EBOARST: An Efficient Edge-Based Obstacle-Avoiding Rectilinear Steiner Tree Construction Algorithm

Jieyi Long, *Student Member, IEEE*, Hai Zhou, *Senior Member, IEEE*, and Seda Ogrenci Memik, *Senior Member, IEEE*

***Abstract*—Obstacle-avoiding Steiner routing has arisen as a fundamental problem in the physical design of modern VLSI chips. In this paper, we present EBOARST, an efficient four-step algorithm to construct a rectilinear obstacle-avoiding Steiner tree for a given set of pins and a given set of rectilinear obstacles. Our contributions are fourfold. First, we propose a novel algorithm, which generates sparse obstacle-avoiding spanning graphs efficiently. Second, we present a fast algorithm for the minimum terminal spanning tree construction step, which dominates the running time of several existing approaches. Third, we present an edge-based heuristic, which enables us to perform both local and global refinements, leading to Steiner trees with small lengths. Finally, we discuss a refinement technique called segment translation to further enhance the quality of the trees. The time complexity of our algorithm is $O(n \log n)$. Experimental results on various benchmarks show that our algorithm achieves 16.56 times speedup on average, while the average length of the resulting obstacle-avoiding rectilinear Steiner trees is only 0.46% larger than the best existing solution.**

***Index Terms*—Graph algorithm, obstacle-avoiding, routing, Steiner tree.**

I. Introduction

EVER SINCE being introduced by Hannan in 1966 [1], the rectilinear Steiner minimal tree (RSMT) problem has been the focus of active research, due not only to its theoretical importance but also its practical implications [1]–[7]. RSMT has wide applications in electronic design automation. Particularly in the routing phase of VLSI circuit physical design, RSMT is usually used as the initial net topology for global routing. Moreover, in earlier design stages like floorplanning and placement, it is also utilized to estimate the total wire length, congestion, and timing. These estimates can further serve as guiding criteria for later timing- or congestion-driven routing [3]. Most of the existing work on the RSMT problem assumes an obstacle-free routing plane. However, as modern VLSI designs shift toward the system-on-a-chip paradigm, many reusable components such as hard IP cores and macro blocks are incorporated to improve design efficiency. These components are predesigned. Hence, the interconnects are not allowed to run over them most of the time. Consequentially, obstacle-avoiding RSMT (OARSMT) construction arises as a more practical problem and has attracted renewed attention in the VLSI physical design community [8]–[15].

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. Note that besides obstacles, during physical synthesis, other practical constraints such as congestion, timing, and layer assignment should be taken into consideration. The OARSMT construction algorithms can be generalized to handle these constraints. For instance, in congestion-driven routing, overcongested regions can be modeled as obstacles [16]. On an obstacle-free routing plane, the OARSMT problem degenerates to the RSMT problem, which has been proven to be NP complete [4]. Therefore, any exact OARSMT construction algorithm 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 [7], [17]. Hence, an efficient heuristic with good solution quality is highly desired.

In this paper, we provide a novel four-phase algorithm EBOARST, which produces obstacle-avoiding rectilinear Steiner trees [OARSTs, not necessarily Steiner minimal trees (SMTs)] with small total wire lengths. The time complexity of the algorithm is $O(n \log n)$. Experimental results on various benchmarks illustrate the effectiveness and efficiency of our algorithm. Even for the largest benchmarks containing up to 10 000 obstacles or pins, our algorithm is able to produce high-quality solutions within 6 s. Across all benchmarks, our algorithm achieves 16.56 times speedup on average, while the average wire length of the resulting Steiner trees is only 0.46% larger than the best existing solution.

The rest of this paper is organized as follows. In Section II, we review the previous work on the OARSMT problem. In Section III, the formal formulation of the problem is presented, followed by the detailed discussion on the four-step algorithm for OARST construction in Section IV. Experimental results are provided in Section V. We conclude with a summary of our contributions and findings in Section VI.

Manuscript received February 6, 2008; revised May 21, 2008 and July 12, 2008. Current version published November 19, 2008. This work was supported in part by the NSF under Grant CNS-0613967. This paper was recommended by Associate Editor D. Z. Pan.

The authors are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: jieyi.long@u.northwestern.edu; jlo198@eecs.northwestern.edu).

Digital Object Identifier 10.1109/TCAD.2008.2006098

II. Related Work

Early works on the OARSMT problem are mostly multipin variants of the maze routing algorithm [18]–[21]. They generally have high space demand but often lead to solutions which are far from optimal. Line-search-based heuristics require less storage space, but the routing quality is not guaranteed for multiple terminals [22], [23].

More recent OARSMT heuristics proposed in the literature adopt different strategies [8]–[11], [13]. They mainly fall into two categories. The first class of OARST algorithms initially generates the Steiner tree without considering the obstacles and then "legalizes" the edges that intersect with the obstacles. Yang *et al.* [13] proposed a four-step algorithm for overlapping edge removal. This kind of approach fails to exploit global blockage information and, thus, may produce low-quality solutions as long routing detours may be introduced in the 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.* [9]. Escape graph is conceptually similar to the Hannan grid [1]. 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.

Shi *et al.* [12] proposed to use the global routing graph (GRG) which contains the escape graph as its subgraph as the connection graph. They developed an interesting circuit-simulation-based technique to select the Steiner points from the vertices of the GRG. Although high-quality solutions are reported for benchmarks with up to 500 pins and 20 obstacles, the algorithm runs relatively slowly, particularly for the test cases with large number of obstacles.

Three algorithms proposed lately also fall into this category [8], [10], [11]. The connection graph used by Feng *et al.* [8] is the so-called obstacle-avoiding constrained Delaunay triangulation, while in the approaches of Shen *et al.* [11] and Lin *et al.* [10], spanning graphs are used. The later steps are common to all: A minimum terminal spanning tree (MTST, defined in Section IV-B) over this connection graph is generated and then refined to become a Steiner tree using heuristics. The algorithm of Feng *et al.* has $O(n \log n)$ worst case running time. However, the Steiner tree produced by their algorithm may have a large total wire length, particularly when the ratio between the number of obstacles and the number of pins is large. Our algorithm is able to produce Steiner trees of significantly better quality with the same computational complexity. Compared with the algorithm of Shen *et al.*, our approach is able to achieve better quality and lower complexity at the same time. The algorithm of Shen *et al.* involves constructing a complete graph whose vertex set contains all the pins and corner vertices. Based on our understanding of their algorithm, its time complexity is at least $O(n^2)$. In fact, it was mentioned that the worst case time complexity of the algorithm of Shen *et al.* is $\Omega(n^2 \log n)$ [16]. The algorithm of Lin *et al.* produces Steiner trees with comparable quality as our approach, but their algorithm is more expensive $(O(n^3))$.

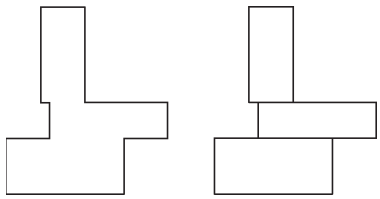

Fig. 1. Rectilinear obstacle and its dissection.

Our OARST construction algorithm shares the common structure of Shen *et al.* and Lin *et al.* Despite the similarity of the frameworks, our algorithm is different from theirs in four aspects: First, we propose a novel algorithm which generates a sparse obstacle-avoiding spanning graph (OASG) in $O(n \log n)$ time. Second, we designed an $O(n \log n)$ algorithm for MTST construction, which dominates the running time in their approaches. Third, our edge-based heuristic employed for Steiner tree refinement can handle both global and local refinements, while to the best of our knowledge, all existing OARST construction techniques make local refinements only. Finally, to further optimize the OARST, we employ a local refinement technique called segment translation.

We designed an $O(n \log n)$ OARST algorithm recently [24]. In this paper, we extend our preliminary work in several aspects.

