FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction

Gaurav Ajwani, *Member, IEEE,* Chris Chu, and Wai-Kei Mak, *Member, IEEE*

***Abstract*—In this paper, we present an algorithm called FOARS for obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) construction. FOARS applies a top-down approach which first partitions the set of pins into several subsets uncluttered by obstacles. Then an obstacle-avoiding Steiner tree is generated for each subset by an obstacle aware version of the rectilinear Steiner minimal tree algorithm FLUTE. Finally, the trees are merged and refined to form the OARSMT. To guide the partitioning of pins, we propose a novel algorithm to construct a linear-sized obstacle-avoiding spanning graph which guarantees to contain a rectilinear minimum spanning tree if there is no obstacle. Experimental results show that FOARS is among the best algorithms in terms of both wirelength and runtime for testcases both with and without obstacles.**

***Index Terms*—Physical design, rectilinear Steiner minimal tree (RSMT), routing, spanning graph.**

I. Introduction

WITH THE ADVENT of re-usability using intellectual property (IP) sharing, the chip in today's design is completely packed with fixed blocks such as IP blocks, macros, and so on. Routing of multi-terminal nets in the presence of obstacles has become a quintessential part of the design and has been studied by many (e.g., [1]–[13]). As pointed out by Hwang [14], in the absence of obstacles multi-terminal net routing corresponds to the rectilinear Steiner minimal tree (RSMT) problem which is NP-complete. The presence of obstacles in the region makes multi-terminal routing problem even harder.

In this paper, we develop a new algorithm called FOARS for OARSMT and RSMT generation by leveraging FLUTE [15]. FLUTE is a very fast and robust tool for RSMT generation. It is widely used in many recent academic physical design tools. FLUTE by its design cannot handle obstacles. A simple strategy to generate an OARSMT would be to call FLUTE once and legalize the edges intersecting with obstacles. Unfortunately, the OARSMT obtained can be far from optimal as its topology is based on an obstacle-oblivious Steiner tree. A better strategy is to break the RSMT produced by FLUTE on edges overlapping with obstacles, recursively call FLUTE to locally optimize the subtrees, and then combine all overlap-free subtrees at the end. However, if the routing region is severely cluttered with obstacles, the quality of the solution produced will degrade because the RSMTs generated by FLUTE may be excessively broken. To tackle this, we propose a partitioning algorithm with a global view of the problem at the top level to divide the problem into smaller uncluttered instances. Even if there is no obstacle, when the number of pins are more than several tens, the partitioning algorithm can improve both the wirelength and runtime of FLUTE as it works better than the greedy net breaking heuristics in FLUTE.

To guide the partitioning algorithm, we propose to use a sparse spanning graph. In the presence of obstacles, this graph will be an obstacle-avoiding spanning graph (OASG). An OASG is used to capture the proximity information among the pins and corners of obstacles, if any. Three categories of graph were used to capture the proximity information during OARSMT construction in the past. References [1], [3], [4], and [10] all used the escape graph. Reference [9] utilized a Delaunay triangulation based graph. Both the escape graph and Delaunay triangulation based graph contain $O(n^2)$ edges, where n is the total number of pins and obstacle corners. References [2] and [5]–[8] are based on various forms of obstacle-avoiding spanning graphs. Shen *et al.* [2] proposed a form of OASG that only contains a linear number of edges which is also adopted in [5]. Later Lin *et al.* [6] proposed adding missing "essential edges" to Shen's OASG. Unfortunately, it increases the number of edges to $O(n^2)$ in the worst case ($O(n \log n)$ in practice) and hence the time complexity of later steps of OARSMT construction is increased to a large extent. In view of that, Long *et al.* [7], [8] proposed a quadrant approach to generate an OASG with a linear number of edges. But as we will see later, the OASG generated by Long's approach is not ideal. In this paper, we present a novel octant approach to generate an $O(n)$-edge OASG with more desirable properties.

Different from [2] and [6]–[8] which directly use an OASG to construct an OARSMT, we only use an OASG to guide the partitioning and construct our final OARSMT using FLUTE. We note that a shortcoming of the former approach is that the resulting OARSMT tends to follow obstacle boundaries and makes detours toward obstacle corners. This makes it easier to lead to congestion when routing many nets in a design.

Manuscript received June 12, 2010; revised August 17, 2010; accepted October 20, 2010. Date of current version January 19, 2011. This work was supported in part by NSC, under Grant NSC 99-2220-E-0007-007. This paper was recommended by Associate Editor Y.-W. Chang.

