u, v))$ can be computed efficiently using a simple variant of the extended-Dijkstra algorithm proposed in Section 3.2. This time we can instead view all the on-MTST vertices as the sources. The only modification we need to make is to set the *dist* parameters of the vertices to be zero if they are on the MTST, and $+\infty$ otherwise. We compute the nearest on-

MTST vertex for each vertex right after we have constructed the MTST and store these vertex pairs for later use.

The last problem is to compute the longest MTST path for a given MTST path pair ($path_{MTST}$, $path'_{MTST}$). In order to find this longest edge efficiently, we created along with the OARMTST its merging binary tree in the extended-Kruskal's algorithm similar to Zhou et al's approach [3]. The leaf nodes of the merging tree represent the pin vertices and the internal nodes represent the MTST paths. It can be proven that the common ancestor of two leaf nodes represents the longest MTST path between the two pin vertices. Tarjan's off-line least common ancestor algorithm can be used to find out the longest edges efficiently [11]. Noticing we have only the path pair ($path_{MTST}$, $path'_{MTST}$) in hand, to exploit the binary merging tree, we need to transform this edge pair to a pin vertex pair ((p_{Query}, p'_{Query}) in Figure 11). Obviously, a simple depth first search (DFS) fulfills our purpose. However, performing the DFS for all the edge pairs incurs $O(n^2)$ time overhead, since there are $O(n)$ edge pairs and each DFS takes $O(n)$ time. Observing that there are lots of overlaps among these DFSs, we can combine them into one Euler trail of the tree to eliminate the redundancy.

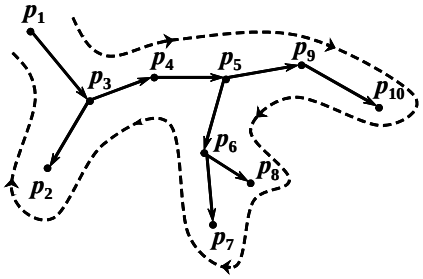

Figure 13. An example of the Euler trail of a directed MTST.

Figure 13 shows an example of the Euler trail on an MTST. We assign directions to the MTST paths to help clarify the illustration. Note that each path will be visited twice. When we travel through a path $path_{MTST}$ for the second time, we check all the path pairs involving $path_{MTST}$. Suppose ($path_{MTST}$, $path'_{MTST}$) is such a pair. If $path'_{MTST}$ has been visited twice already, the vertex pair for ($path_{MTST}$, $path'_{MTST}$) will be the starting vertices of these two paths. If $path'_{MTST}$ has just been visited only once, the vertex pair for ($path_{MTST}$, $path'_{MTST}$) will consist of the ending vertex of $path'_{MTST}$ and the starting vertex of $path_{MTST}$. If $path'_{MTST}$ has not been visited yet, we perform no action.

Lemma 7. The Euler trail procedure produces the pin vertex pairs for merging tree least common ancestor query in O(n) time.

Proof. To prove the correctness, we only need to note that when we visit $path_{MTST}$ for the second time, a path that has been gone through just once if and only if it is an ancestor of $path_{MTST}$ in the directed MTST. Notice that there are at most $O(n)$ path pairs and each path pair is to be checked twice. Besides, during the Euler traversal, each MTST path will be visited twice. Therefore, the time complexity of this procedure is $O(n)$. □

The edge substitution operations will then be made in a non-decreasing order of their gains. An edge substitution can only be made if none of $path_{MTST}$, $path'_{MTST}$ and $path_{longest}$ has been modified. The pseudo code for the edge-based Steiner tree refinement heuristic is provided in Figure 14.

The edge-based refinement involves computing the closest on-MTST vertex for each vertex, sorting the vertex-edge pairs according to their gain, transforming the edge pairs into vertex pairs, and performing merging tree least common ancestor query. Computing the closest on-MTST vertices using the variant of extended-Dijkstra's algorithm requires $O(n\log n)$ time. Sorting takes $O(n\log n)$ time also as there are at most $O(n)$ vertex-edge pairs. The time to transform the edge pairs into vertex pairs has been analyzed earlier, and it is $O(n)$. Tarjan's off-line least common ancestor query algorithm takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function which grows extremely slowly. Hence, the time complexity of the refinement step is still $O(n\log n)$.

```
ALGORITHM Edge-Substitution(MTST, T_merg)
INPUT: MTST // the minimum terminal spanning tree
       T_merg // merging tree of the MTST
OUTPUT: OARST // a obstacle-avoiding rectilinear Steiner tree
BEGIN
  Compute the gains for all the vertex-edge pairs;
  Sort the vertex-edge pairs according to their gains
    in non-decreasing order;
  FOR EACH vertex-edge pair (u, e_sub) in the order BEGIN
    IF none of path_MTST, path'_MTST and path_longest
      has been modified BEGIN
      Make the edge substitution, i.e., connect (s, a), (s, b),
        (s, u) and path (u, v), delete e_sub and path_longest;
    END
  END
END
```

Figure 14. Pseudo code for the edge-based Steiner tree refinement heuristic.