1) We designed a local refinement technique called *segment translation*. Segment translation improves the solution quality while incurring negligible time overhead.
2) We present a detailed theoretical background which demonstrates the correctness and efficiency of our algorithm.
3) We present experimental results on an extended set of benchmarks and perform comparisons between our approach and previous work in more detail.

III. 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 shown in Fig. 1. Hence, without loss of generality, we assume that 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 inputs to the algorithm

are $n = m + 4k$ vertices. In the rest of this paper, we will use this number as the estimation of the algorithm input size.

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 OARST such that the length of tree is minimized.

IV. OARST Construction

In this section, we will present EBOARST, an efficient edge-based OARST construction algorithm. It consists of the following four steps.

1) OASG (defined in Section IV-A) generation: In this step, an OASG connecting all the pin vertices and all the corner vertices of the rectangular obstacles is generated efficiently.
2) MTST 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) OARST 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.
4) Local refinement: In this step, we apply a local refinement technique called segment translation to further improve the quality of the OARST.

A. 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 OASG if none of its edges intersects with the obstacles.

Zhou [7] considered the problem of constructing the spanning graph on an obstacle-free plane. 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° lines through u, as shown in Fig. 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, the spanning graph of Lin *et al.* 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.

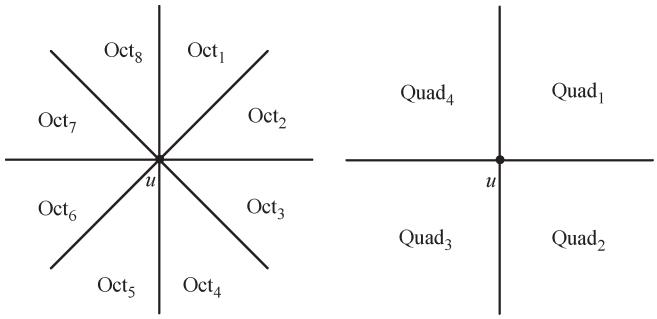

Fig. 2. Octal and quadrant partition of the plane with respect to u.

```
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 active edge sets;
    END
    Add v to Av;
  END
END
```

Fig. 3. Pseudocode for of the OASG edge connection algorithm for Quadrant 1.

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, since we employ two powerful refinement heuristics capable of handling both local and global enhancements in the last two steps, a poor initial solution may not necessarily lead to a Steiner tree with large length. As indicated by the experimental results presented in Section V, our tradeoff 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.* [11] have claimed an $O(n \log n)$ OASG construction algorithm. These two OASG algorithms, although having the same time complexity and similar outcome, do not share a common structure. For instance, in the algorithm of Shen *et al.*, both horizontal/vertical sweeping and 45° sweeping are required. However, in our proposed algorithm, only 45° sweeping is needed. Moreover, Shen *et al.* did not give full description or complete complexity analysis for their algorithm. In particular, the procedure for the 45° sweeping is omitted.

Different from the original idea of Zhou, here, we consider *quadrant partition* (shown in Fig. 2) only. Fig. 3 shows the pseudocode of the OASG edge connection algorithm for Quad_1. The rest of the quadrants are symmetrical, so we can easily extend the discussion to handle them.

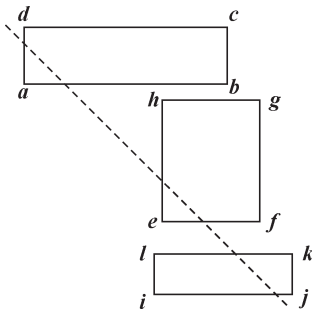

Fig. 4. Active horizontal and vertical edges.

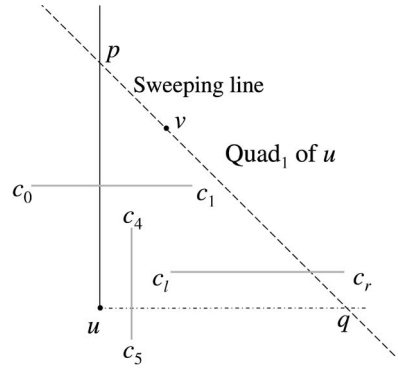

Fig. 5. Illustration of the blockage checking.

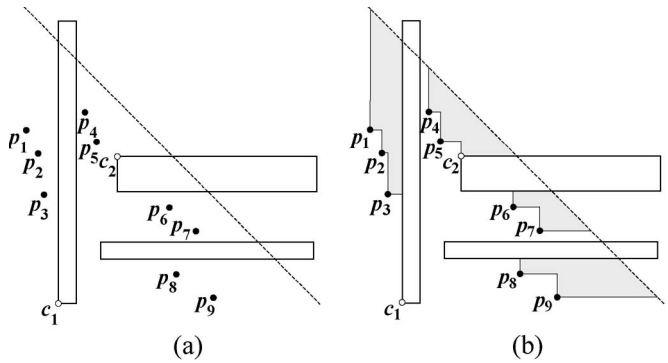

Fig. 6. (a) Active vertices $p_1 \sim p_9$ and $c_1 \sim c_2$. (b) Active area composed of several disjoint active regions.

For Quad_1, we first sort all the vertices (both pins and corners) according to nondecreasing $x + y$. 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} (or A_{eh}) contains the left vertical (or lower horizontal) edges of the rectangular obstacle that are intersecting with the current sweeping line. For instance, in Fig. 4, the current sweeping line intersects with three rectangular obstacles, and the active vertical edge set A_{ev} contains edges $e(a, d)$ and $e(e, h)$, while the active horizontal edge set A_{eh} contains edges $e(a, b)$, $e(e, f)$, and $e(i, j)$. 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}). For instance, when the sweeping line reaches vertex a, edge $e(a, d)$ is added to set A_{ev}, and edge $e(a, b)$ is added to set 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}). Still, by using the example provided in Fig. 4, when the sweeping line arrives at vertex d, edge $e(a, d)$ will be removed from set A_{ev}; when the sweeping line arrives at vertex b, edge $e(a, b)$ will be removed from set A_{eh}.

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 at least one horizontal obstacle edge if and only if for the left end point $(x_{\text{cl}}, y_{\text{cl}})$of the active horizontal edge with the smallest $y_{\text{cl}} \geq y_u$, we have $y_{\text{cl}} < y_v$ and $x_{\text{cl}} < x_u$.

Proof: i) Sufficiency: denote the active horizontal edge with the smallest $y_{\text{cl}} \geq y_u$ by e_h. Firstly, if $x_{\text{cl}} < x_u$ holds, then the left end point of e_h must be on the left of line pu. Secondly, since e_h is an active edge, its right end point must lie on the right side of the sweeping line. Finally, since $y_v > y_{\text{cl}} \geq y_u$, e_h must intersects with the Manhattan connection between u and v.

ii) Necessity: Fig. 5 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 several obstacle edges. If the Manhattan connection between u and v intersects with one horizontal obstacle edge, say edge (c_0, c_1), let us denote the active horizontal edge with the smallest $y_{\text{cl}} \geq y_u$ by (c_l, c_r). Firstly we have $y_{\text{cl}} \leq y_{c0} < y_v$. Secondly, if point c_l lies on the right of line pu, the Manhattan connection between u and c_l must be blocked by an vertical obstacle edge, say (c_4, c_5), since the closest neighboring vertex of u in its Quad_1 is yet to be found. However, in this case, it can be shown that u must have already connected to one point in Quad_1, which again contradicts with our assumption that the closest neighboring vertex of u in its Quad_1 still needs to be discovered. Therefore, the left end point c_l must be on the left of line pu, i.e., $x_{cl} < x_u$. Similarly, we can show that under the assumption that the closest neighboring vertex of u in its Quad_1 has not been found, point c_r must lie on the right of the sweeping line. Hence, edge (c_l, c_r) must be an active edge. ■

We used the balanced binary search tree data structure to store the active horizontal edges with the y_{cl} values as their keys. When performing the intersection checking, we extract the active horizontal edge with the smallest y_{cl} value among those active edges satisfying $y_u \leq y_{\text{cl}}$. Extracting such an edge takes $O(\log n)$ query time. Then, we check whether conditions $y_{\text{cl}} \leq y_v$ and $x_{\text{cl}} < x_u$ hold true. This condition checking can be performed in constant time. Hence, at every attempt to connect a spanning graph edge, only $O(\log n)$ 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

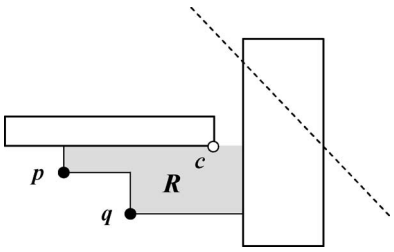

Fig. 7. "Active region" that does not touch the sweeping line.