G. Ajwani is with Intel, Hillsboro, OR 97124 USA (e-mail: gajwani@iastate.edu).

C. Chu is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011 USA.

W.-K. Mak is with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2096571

(Adding essential edges as in [6] will help but will result in $O(n^2)$ edges as an escape graph.) On the other hand, since we only utilize the OASG to guide our partitioning and use FLUTE for local optimization, the OARSMT thus constructed will follow an obstacle boundary only when absolutely necessary. In addition, the OASG generated by our proposed octant approach has a linear number of edges like Long's [7], [8] and possesses other desirable properties not found in Long's OASG. For example, our OASG is guaranteed to contain at least one rectilinear minimum spanning tree in the absence of obstacle while Long's OASG does not have such a guarantee.

In this paper, we also propose an obstacle tree data structure to accelerate the checking of overlap with obstacles. With the aid of the obstacle tree data structure, the runtime of FOARS is reduced by 59% as compared with [16].

Also, in this paper we would like to bring to notice that in [16], we made a mistake in the run time analysis and incorrectly claimed that the time complexity is $O(n \log n)$. For some extreme cases, it can take $O(n^2)$ time. But our experimental results indicate that it is extremely efficient for all practical purposes.

We compared our results with the state-of-the-art OARSMT and RSMT algorithms. Our results show that FOARS is among the best in terms of both wirelength and runtime for testcases both with and without obstacles and especially for large testcases.

The rest of this paper is organized as follows. We first provide an overview of the main steps of our OARSMT construction approach in Section II. Each main step is described in detail in Sections III–VII. The experimental results are reported in Section VIII. Finally, we give our conclusion in Section IX.

II. Overview of FOARS

Our algorithm can be distinctly divided into the following five stages.

1) *Stage 1: OASG generation*. First, we obtain the connectivity information between the pins and obstacle corner vertices using a novel octant OASG generation algorithm. Section III describes the OASG algorithm in detail.
2) *Stage 2: OPMST generation*. Based on the OASG, we construct a minimum terminal spanning tree (MTST) using the approach mentioned in [17] and then obtain an obstacle penalized minimal spanning tree (OPMST) from the MTST. Section IV talks about OPMST construction in detail.
3) *Stage 3: OAST generation*. We partition the pin vertices based on the OPMST constructed in the previous step. After partitioning, we pass the subproblems to OA-FLUTE which calls FLUTE recursively to construct an obstacle-aware Steiner tree (OAST). Section V talks about the partitioning and OA-FLUTE in more detail.
4) *Stage 4: OARSMT generation*. In this step, we rectilinearize the pin-to-pin connections avoiding obstacles to construct an OARSMT. Section VI discusses OARSMT construction.

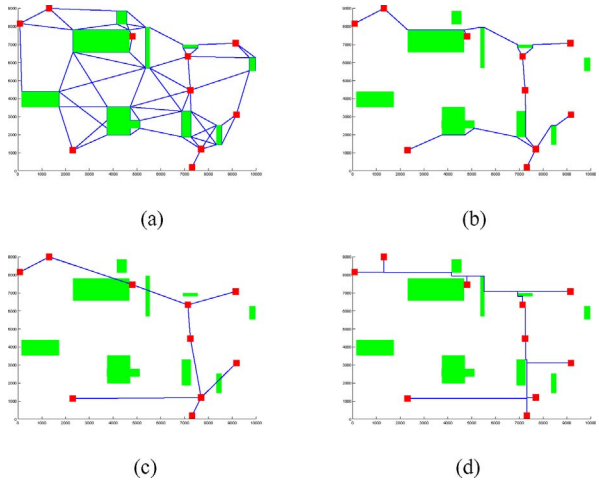


Fig. 1. Demonstration of the major steps of FOARS using the benchmark RC01. (a) OASG. (b) MTST. (c) OPMST. (d) OARSMT.

5) *Stage 5: refinement*. To further reduce the wirelength, we perform V-shape refinement on the OARSMT. Details for it can be found in Section VII.

Fig. 1 depicts the outputs after various stages of the algorithm.

III. OASG Generation

A. Previous Approaches

We first define what we meant by an obstacle-avoiding spanning graph.