3.4 Time Complexity Analysis

We have shown in Section 3.1, 3.2, and 3.3 that the time complexity of OASG generation, MTST construction and edge-based refinement are all $O(n\log n)$.

Theorem 2. Given m pin vertices and k rectangular obstacles on a plane, our algorithm generates an obstacle-avoiding rectilinear Steiner tree in $O(n\log n)$ time, where $n = m + 4k$.

4. EXPERIMENTAL RESULTS

In this section, we provide the experimental results on several commonly used test cases [7, 9]. We also randomly generated some large test cases for further comparison.

We have implemented our algorithm in C++ language and compiled it using gcc 3.4.6. Regarding the difficulty of realizing the hierarchical binary search tree, in our actual implementation, we store the active vertices in a normal binary search tree. Thus, the running time complexity of our program is higher. However, as shown later, the empirical running time of our implementation has been quite small. Our experiments were conducted on a Redhat Linux sever with two 2.1GHz Dual Core AMD Opteron™ processors and 2GB memory.

We compared our results with Feng et al.'s, Shen et al.'s, and Lin et al.'s. We executed Shen et al.'s and Lin et al.'s algorithms on our platform. Feng et al.'s results are quoted from their paper, where their algorithm was tested on a Sun V800 fire workstation with a 755MHz CPU and 4GB memory [7]. Comparison among the four algorithms is provided in Table 1, where IND01~IND05, RC01~RC12 are test cases used in previous works [7-9], and RL01~RL05 are five randomly generated large test cases by us. Column "$\Delta w\%$" provides the relative improvement of our OARSTs over Lin et al.'s and is calculated by

Table 1. Comparison of the experimental results among different techniques.

			Tree Weight					Runnig Time (sec)				
Benchmark	**m**	**k**	**Feng et al.**	**Shen et al.**	**Lin et al.**	**Ours**	**Δw%**	**Feng et al.**	**Shen et al.**	**Lin et al.**	**Ours**	**speedup**
IND01	10	32	—	646	632	649	-2.69%	—	0.01	0.01	0.01	1.0 x
IND02	10	43	—	10,100	9,600	10,100	-5.21%	—	0.01	0.01	0.01	1.0 x
IND03	10	50	—	623	613	623	-1.63%	—	0.01	0.01	0.01	1.0 x
IND04	25	79	—	1,121	1,121	1,131	-0.89%	—	0.02	0.02	0.02	1.0 x
IND05	33	71	—	1,392	1,364	1,379	-1.10%	—	0.02	0.02	0.02	1.0 x
RC01	10	10	30,410	27,730	26,900	27,540	-2.38%	0.01	0.01	0.01	0.01	1.0 x
RC02	30	10	45,640	42,840	42,210	42,030	0.43%	0.01	0.02	0.01	0.01	1.0 x
RC03	50	10	58,570	56,440	55,750	56,070	-0.57%	0.01	0.02	0.01	0.01	1.0 x
RC04	70	10	63,340	60,840	60,350	59,550	1.33%	0.01	0.02	0.02	0.02	1.0 x
RC05	100	10	83,150	76,970	76,330	76,320	0.01%	0.01	0.03	0.02	0.02	1.0 x
RC06	100	500	149,750	86,403	83,365	87,432	-4.88%	0.06	0.22	0.16	0.14	1.1 x
RC07	200	500	181,470	117,427	113,260	117,855	-4.06%	0.06	0.37	0.30	0.15	2.0 x
RC08	200	800	202,741	123,366	118,747	124,852	-5.14%	0.10	0.52	0.45	0.27	1.7 x
RC09	200	1,000	214,850	119,744	116,168	120,554	-3.78%	0.13	0.71	0.63	0.36	1.8 x
RC10	500	100	198,010	171,450	170,690	168,859	1.07%	0.03	0.33	0.62	0.08	7.8 x
RC11	1,000	100	250,570	238,111	236,615	235,795	0.35%	0.04	1.10	1.27	0.15	8.5 x
RC12	1,000	10,000	1,723,990	843,529	789,097	852,401	-8.02%	2.82	63.82	79.59	5.93	13.4 x
RL01	5,000	5,000	—	503,032	492,856	504,887	-2.44%	—	136.41	161.06	5.18	31.1 x
RL02	10,000	500	—	648,898	648,508	641,445	1.09%	—	143.03	218.73	2.28	95.9 x
RL03	10,000	100	—	652,323	652,241	644,616	1.17%	—	127.82	204.61	2.04	100.3 x
RL04	10,000	10	—	710,005	709,904	701,088	1.24%	—	124.84	256.81	1.85	138.8 x
RL05	10,000	0	—	741,978	741,697	731,790	1.34%	—	127.69	284.26	1.84	154.5 x
Average	—	—	—	—	—	—	-1.58%	—	—	—	—	25.8 x

$$\Delta w\% = -(length_{ours} - length_{Lin\ et\ al.'s}) / length_{Lin\ et\ al.'s} \times 100\%.$$