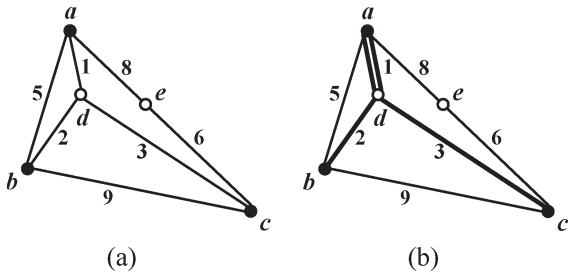

Fig. 8. (a) Nonnegative weighted graph G where the terminals are represented by black dots. (b) MTST of G is shown in bold lines. Notice that edge ad is included twice in the MTST.

no vertex in the active vertex set can be in Quad_1 of another vertex in the same set [7]. This property enables the balanced-binary-search-tree-based implementation of the active vertex set, leading to an $O(n \log n)$ spanning graph generation algorithm. When there are obstacles, as shown in Fig. 6(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 Fig. 6(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.

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.

Proof: First, it is obvious that the active area is composed of one or more disjoint region(s). Second, assume that there are regions that do not touch the sweeping line and that R is one of them (Fig. 7). Since R is constructed by shading Quad_1 of a set of active vertices (denoted by P) until hitting the sweeping line or the edges of the obstacles, there should not be any obstacle inside R. Hence, R can be viewed as an obstacle-free routing region. If R does not touch the sweeping line, then R at least touches a horizontal and a vertical edge of the obstacles, which implies that there is at least one corner point on the boundary of region R (corner c in Fig. 7). As R can be viewed as an obstacle-free routing region, direct connection can be made between these on-boundary corner points and the points in vertex set P. This fact conflicts with our assumption that the nearest points in Quad_1 of the vertices in vertex set P are still to be determined. Finally, as we have proved that each disjoint region shares a segment with the sweeping line, it is obvious that these on-sweeping-line segments are nonoverlapping. ■

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 $O(n \log n)$.

B. MTST Construction

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

Definition 2: Given a nonnegative 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 the following are true: 1) Its two end vertices are both terminals, and 2) it does not contain other terminals except for the two end vertices.

Definition 3: Given a nonnegative weighted graph G with a subset of its vertices identified as terminals, a graph G' composed of some terminal paths is called an MTST of G if the following are true: 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 Fig. 8(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 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 nonnegative weighted graph G, construct the MTST 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 nonterminal vertices that may or may not be present on the MTST, 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

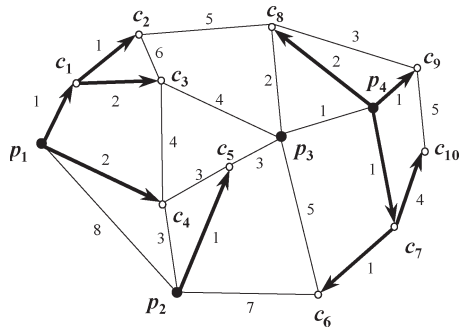

Fig. 9. Nonnegative weighted graph with terminal vertices $p_1 \sim p_4$ and nonterminal vertices $c_1 \sim c_{10}$. Its shortest path pin forest, which consists of four pin trees, is shown by the bold directed lines.

computed by Dijkstra's or Floyd–Washall algorithm. Then, they may apply either 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. Particularly in the algorithm of Lin *et al.*, 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 the algorithms of Lin *et al.* and Shen *et al.* 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 nonnegative weighted graph G, a directed subgraph of G is called a *terminal forest* on G if the following are true: 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 nonterminal 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 nonnegative weighted graph G and a terminal forest F on it, F is called the *shortest path terminal forest* if the following are true: 1) Each tree in F is the shortest path tree, and 2) for any vertex v, its root terminal is the nearest one among all the terminals on G.

Fig. 9 shows an example of a nonnegative weighted graph, where the black dots $p_1 \sim p_4$ are terminal vertices and hollow dots $c_1 \sim c_{10}$ are nonterminal 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 nonnegative 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. Moreover, 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 *intratree edge*.

In Fig. 9, 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 intratree edges.

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

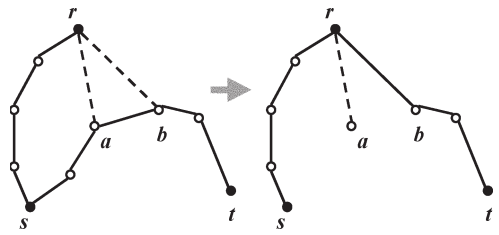

Fig. 10. Illustration of the proof of Lemma 3.

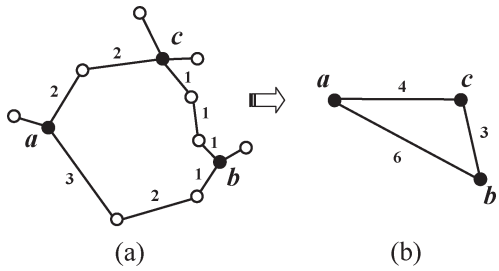

Fig. 11. Illustration of the proof of Lemma 4. (a) $G_{\rm fb}$, which consists of all the on-forest edges and bridge edges. (b) The corresponding $G'_{\rm fb}$.

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

Proof: Suppose that intratree edge $e(a, b)$ in Fig. 10 is part of the MTST path $\text{path}_{\text{MTST}}(s, t)$. Since $e(a, b)$ is an intratree edge, a and b should have a common root terminal r. We first remove path (s, b) from the MTST, and the MTSTs are divided into two components. Without lost of generality, we assume that 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 that this operation eliminates intratree edge (a, b) without introducing any new intratree edge into the MTST. Therefore, starting from any MTST, we can repeat the aforementioned process to obtain an MTST consisting of only bridge edges and on-forest edges. ■

Given a nonnegative weighted graph G and its subgraph $G_{\rm fb}$ 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 $G_{\rm fb}$ is the longest terminal path on a cycle on $G_{\rm fb}$, then there is at least one MTST of G that does not contain this terminal path.

Proof: For a given $G_{\rm fb}$, we construct a graph $G'_{\rm fb}$ whose vertex set contains only the terminal vertices of $G_{\rm fb}$. We add a weighted edge between two vertices of $G'_{\rm fb}$ if there is a terminal path between the two corresponding terminal vertices in $G_{\rm fb}$. The length of an edge in $G'_{\rm fb}$ is set to the length of the corresponding terminal path in $G_{\rm fb}$. Suppose that l is the longest terminal path on a cycle on $G_{\rm fb}$ and e' is the corresponding edge on $G'_{\rm fb}$. Then e' must be the longest edge on a cycle on $G'_{\rm fb}$. According to the cycle property, there should be at least one minimum spanning tree of $G'_{\rm fb}$ that does not contain e'. Since there is a one-to-one mapping between the edges of $G'_{\rm fb}$ and the terminal paths of $G_{\rm fb}$, we can directly map this minimum spanning tree to an MTST of $G_{\rm fb}$ (which is also an MTST of G) which does not contain l. ■

```
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
```

Fig. 12. Pseudocode of the extended Dijkstra's algorithm.

```
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();
    s_1 = Find-Set(u);
    s_2 = Find-Set(v);
    IF s_1 ≠ s_2 BEGIN
      Connect MTST edge e_MTST(u.root, v.root);
      s = Set-Union(s_1, s_2);
      s.edge = e_MTST;
      T_merg.merge(s, s_1, s_2);
    END
  END
END
```

Fig. 13. Pseudocode of the extended Kruskal's algorithm.