Definition 1: *Given an edge $e(u, v)$ and an obstacle b, e is completely blocked by b if every monotonic Manhattan path connecting u and v intersects with a boundary of b.*

Definition 2: *Given a set of m pins and k obstacles, an undirected graph $G = (V, E)$ connecting all pin and corner vertices is called an OASG if none of its edges is completely blocked by an obstacle.*

Although Definition 2 does not necessitate a linear number of edges for an OASG, in order to have a fast runtime it is desired to limit the solution space. In the past, there have been a couple of efforts to construct an OASG with a linear number of edges. Shen *et al.* [2] suggested a quadrant approach in which each point can connect in four quadrants in the plane formed by horizontal and vertical line going through the point. Shen did not clearly explain their algorithm in the paper.

Long *et al.* [7] recently described an $O(n \log n)$-time approach for OASG generation with a linear number of edges by considering quadrant partition of the plane. They suggested scanning along $\pm 45°$ lines and maintaining an *active vertex list*, a set of vertices in the graph which are not yet connected to their nearest neighbor. After scanning any vertex v, they searched for its nearest neighbor u in the active vertex list, such that the edge (u, v) is not completely blocked by any obstacle in the graph. This is followed by deletion of u from the list and addition of v in the list.

We found that the OASG generation algorithm in [7] has a few shortcomings. First in their algorithm, the nearest neighbor for any vertex in a quadrant is contingent upon the direction of scanning which means they have to scan along all four

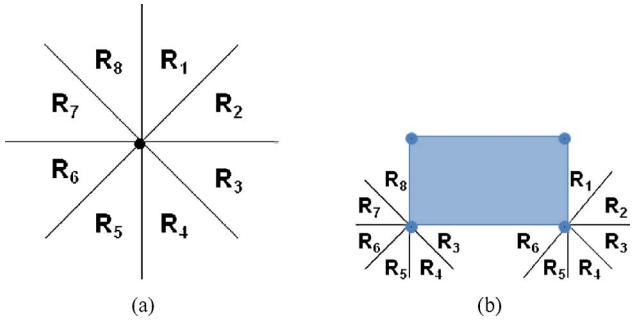


Fig. 2. Octant partition for (a) a pin vertex and (b) an obstacle corner.

quadrants of a vertex in order to capture its connectivity information. Second, in the absence of obstacles, their algorithm cannot guarantee the presence of at least one minimum spanning tree in their spanning graph. Third, their algorithm cannot handle abutting obstacles due to minor mistakes in the inequality conditions.

B. Our Approach for OASG

In [18], Zhou *et al.* proposed an elegant algorithm to generate a spanning graph with a linear number of edges guaranteed to contain a minimum rectilinear spanning tree on an obstacle-free plane by considering octant partition of the plane. So, we also propose an OASG algorithm based on octant partition. Fig. 2(a) and (b) describes octant partition for a pin vertex and an obstacle corner, respectively.

A property of octant partition is that a contour of equidistant points from any point forms a *line segment* in each region. In regions R_1, R_2, R_5, R_6, these segments are captured by an equation of the form $x + y = c$ for some constant c; in regions R_3, R_4, R_7, R_8, they are described by the equation $x - y = c$ for some constant c. Now this property can be exploited when we generate an obstacle-avoiding spanning graph.

The pseudocode for OASG generation for R_1 is provided in Fig. 3. As R_1 and R_2 both follow the same sweep sequence we process them together in one pass. It is worth noting that it is sufficient to sweep for R_1, R_2, R_3, and R_4 only. For any point, we only need to sweep twice to determine its connectivity information once for R_1/R_2 and once for R_3/R_4.

For octants R_1 and R_2, we sweep on a list of vertices in V which contains both pins as well as obstacle corners with respect to increasing $(x+y)$. During sweeping we maintain an active vertex list A_{active}. An active vertex is a vertex whose nearest neighbor in R_1 still needs to be discovered.

For the currently scanned vertex v, while looking in R_5 of v we extract a subset $S(v)$ from A_{active}. Any node u in this subset $S(v)$ has v in R_1 (lines 3–6). We connect v to its nearest neighbor u^* in $S(v)$ for which, $e(u^*, v)$ is not completely blocked (line 7). After connecting with the nearest point we delete all the points in $S(v)$ from A_{active} (line 8) and add v to A_{active} (line 18).