Column "speedup" compares the execution time of our algorithm and Lin et al.'s. It is calculated by

$$speedup = (execution\ time)_{Lin\ et\ al.'s} / (execution\ time)_{ours}.$$

First, we observe that compared to Feng et al.'s algorithm, our algorithm performs consistently better in terms of OARST quality. Especially for the benchmarks with large *k*/*m* ratio (RC06, RC07, RC08, RC09, RC12), our algorithm produces OARSTs with substantially smaller length. For instance, for RC12, length of our OARST is less than half of Feng et al.'s.

Secondly, compared to Shen et al.'s and Lin et al.'s algorithms, our algorithm terminates in much shorter time, especially for the large benchmarks (RC12, RL01~RL06). For all the test cases, on average, our algorithm runs 33.1 times faster than Lin's algorithm. On the other hand, in terms of OARST quality, our algorithm performs comparable to Shen et al.'s and Lin et al.'s. On average, our OARSTs are only 0.61 % longer than those of Lin et al.'s.

We also observed that our algorithm produces better OARSTs when the ratio *k*/*m* is less than one. For example, for test case RC02, RC03, RC04, RC05, RC10 and RC11, our OARST has smaller length than those of Shen et al's and Lin et al.'s. Furthermore, experimental results for the six large benchmarks RL01~RL06 reveal that as *k*/*m* approaches zero our algorithm performs better in terms of solution quality. In the limiting case (RL06), *k*/*m* equal to zero, the problem becomes constructing an SMT on an obstacle-free plane. Existing works have shown that global refinement techniques such as edge-based heuristic perform better than the local refinement techniques in this limiting case. Our results are consistent with this observation.

5. CONCLUSIONS

In this paper, we have presented an efficient three-step algorithm for obstacle-avoiding rectilinear Steiner tree construction. We devise a novel algorithm to efficiently generate the OASG and the MTST. We also adapt an edge-based global refinement technique into our scheme. Experimental results indicate that our approach is an efficient yet effective approach for OARST construction. Compared to Lin et al.'s heuristic, our algorithm achieves 33.1 times speedup on average, while the length of the resulting OARSTs is only 0.61% larger on average.

6. ACKNOWLEDGMENTS

This work was partially supported by NSF under CNS-0613967.

7. REFERENCES

1. Borah, M., R.M. Owens, and M.J. Irwin, *An Edge-Based Heuristic for Steiner Routing.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 1994. **13**(12): p. 1563-1568.
2. Hannan, M., *On Steiner's Problem with Rectilinear Distance.* SIAM Journal on Applied Mathematics, 1966. **14**: p. 255-265.
3. Zhou, H., *Efficient Steiner Tree Construction Based on Spanning Graph.* IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2004. **23**(5): p. 704-710.
4. Garey, M. and D. Johnson, *The Rectilinear Steiner Tree Problem is NP-Complete.* SIAM Journal on Applied Mathematics, 1977. **32**: p. 826-834.
5. Ganley, J.L. and J.P. Cohoon. *Routing a Multi-Terminal Critical Net: Steiner Tree Construction in the Presence of Obstacles*. in *Int. Symp. on Circuits and Systems*. 1994.
6. Yang, Y., et al. *Rectilinear Steiner Minimal Tree among Obstacles*. in *Int. Conf. on ASIC*. 2003.
7. Feng, Z., et al. *An O(nlogn) Algorithm for Obstacle-Avoiding Routing Tree Construction in the Lambda-Geometry Plane*. in *Int. Symp. on Physical Design*. 2006.
8. Shen, Z., C. Chu, and Y. Li. *Efficient Rectilinear Steiner Tree Construction with Rectilinear Blockages*. in *Int. Conf. on Computer Design*. 2005.
9. Lin, C., et al. *Efficient Obstacle-Avoiding Rectilinear Steiner Tree Construction*. in *Int. Symp. on Physical Design*. 2007.
10. Pan, M. and C. Chu. *FastRoute: A Step to Integrate Global Routing into Placement*. in *Int. Conf. Computer Aided Design*. 2006.
11. Cormen, T.H., et al., *Introduction to Algorithms*. 1989: MIT Press.