Fig. 11 shows the proof of Lemma 4. Fig. 11(a) gives an example of $G_{\rm fb}$, and Fig. 11(b) provides the corresponding $G'_{\rm fb}$. Apparently, the path between a and b is the longest path on the only cycle on $G_{\rm fb}$, and correspondingly, edge $e(a, b)$ is the longest edge on the only cycle on $G'_{\rm fb}$. According to cycle property, edge $e(a, b)$ is not included in the minimum spanning tree of $G'_{\rm fb}$. Correspondingly, the path between a and b is not part of the MTST of $G_{\rm fb}$.

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

Fig. 12 shows the pseudocode 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 $+\infty$. By 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 employed to record the root of each terminal tree.

Lemma 5: The extended Dijkstra's algorithm generates the shortest path terminal forest for any nonnegative weighted graph.

Proof: For a given nonnegative weighted graph G, we add one vertex v_0 to G, and add zero-weighted edge (v_0, u) to G for each terminal vertex u. We denote the newly obtained graph by G^*. If we set vertex v_0 to be the source vertex and apply the original Dijkstra's algorithm on G^*, it is obvious that the extended Dijkstra's algorithm will update the dist parameters of the nonterminal vertices on G in exactly the same way as the original Dijkstra's algorithm does on G^*. Hence, the final values of the dist parameters of the vertices set by the extended Dijkstra's algorithm are identical to the values set by the original Dijkstra's algorithm. On the other hand, for a given vertex v on G^*, its final dist value set by the original Dijkstra's algorithm is the length of the shortest path from v to v_0 and, therefore, the distance from v to the nearest terminal vertex, as the edges between the terminal vertices and v_0 all have zero weight. Therefore, the final value of the dist parameter of each vertex set by the extended Dijkstra's algorithm is the distance between this vertex and the nearest terminal vertex. ■

Now, we can present the extended Kruskal's algorithm for MTST construction. The pseudocode is shown in Fig. 13. 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 $G_{\rm fb}$, 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 $u.\text{dist} + e.\text{length} + v.\text{dist}$, where $u.\text{dist}$ and $v.\text{dist}$ record the distances of u and v to their root terminals, respectively, and have been computed previously by the extended Dijkstra's 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 [7]).

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

Proof: The correctness of the algorithm is guaranteed by Lemma 4 and Lemma 5. On the other hand, 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 applies to the extended Kruskal's algorithm. Therefore, the time complexity of MTST generation is $O(n \log n)$. ■

Note that when the vertices of the given nonnegative 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.

Note that Mehlhorn has proposed an MTST construction algorithm which shares similar ideas and has the same time complexity [25]. However, there are several differences between our technique and that of Mehlhorn in terms of implementation. For instance, Mehlhorn's implementation involves the construction of an auxiliary graph, while in our implementation, all the operations are performed in the OASG.

Finally, the spanning graphs used by Shen *et al.*, Lin *et al.*, and us, although having different definitions, are all instances of the nonnegative weighted graph. Thus, regarding the fact that the MTST generation step is the bottleneck of both schemes of Shen *et al.* and Lin *et al.*, our extended Dijkstra–Kruskal algorithm can be incorporated to speed them up.

C. OARST Construction

Having generated an 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 MTST. Hence, the improvement over the initial solution may be small [7].

Borah *et al.* [2] proposed an edge-substitution heuristic, a simple yet effective approach for Steiner tree refinement (on an obstacle-free plane). Zhou [7] observed that the geometrical proximity information embedded in the spanning graph could be leveraged to simplify the heuristic. 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.

Fig. 14 shows the enhanced edge-substitution technique. As defined earlier, an MTST path is a terminal path on the MTST. Furthermore, *subedges* of an MTST path refer to the OASG edges on this MTST path. For each subedge of each MTST path, we consider the vertex–edge pair formed by the subedge and each of its OASG neighboring vertices. An OASG vertex is called a neighboring vertex of a subedge if it is connected to either of the end points of the subedge. In Fig. 14, suppose u is a neighboring vertex of $e_{\rm sub}(a,b)$, we calculate the gain of vertex–edge pair $(u, e_{\rm sub})$ 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 Fig. 14). Suppose that $e_{\rm sub}$ and v are parts of MTST paths $\text{path}_{\rm MTST}$ and $\text{path}'_{\rm MTST}$, respectively. We will next find out the longest MTST path $\text{path}_{\rm longest}$ between $\text{path}_{\rm MTST}$ and $\text{path}'_{\rm MTST}$ ($\text{path}_{\rm MTST}$ and $\text{path}'_{\rm MTST}$ 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 $e_{\rm sub}$ and $\text{path}_{\rm longest}$ to maintain the tree topology. As the length

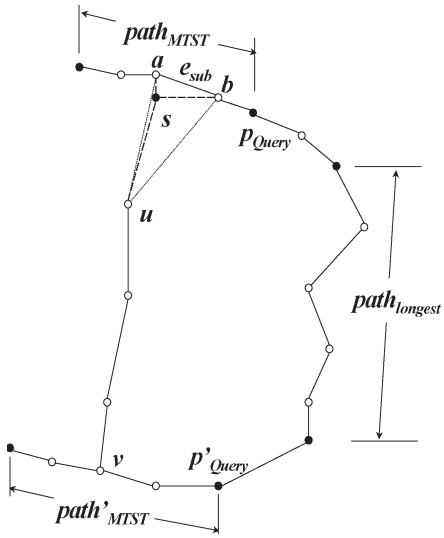

Fig. 14. Illustration of the edge-substitution heuristic.

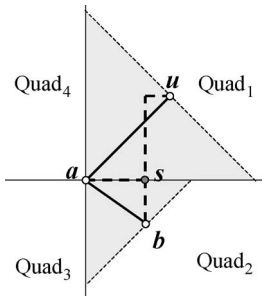

Fig. 15. Steiner connection of a vertex–edge pair.

of $e_{\rm sub}$ 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

$$\text{gain}(u, e_{\rm sub}) = \text{len}(\text{path}_{\rm longest}) - \text{len}\left(\text{path}(u,v)\right) - \text{len}\left((s,u)\right).$$

As the following lemma implies, $\text{len}((s,u))$ 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 subedge $e_{\rm sub}(a,b)$, 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 nonneighboring quadrants (e.g., Quad_1 and Quad_3), the problem becomes trivial since a is overlapping with s; second, as shown in Fig. 15, if u and b reside in two neighboring quadrants (e.g., 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 quadrants. Therefore, Manhattan connection from s to a, b, and u can be made as the dashed bold lines in Fig. 15. ■

The value $\text{len}(\text{path}(u,v))$ can be computed efficiently using a simple variant of the extended Dijkstra's algorithm proposed in Section IV-B. 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

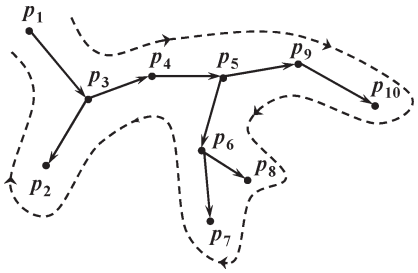

Fig. 16. Example of the Euler trail of a directed MTST.

```
ALGORITHM Edge-Substitution(MTST, T_merg)
INPUT: MTST // the minimum terminal spanning tree
       T_merg // merging tree of the MTST
OUTPUT: OARST
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
```

Fig. 17. Pseudocode for the edge-based Steiner tree refinement heuristic.