In order to determine if an edge is blocked by an obstacle, we maintain two active obstacle boundary lists, A_{bottom} for the bottom boundaries and A_{left} for the left boundaries. It is evident that if an edge is blocked by an obstacle in R_1, it will intersect with either its bottom or its left boundary. Next, if

***Algorithm:* OASG Generation for R_1**

```
1  A_active = A_bottom = A_left = ∅
2  for all v ∈ V in increasing (x + y) order do
3     S(v) = ∅
4     for all u ∈ A_active which have v in their R1 do
5        Add u to S(v)
6     end for
7     Connect v to the nearest point u* ∈ S(v) such that
        e(u*, v) is not completely blocked
        by obstacle boundaries in A_bottom and A_left
8     Delete all points in S(v) from A_active
9     if v is a bottom left corner then
10       Add the bottom boundary containing v to A_bottom
           and the left boundary containing v to A_left
11    else if v is a top left corner then
12       Delete the left boundary containing v from A_left
13    else if v is a bottom right corner then
14       Determine the bottom boundary B containing v
15       Delete B from A_bottom
16       Delete from A_active all points which are
           completely blocked by B
17    end if
18    Add v to A_active
19 end for
```

Fig. 3. Pseudocode for OASG generation algorithm.

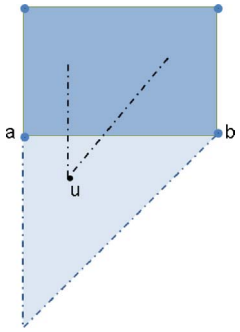


Fig. 4. Any vertex within the lightly shaded triangle is completely blocked by boundary (a, b).

our scanned vertex is the bottom left corner of an obstacle, its bottom boundary is added to A_{bottom} and its left boundary is added to A_{left}. It implies that both the left and the bottom boundaries of that obstacle become active. When we come across the top left (bottom right) corner, the corresponding boundary is removed from A_{left} (A_{bottom}) implying that the left (bottom) boundary for that obstacle becomes inactive at that point (lines 12 and 15).

To explain lines 13 to 17, let us refer to Fig. 4 where vertex b is the bottom right corner of an obstacle. It is easy to see that if any vertex u lying within the 45–45–90 triangle shown is still in A_{active} after scanning b, it can be removed from A_{active}. Since in this case all vertices in R_1 of u are completely blocked from u by the obstacle.

We have the following lemma that relates our obstacle-avoiding spanning graph generation algorithm to the obstacle-free spanning graph generation algorithm in [18].

Lemma 1: *The algorithm of Zhou et al. [18] is a special case of our OASG generation algorithm.*

Proof: If we consider an instance which has no obstacle, then we can simply ignore the blockage check in line 7 and lines 9–17 from the algorithm in Fig. 3. The resulting algorithm would be exactly the same as the spanning graph generation algorithm in [18]. □

Corollary 2: In the absence of obstacle, our OASG generation algorithm generates a spanning graph that contains at least one minimum rectilinear spanning tree of the given pins.

Proof: By Lemma 1, our OASG algorithm generates the same result as the obstacle-free spanning graph generation algorithm in [18] when there is no obstacle. Moreover, it has been proved in [18] that the obstacle-free spanning graph generated is guaranteed to contain at least one minimum spanning tree of the given pins. □

C. An Efficient Implementation

In this section, we show how to efficiently perform the following fundamental operations in the OASG generation algorithm: 1) Given a vertex v, find the subset of points in A_{active} which have v in their R_1; 2) given an edge, check if it is completely blocked by any obstacle boundary in A_{bottom} or A_{left}; and 3) given a bottom boundary of an obstacle, find all points in A_{active} which are completely blocked by the boundary. We address these issues one by one in the following paragraphs.

To find the subset of points in A_{active} that have a given point in their R_1, we first state and prove two lemmas and a corollary for our OASG generation algorithm. Similar ideas have been outlined in [18].

Lemma 2: *Point v is in the R_1 region of point p if and only if $x_p \leq x_v$ and $x_p - y_p \geq x_v - y_v$.*

Proof: By definition, the R_1 region of p is the region to the right of the vertical line passing through p and above the line with slope = 1 passing through p (see Fig. 2). In other words, point v is in the R_1 region of p if and only if $x_p \leq x_v$ and $\frac{y_v - y_p}{x_v - x_p} \geq 1$. Rearranging the terms, the necessary and sufficient condition is $x_p \leq x_v$ and $x_p - y_p \geq x_v - y_v$. □

Lemma 3: *At any time, no point in the active set can be in the R_1 region of another point in the set.*

Proof: Before we add a new point v to the active set (line 18), we would delete all points in the active set that have v in their R_1 regions (line 8). In addition, any point already in the active set cannot be in the R_1 region of point v as we are processing in increasing $(x + y)$ order. Hence, no point in the active set can be in the R_1 region of another point in the set at any time. □

Corollary 2: *For any two points p and q in the active set, we have $x_p \neq x_q$, and if $x_p < x_q$ then $x_p - y_p \leq x_q - y_q$.*