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 $(\text{path}_{\text{MTST}}, \text{path}'_{\text{MTST}})$. 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 the approach of Zhou [7]. 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 [26] offline least common ancestor algorithm can be used to find out the longest edges efficiently. Noticing we have only the path pair $(\text{path}_{\text{MTST}}, \text{path}'_{\text{MTST}})$ in hand, to exploit the binary merging tree, we need to transform this edge pair to a pin vertex pair $[(p_{\text{Query}}, p'_{\text{Query}})$ in Fig. 14]. 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.

Fig. 16 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 $\text{path}_{\text{MTST}}$ for the second time, we check all the path pairs involving $\text{path}_{\text{MTST}}$. Suppose that $(\text{path}_{\text{MTST}}, \text{path}'_{\text{MTST}})$ is such a pair. If $\text{path}'_{\text{MTST}}$ has been visited twice already, the vertex pair for $(\text{path}_{\text{MTST}}, \text{path}'_{\text{MTST}})$ will be the starting vertices of these two paths. If $\text{path}'_{\text{MTST}}$ has just been visited only once, the vertex pair for $(\text{path}_{\text{MTST}}, \text{path}'_{\text{MTST}})$ will consist of the ending vertex of $\text{path}'_{\text{MTST}}$ and the starting vertex of $\text{path}_{\text{MTST}}$. If $\text{path}'_{\text{MTST}}$ 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 $\text{path}_{\text{MTST}}$ for the second time, a path has been gone through just once if and only if it is an ancestor of $\text{path}_{\text{MTST}}$ in the directed MTST. Notice that there are at most $O(n)$ path pairs and that each path pair is to be checked twice. Moreover, 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 nondecreasing order of their gains. The whole edge substitution algorithm works in a "batched mode." In one pass, it first computes all the possible vertex–edge pairs with positive gain before applying any of the updates. Hence, an edge substitution can only be made if none of $\text{path}_{\text{MTST}}$, $\text{path}'_{\text{MTST}}$, and $\text{path}_{\text{longest}}$ has been modified. The pseudocode for the edge-based Steiner tree refinement heuristic is shown in Fig. 17.

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 reason that we have at most $O(n)$ vertex–edge pairs is that we connect a vertex with the nearest vertex in each quadrant. Hence, for a given edge on the OASG, there are no more than eight neighboring vertices. On the other hand, there are $O(n)$ edges; therefore, we need to consider 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 offline 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 edge-based refinement is still $O(n \log n)$.

D. Local Refinement

Although the edge substitution algorithm has effectively shortened the tree length, since it operates in a "batched mode," the resulting Steiner tree can still be further improved. To improve the solution quality, we perform some local refinements on the Steiner tree. Before doing the local refinements, we rectilinearize the Steiner tree by simply transforming each slant edge into a vertical edge and a horizontal edge.

The techniques we adopted for the local refinement is segment translation. A *segment* is composed of several chained edges that lie on the same line. We say that a segment s_1 is a neighbor of another segment s_2 if they intersect with each other. Fig. 18(a) shows the concept of segment where u_0u_4 is a segment consisting of edges u_0u_1, u_1u_2, u_2u_3, and u_3v_4. Its neighboring segments include u_0v_0, u_1v_1, and u_4v_3. Fig. 18(b) shows an example of segment translation. Lin *et al.*

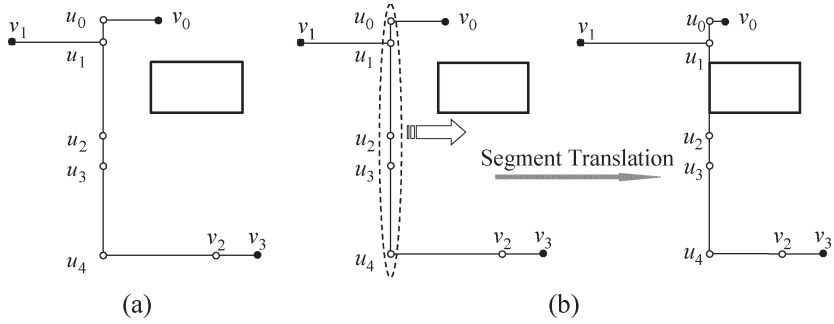

Fig. 18. (a) Examples of segments. (b) Illustration of segment translation.

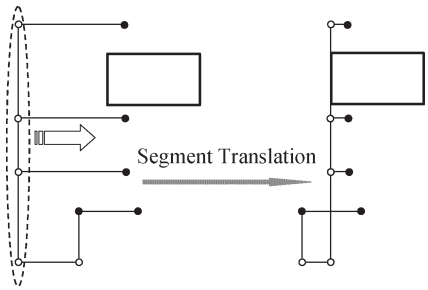

Fig. 19. Case where segment translation may lead to creation of cycles.

introduced a local refinement technique called the "U-shape pattern refinement." In fact, it can be viewed as a special case of segment translation. Moreover, Jariwala and Lilis proposed a global routing optimization technique call segment migration, which shares a similar idea [27]. However, their work does not involve the concept of obstacle avoidance. Therefore, the problem constraints and solutions are fundamentally different.

A segment translation is legitimate and beneficial if the following conditions are met: 1) The segment does not intersect with the obstacles after the translation; 2) no cycle is created after the translation; and 3) the segment has more neighboring segments on one side than the other side, and the segment is moved toward this side. Requirements 1) and 3) are easy to understand. Fig. 19 shows a case where Requirement 2) is violated after edge translation. To avoid the creation of cycles, we require that a segment does not pass over another segment parallel to it during translation.

We first dissect the OARST into disjoint segments by doing a tree traversal in $O(n)$ time. Then, it is easy to determine the number of its neighboring segments on each side [Requirement 1)]. On the other hand, we have developed an efficient sweeping line algorithm to calculate the maximum distance each segment can move [Requirements 2) and 3)].

Fig. 20 elaborates the sweeping line algorithm for calculating the maximum distance each vertical segment can move toward right. Other situations are symmetrical and can be handled in similar ways.

We first sort all the vertical segments and obstacles in nondecreasing order. The keys of the vertical segments (obstacles) for sorting are the x-coordinates of their lower end vertices (lower left corner vertices). During the sweeping, we maintain an active vertex set A_S. It consists of the vertical segments whose maximum allowable moving distances are still to be calculated. Initially, it is an empty set. We then scan the ordered elements

```
ALGORITHM Max-Mov-Dist(S_V, O)
INPUT: S_V // the set of the vertical segments
       O // the set of obstacles
OUTPUT: D_max // the maximal distance each vertical
                // segment can move towards right
BEGIN
  A_S = Φ;
  Sort all the elements in S_V ∪ O along x-axis;
  FOR EACH element elem in the order BEGIN
    FOR EACH segment seg in A_S whose projection
      on y-axis overlap with elem's BEGIN
      Calculate the horizontal distance
        between elem and seg;
      Remove seg from A_S;
    END
    IF elem is a segment BEGIN
      Add elem to A_S;
    END
  END
END
```

Fig. 20. Pseudocode of the sweeping line algorithm for calculating the maximal distance a vertical segment can be moved toward right.

(an element can be either a segment or an obstacle). For each element being scanned, we find out each segment in A_S such that its projection on y-axis overlaps with the projection of the element. We then calculate the difference between the key of each such segment and that of the element, which is the maximum allowable moving distance of that segment to the right. After that, each such segment is removed from A_S. If the currently scanned element is a segment, it will be added to A_S.

Notice that at any point during the sweeping process, the projections of the segments in A_S on y-axis do not overlap with each other, since the nearest obstacle/segment on their right is still to be determined. This property, again, allows us to exploit the binary search tree data structure to implement active set A_S efficiently. The query, insertion, and removal operation can be perform in $O(\log n)$ time. On the other hand, as we need to scan $O(n)$ elements, the time complexity of the maximum allowable moving distance calculation is $O(n \log n)$.

After calculating the maximum allowable moving distance for all the segments, the actual segment translation operation is straightforward and can be done in $O(n)$ time. Therefore, the time complexity of the local refinement step is $O(n \log n)$.

The following theorem gives the time and space complexity of our algorithm.

E. Time and Space Complexity Analysis

Theorem 2: Given m pin vertices and k rectangular obstacles on a plane, our algorithm generates an OARST in $O(n \log n)$ time using $O(n)$ storage elements, where $n = m + 4k$.

TABLE I
COMPARISON OF THE QUALITY OF THE OARSTS GENERATED BY DIFFERENT ALGORITHMS

Bench	m	k	Feng	Shen	Lin	Ours	$\Delta w\%$	Bench	m	k	Feng	Shen	Lin	Ours	$\Delta w\%$
IND01	10	32	—	646	632	639	-1.11%	RT04	100	1,000	—	10,629	10,459	10,580	-1.16%
IND02	10	43	—	10,100	9,700	10,000	-3.09%	RT05	200	2,000	—	55,535	54,683	55,286	-1.10%
IND03	10	50	—	623	623	623	0.00%	RL01	5,000	5,000	—	503,032	492,865	498,266	-1.10%
IND04	25	79	—	1,121	1,121	1,126	-0.45%	RL02	10,000	500	—	648,898	648,508	634,151	2.21%
IND05	33	71	—	1,392	1,392	1,379	0.93%	RL03	10,000	100	—	652,323	652,241	635,332	2.59%
RC01	10	10	30,410	27,730	27,790	27,540	0.90%	RL04	10,000	10	—	710,005	709,904	692,263	2.48%
RC02	30	10	45,640	42,840	42,240	41,930	0.73%	RL05	10,000	0	—	741,978	741,697	722,882	2.54%
RC03	50	10	58,570	56,440	56,140	54,180	3.49%	Adaptec2	100	566	—	113,500	111,169	111,202	-0.03%
RC04	70	10	63,340	60,840	60,800	59,050	2.88%		1,000	566	—	307,641	306,114	303,238	0.94%
RC05	100	10	83,150	76,970	76,760	75,630	1.47%	Adaptec4	100	1,329	—	192,095	185,950	190,971	-2.70%
RC06	100	500	149,750	86,403	84,193	86,381	-2.60%		1,000	1,329	—	556,700	544,615	551,025	-1.18%
RC07	200	500	181,470	117,427	114,173	117,093	-2.56%	Bigblue1	100	560	—	82,667	83,137	81,922	1.46%
RC08	200	800	202,741	123,366	120,492	122,306	-1.51%		1,000	560	—	253,921	253,632	248,902	1.86%
RC09	200	1,000	214,850	119,744	117,647	119,308	-1.41%	Bigblue2	100	23,084	—	164,767	163,720	166,340	-1.60%
RC10	500	100	198,010	171,450	171,519	167,978	2.06%		1,000	23,084	—	446,370	437,109	444,327	-1.65%
RC11	1,000	100	250,570	238,111	237,794	232,381	2.28%	Bigblue3	100	1,293	—	193,786	188,681	192,220	-1.88%
RC12	1,000	10,000	1,723,990	843,529	803,483	842,689	-4.88%		1,000	1,293	—	522,539	517,878	513,186	0.91%
RT01	10	500	—	2,438	2,289	2,362	-3.19%	Bigblue4	100	8,170	—	268,443	264,393	271,693	-2.76%
RT02	50	500	—	51,987	48,858	52,218	-6.88%		1,000	8,170	—	790,142	766,804	790,708	-3.12%
RT03	100	500	—	8,783	8,508	8,645	-1.61%	Average							-0.46%

TABLE II
BREAKDOWN OF THE IMPROVEMENTS MADE BY THE GLOBAL REFINEMENT AND LOCAL REFINEMENT STEPS

Bench	m	k	MTST	GlbRef	*impr%*	LocRef	*impr%*	Bench	m	k	MTST	GlbRef	*impr%*	LocRef	*impr%*
IND01	10	32	669	649	2.99%	639	1.56%	RT04	100	1,000	11,333	10,656	5.97%	10,580	0.72%
IND02	10	43	10,501	10,201	2.86%	10,000	2.01%	RT05	200	2,000	59,603	56,427	5.33%	55,286	2.06%
IND03	10	50	633	623	1.58%	623	0.00%	RL01	5,000	5,000	550,554	504,887	8.29%	498,266	1.33%
IND04	25	79	1,195	1,131	5.36%	1,126	0.44%	RL02	10,000	500	709,275	641,445	9.56%	634,151	1.15%
IND05	33	71	1,453	1,379	5.09%	1,379	0.00%	RL03	10,000	100	712,434	644,616	9.52%	635,332	1.46%
RC01	10	10	30,269	27,540	9.02%	27,540	0.00%	RL04	10,000	10	776,605	701,088	9.72%	692,263	1.27%
RC02	30	10	45,500	42,030	7.63%	41,930	0.24%	RL05	10,000	0	810,186	731,790	9.68%	722,882	1.23%
RC03	50	10	60,139	56,230	6.50%	54,180	3.78%	Adaptec2	100	566	121,195	111,659	7.87%	111,202	0.41%
RC04	70	10	65,490	59,540	9.09%	59,050	0.83%		1,000	566	332,857	306,049	8.05%	303,238	0.93%
RC05	100	10	83,890	76,329	9.01%	75,630	0.92%	Adaptec4	100	1,329	202,652	193,956	4.29%	190,971	1.56%
RC06	100	500	93,642	87,513	6.55%	86,381	1.31%		1,000	1,329	605,633	556,800	8.06%	551,025	1.05%
RC07	200	500	126,905	118,144	6.90%	117,093	0.90%	Bigblue1	100	560	90,397	82,236	9.03%	81,922	0.38%
RC08	200	800	133,301	124,850	6.34%	122,306	2.08%		1,000	560	276,718	251,531	9.10%	248,902	1.06%
RC09	200	1,000	128,652	120,629	6.24%	119,308	1.11%	Bigblue2	100	23,084	171,642	167,248	2.56%	166,340	0.55%
RC10	500	100	186,619	169,129	9.37%	167,978	0.69%		1,000	23,084	479,678	448,756	6.45%	444,327	1.00%
RC11	1,000	100	261,778	235,679	9.97%	232,381	1.42%	Bigblue3	100	1,293	203,435	193,338	4.96%	192,220	0.58%
RC12	1,000	10,000	905,693	852,126	5.91%	842,689	1.12%		1,000	1,293	563,429	519,393	7.82%	513,186	1.21%
RT01	10	500	2,705	2,414	10.76%	2,362	2.20%	Bigblue4	100	8,170	282,613	276,241	2.25%	271,693	1.67%
RT02	50	500	56,185	52,410	6.72%	52,218	0.37%		1,000	8,170	851,405	805,077	5.44%	790,708	1.82%
RT03	100	500	9,423	8,697	7.70%	8,645	0.60%	Average					6.91%		1.10%

Proof: We have analyzed the four phases of the algorithm (OASG generation, MTST construction, edge-based refinement, and local refinement), and their time complexities are all $O(n \log n)$. Therefore, the overall time complexity of our OARST algorithm is $O(n \log n)$.

The data structure of our algorithm includes the active sets, the heaps, the binary merging trees, the OASG, the OARMST, and the OARST. The cardinalities of these data structures are all proportional to the number of the vertices. Therefore, the space complexity of our algorithm is $O(n)$. ■

V. EXPERIMENTAL RESULTS

In this section, we provide the experimental results on several commonly used test cases [8], [10], [24], [28]. 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 2.1-GHz Dual Core AMD Opteron processors and 2-GB memory.

We compared our results with those of Feng *et al.*, Shen *et al.*, and Lin *et al.* We executed the algorithms of Shen *et al.* and Lin *et al.* on our platform. The results of Feng *et al.* are quoted from their paper, where their algorithm was tested on a Sun V800 fire workstation with a 755-MHz CPU and 4-GB memory [8]. Comparison of the quality of the Steiner trees generated by the four algorithms is provided in Table I. Benchmarks IND01∼IND05, RC01∼RC12, RT01∼RT05, and RL01∼RL05 are test cases used in previous works [8], [10], [11], [24]. Benchmarks Adaptec2∼Bigblue4 are based on the placement benchmarks used in ISPD 2005 Placement Contest [28]. The original benchmarks used in ISPD 2005 Placement Contest contain both fixed macroblocks and movable standard cells. In our experiment, for each placement contest benchmark, we extracted the fixed macroblocks as the obstacles. For each placement contest benchmark, we experimented with two sets of pins, where one includes 100 pins and the other contains 1000 pins. The locations of the pins are randomly generated.

TABLE III
COMPARISON OF THE RUNNING TIME OF DIFFERENT ALGORITHMS

Bench	m	k	Feng	Shen	Lin	Ours	*speedup*	Bench	m	k	Feng	Shen	Lin	Ours	*speedup*
IND01	10	32	—	0.01	0.01	0.01	1.00 x	**RT04**	100	1,000	—	0.35	0.42	0.38	1.11 x
IND02	10	43	—	0.01	0.01	0.01	1.00 x	**RT05**	200	2,000	—	1.61	1.43	1.06	1.35 x
IND03	10	50	—	0.01	0.01	0.01	1.00 x	**RL01**	5,000	5,000	—	138.2	146.12	5.18	28.21 x
IND04	25	79	—	0.02	0.02	0.02	1.00 x	**RL02**	10,000	500	—	140.26	217.39	2.24	97.05 x
IND05	33	71	—	0.02	0.02	0.02	1.00 x	**RL03**	10,000	100	—	123.61	202.69	1.97	102.89 x
RC01	10	10	0.01	0.01	0.01	0.01	1.00 x	**RL04**	10,000	10	—	123.98	254.37	1.84	138.24 x
RC02	30	10	0.01	0.02	0.01	0.01	1.00 x	**RL05**	10,000	0	—	127.97	282.59	1.87	151.12 x
RC03	50	10	0.01	0.02	0.01	0.01	1.00 x	**Adaptec2**	100	566	—	0.2	0.21	0.08	2.63 x
RC04	70	10	0.01	0.02	0.02	0.02	1.00 x		1,000	566	—	2.25	2.23	0.19	11.74 x
RC05	100	10	0.01	0.03	0.02	0.02	1.00 x	**Adaptec4**	100	1,329	—	0.56	0.64	0.35	1.83 x
RC06	100	500	0.06	0.22	0.16	0.13	1.23 x		1,000	1,329	—	4.49	4.79	0.55	8.71 x
RC07	200	500	0.06	0.37	0.3	0.15	2.00 x	**Bigblue1**	100	560	—	0.19	0.24	0.06	4.00 x
RC08	200	800	0.1	0.52	0.47	0.27	1.74 x		1,000	560	—	2.19	2.63	0.18	14.61 x
RC09	200	1,000	0.13	0.71	0.6	0.36	1.67 x	**Bigblue2**	100	2,3084	—	39	70.55	16.61	4.25 x
RC10	500	100	0.03	0.33	0.32	0.08	4.00 x		1,000	2,3084	—	124.07	170.47	17.89	9.53 x
RC11	1,000	100	0.04	1.11	1.26	0.14	9.00 x	**Bigblue3**	100	1,293	—	0.5	0.56	0.32	1.75 x
RC12	1,000	10,000	2.82	63.79	82.56	5.88	14.04 x		1,000	1,293	—	4.26	4.74	0.52	9.12 x
RT01	10	500	—	0.09	0.06	0.12	0.50 x	**Bigblue4**	100	8,170	—	8.36	13.37	4.76	2.81 x
RT02	50	500	—	0.16	0.11	0.11	1.00 x		1,000	8,170	—	38.01	44.58	5.24	8.51 x
RT03	100	500	—	0.22	0.17	0.13	1.31 x	**Average**							16.56 x

TABLE IV
RUNTIME OVERHEAD OF THE LOCAL REFINEMENT STEP

Bench	t_{LocRef}	t_{tot}	*LocRef%*	Bench	t_{LocRef}	t_{tot}	*LocRef%*
IND01	0.00	0.01	0.00%	**RT04**	0.01	0. 38	2.63%
IND02	0.00	0.01	0.00%	**RT05**	0.02	1. 06	1.89%
IND03	0.00	0.01	0.00%	**RL01**	0.06	5. 18	1.16%
IND04	0.00	0.02	0.00%	**RL02**	0.03	2. 24	1.34%
IND05	0.00	0.02	0.00%	**RL03**	0.02	1. 97	1.02%
RC01	0.00	0.01	0.00%	**RL04**	0.02	1. 84	1.09%
RC02	0.00	0.01	0.00%	**RL05**	0.02	1. 87	1.07%
RC03	0.00	0.01	0.00%	**Adaptec2**	0.00	0.08	0.00%
RC04	0.00	0.02	0.00%		0.00	0. 19	0.00%
RC05	0.00	0.02	0.00%	**Adaptec4**	0.01	0.35	2.86%
RC06	0.00	0.13	0.00%		0.02	0. 55	3.64%
RC07	0.00	0.15	0.00%	**Bigblue1**	0.00	0.06	0.00%
RC08	0.00	0.27	0.00%		0.00	0. 18	0.00%
RC09	0.01	0.36	2.78%	**Bigblue2**	0.16	16.61	0.96%
RC10	0.00	0.08	0.00%		0.16	17.89	0.89%
RC11	0.00	0.14	0.00%	**Bigblue3**	0.01	0.32	3.13%
RC12	0.05	5.88	0.85%		0.01	0. 52	1.92%
RT01	0.00	0.12	0.00%	**Bigblue4**	0.03	4.76	0.63%
RT02	0.00	0.11	0.00%		0.04	5. 24	0.76%
RT03	0.00	0.13	0.00%	**Average**			0.73%

Column "$\Delta w\%$" provides the relative improvement of our OARSTs over that of Lin *et al.* and is calculated by

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

First, we observe that compared to the algorithm of Feng *et al.*, our algorithm performs consistently better in terms of OARST quality. Particularly for the benchmarks with large k/m ratio (RC06, RC07, RC08, RC09, and RC12), our algorithm produces OARSTs with substantially smaller length. For instance, for RC12, the length of our OARST is less than half that of Feng *et al.* Compared to the algorithm of Shen *et al.*, our algorithm produces Steiner trees with higher quality in most cases (except for IND04, RT02, Bigblue2 with 100 pins, Bigblue4 with 100 pins, and Bigblue4 with 1000 pins).

Second, we also observed that our algorithm produces better OARSTs when the ratio k/m is less than one. For example, for test cases RC02, RC03, RC04, RC05, RC10, RC11, Adaptec2 with 1000 pins, and Bigblue1 with 1000 pins, our OARST has smaller length than those of Lin *et al.* Furthermore, experimental results for the six large benchmarks RL01∼RL05 reveal that as k/m approaches zero, our algorithm performs better in terms of solution quality. In the limiting case (RL05), 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.

In terms of quality of the OARST, our algorithm performs comparable to that of Lin *et al.* On average, our OARSTs are only 0.46% longer than those of Lin *et al.*

Table II provides the breakdown of the improvement made by the global refinement technique—edge-based heuristic and the local refinement technique—segment translation. Column "MTST" contains the total length of the MTST for each

benchmark. Column "GlbRef" presents the lengths of the refined Steiner trees after applying the edge-based heuristics, and the next column "impr%" is the relative improvement in total tree length compared to MTST. Similarly, column "LocRef" and the next "impr%" give the total tree length and the relative improvement gained by applying segment translation. We observe that the relative improvement made by the edge-based heuristic ranges from 1.58% to 10.76%, while the relative improvement made by segment translation varies between 0.00% and 2.20%. On average, edge-based heuristic and segment translation gain 6.91% and 1.10% improvement, respectively.

We provide the running time of the different algorithms in Table III. The unit of the running time is in seconds. Column "speedup" compares the execution time of our algorithm and that of Lin *et al.* It is calculated by

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

Compared to the algorithms of Shen *et al.* and Lin *et al.*, our algorithm terminates in shorter time, particularly for the large benchmarks (RC11, RC12, RL01∼RL05, Adaptec2 with 1000 pins, Bigblue2, and Bigblue4). For all the test cases, on average, our algorithm runs 16.56 times faster than that of Lin *et al.*

Finally, we provide the runtime overhead of the newly introduced local refinement step for each benchmark in Table IV. The column "t_{LocRef}" gives the runtime of the local refinement step in seconds. The same table also includes the total runtime (column "t_{tot}") for each benchmark. Column "LocRef%" compares the runtime of the local refinement step with the total runtime. It is calculated by

$$\text{LocRef}\% = (t_{\text{LocRef}}/t_{\text{tot}}) \times 100\%.$$

Obviously, the runtime overhead for the local refinement step is negligible. On average, the runtime of the local refinement step only consists of 0.73% of the total runtime.

VI. Conclusion

In this paper, we have presented EBOARST, an efficient four-step algorithm for OARST construction. We devise a novel algorithm to efficiently generate the OASG and the MTST. We also incorporate an edge-based global refinement technique as well as a local refinement technique call "segment translation" into our scheme. Experimental results indicate that our approach is an efficient yet effective approach for OARST construction. Compared to the heuristic of Lin *et al.*, our algorithm achieves 16.56 times speedup on average, while the length of the resulting OARSTs is only 0.46% larger on average.

References

[1] M. Hanan, "On Steiner's problem with rectilinear distance," *SIAM J. Appl. Math.*, vol. 14, no. 2, pp. 255–265, Mar. 1966.

[2] M. Borah, R. M. Owens, and M. J. Irwin, "An edge-based heuristic for Steiner routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 12, pp. 1563–1568, Dec. 1994.

[3] C. Chu and Y. Wong, "Fast and accurate rectilinear Steiner minimal tree algorithm for VLSI design," in *Proc. Int. Symp. Phys. Des.*, 2005, pp. 28–35.

[4] M. Garey and D. Johnson, "The rectilinear Steiner tree problem is NP-complete," *SIAM J. Appl. Math.*, vol. 32, no. 4, pp. 826–834, Jun. 1977.

[5] J. Griffith *et al.*, "Closing the gap: Near-optimal Steiner trees in polynomial time," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 11, pp. 1351–1365, Nov. 1994.

[6] I. I. Mandoiu, V. V. Vazirani, and J. L. Ganley, "A new heuristic for rectilinear Steiner trees," in *Proc. Int. Conf. Comput.-Aided Des.*, 1999, pp. 157–162.

[7] H. Zhou, "Efficient Steiner tree construction based on spanning graphs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 5, pp. 704–710, May 2004.

[8] Z. Feng *et al.*, "An $O(n \log n)$ algorithm for obstacle-avoiding routing tree construction in the lambda-geometry plane," in *Proc. Int. Symp. Phys. Des.*, 2006, pp. 127–134.

[9] J. L. Ganley and J. P. Cohoon, "Routing a multi-terminal critical net: Steiner tree construction in the presence of obstacles," in *Proc. Int. Symp. Circuits Syst.*, 1994, pp. 113–116.

[10] C. Lin *et al.*, "Efficient obstacle-avoiding rectilinear Steiner tree construction," in *Proc. Int. Symp. Phys. Des.*, 2007, pp. 127–134.

[11] Z. Shen, C. Chu, and Y. Li, "Efficient rectilinear Steiner tree construction with rectilinear blockages," in *Proc. Int. Conf. Comput. Des.*, 2005, pp. 38–44.

[12] Y. Shi *et al.*, "Circuit simulation based obstacle-aware Steiner routing," in *Proc. Des. Autom. Conf.*, 2006, pp. 385–388.

[13] Y. Yang *et al.*, "Rectilinear Steiner minimal tree among obstacles," in *Proc. Int. Conf. ASIC*, 2003, pp. 348–351.

[14] C. Lin *et al.*, "Efficient multi-layer obstacle-avoiding rectilinear Steiner tree construction," in *Proc. Int. Conf. Comput.-Aided Des.*, 2007, pp. 380–385.

[15] C. Liu *et al.*, "Efficient multilayer routing based on obstacle-avoiding preferred direction Steiner tree," in *Proc. Int. Symp. Phys. Des.*, 2008, pp. 118–125.

[16] C. Lin *et al.*, "Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 4, pp. 643–653, Apr. 2008.

[17] M. Pan and C. Chu, "FastRoute: A step to integrate global routing into placement," in *Proc. Int. Conf. Comput.-Aided Des.*, 2006, pp. 464–471.

[18] S. B. Akers, "A modification of Lee's path connection algorithm," *IEEE Trans. Electron. Comput.*, vol. EC-16, no. 1, pp. 97–98, Feb. 1967.

[19] J. Soukup, "Fast maze router," in *Proc. Des. Autom. Conf.*, 1978, pp. 100–102.

[20] F. O. Hadlock, "A shortest path algorithm for grid graphs," *Networks*, vol. 7, no. 4, pp. 323–334, 1977.

[21] F. Rubin, "The Lee path connection algorithm," *IEEE Trans. Comput.*, vol. C-23, no. 9, pp. 907–914, Sep. 1974.

[22] D. W. Hightower, "A solution to the line routing problem on the continuous plane," in *Proc. Des. Autom. Conf.*, 1969, pp. 1–24.

[23] K. Mikami and K. Tabuchi, "A computer program for optimal routing of printed circuit connectors," in *Proc. IFIPS*, 1968, pp. 1475–1478.

[24] J. Long, H. Zhou, and S. O. Memik, "An $O(n \log n)$ edge-based algorithm for obstacle-avoiding rectilinear Steiner tree construction," in *Proc. Int. Symp. Phys. Des.*, 2008, pp. 126–133.

[25] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 125–128, Mar. 1988.

[26] R. Tarjan, "Applications of path compression on balanced trees," *J. ACM*, vol. 26, no. 4, pp. 690–715, Oct. 1979.

[27] D. Jariwala and J. Lilis, "Trunk decomposition based global routing optimization," in *Proc. Int. Conf. Comput.-Aided Des.*, 2006, pp. 472–479.

[28] *ISPD 2005 Placement Contest Benchmark Suite*, 2005. [Online]. Available: http://www.sigda.org/ispd2005/contest.htm

Jieyi Long (S'07) received the B.S. degree in microelectronics from Peking University, Beijing, China, in 2006. He is currently working toward the Ph.D. degree in the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL.

His research interests include physical design, thermal monitoring and management for high-performance VLSI systems, and design automation for self-adjusting architectures.

Hai Zhou (M'04–SM'04) received the B.S. and M.S. degrees in computer science and technology from Tsinghua University, Beijing, China, in 1992 and 1994, respectively, and the Ph.D. degree in computer science from the University of Texas, Austin, in 1999.

He is currently an Associate Professor in electrical engineering and computer science with Northwestern University, Evanston, IL. His research interests include VLSI computer-aided design, algorithm design, and formal methods.

Dr. Zhou was the recipient of CAREER Award from the National Science Foundation in 2003.

Seda Ogrenci Memik (SM'05) received the B.S. degree in electrical and electronic engineering from Bogazici University, Istanbul, Turkey and the Ph.D. degree in computer science from the University of California, Los Angeles.

She is currently an Assistant Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL. Her research interests include embedded and reconfigurable computing, thermal-aware design automation, and thermal management for high-performance microprocessor systems.

Dr. Memik has served as a technical program committee member, organizing committee member, and subcommittee Chair of several conferences, including ICCAD, DATE, FPL, GLSVLSI, and ARC. She was the recipient of the National Science Foundation Early Career Development (CAREER) Award in 2006.