Proof: Assume points p and q are in the active set. Then we cannot have $x_p = x_q$, otherwise p would be in the R_1 region of q or vice versa by Lemma 2 which contradicts Lemma 3. And we cannot have $x_p < x_q$ and $x_p - y_p \geq x_q - y_q$, otherwise q would be in the R_1 region of p by Lemma 2 which again contradicts Lemma 3. Hence, the corollary is proved. □

To facilitate finding the points in A_{active} that have a given point in their R_1 regions, we keep the points in A_{active} in increasing order of their x-coordinate. To find the subset of points which have v in their R_1, we first find the largest x in A_{active} such that $x \leq x_v$. We then scan A_{active} in decreasing order of x until $x - y < x_v - y_v$. Note that by Corollary 2, decreasing order of x automatically implies non-increasing order of $x - y$. Any point in between has $x \leq x_v$ and $x - y \geq x_v - y_v$, and hence has v in its R_1 by Lemma 2. We use a balanced binary search tree to implement A_{active} in order to have $O(\log n)$ query operation to find the largest x in A_{active} such that $x \leq x_v$.

An edge $e(u, v)$ formed by points (x_u, y_u) and (x_v, y_v) is completely blocked by a bottom obstacle boundary (a, b) formed by the points (x_a, y_h) and (x_b, y_h), if and only if, $y_u < y_h < y_v$, $x_a < x_u$, and $x_b > x_v$. Note that at line 7, all bottom boundaries satisfying the condition must be present in the list A_{bottom}. We use a balanced binary search tree data structure for A_{bottom} with the y-coordinate of a boundary as a key value. If there are k bottom boundaries between y_u and y_v, it takes $O(\log n + k)$ time to check if any of them blocks edge e. Checking if an edge is completely blocked by a left boundary can be done similarly.

To determine all the completely blocked vertices u in A_{active} by a horizontal boundary (a, b) in line 16, we need to check if $y_u < y_h$, $x_a < x_u$ and $x_u - y_u + y_h \leq x_b$ (the lightly shaded region in Fig. 4). Since we already have A_{active} as a sorted list in increasing x we can check all points which lie between x_a and x_b and test for the above conditions to see if they are completely blocked.

In [16], we made a claim about runtime complexity of the overall algorithm being $O(n \log n)$. It has been recently brought to our attention that there may exist extreme cases for which the total time spent by line 7 over all iterations is $O(n^2)$. We concede with the argument but our algorithm is extremely fast for all practical purposes as indicated by our experimental results and it results in an OASG with a linear number of edges which limits the solution space resulting in better runtime complexity for subsequent stages.

IV. OPMST Generation

A. MTST Generation

After capturing the initial connectivity among pin vertices, the next logical step is to extract a MTST from the OASG that connects all pin vertices and avoid obstacles. Shen *et al.* [2] and Lin *et al.* [6] both used an indirect approach for this step. They first constructed a complete graph over all pin vertices where the edge weight is the shortest path length between the two pin vertices. On this complete graph they used either Prim's or Kruskal's algorithm to obtain a MST. Although it is effective, the approach described above seems to be an overkill as it is unnecessary to construct a complete graph when we already have OASG. Back in the 1980s, Wu *et al.* [17] suggested a method using Dijkstra's and Kruskal's algorithms on a graph similar to an OASG to obtain a MTST. Recently, Long *et al.* [8] adopted their approach to solve the problem on the OASG.

In this paper, we adopt the approach based on the extended Dijkstra's algorithm and the extended Kruskal's algorithm as defined in [8]. For every corner vertex in the OASG, we want to connect it with the nearest pin vertex. This can be easily

done using Dijkstra's shortest path algorithm considering every pin vertex as a source. After running the extended Dijkstra's algorithm we are left with a forest of m terminal trees, m being the number of pin vertices. The root of every terminal tree in the forest obtained above is a pin vertex. In order to connect all disjoint trees we use the extended Kruskal's algorithm on the forest. A priority queue Q is used to store the weights of all possible edges termed as *bridge edges* in [8] which can be used for linking the trees.

Definition 4 [8]: *An edge $e(u, v)$ is called a bridge edge if its two end vertices belong to different terminal trees.*

From Definition 4, it can be deduced that if each tree was a single vertex in the graph then bridge edges will be the edges connecting these vertices and we can use Kruskal's algorithm to obtain a MST in such a graph. The extended Kruskal's algorithm is simply an extended version of the original Kruskal's algorithm tailored to obtain a MST in a forest. It is important to note that in case we do not have any obstacle, the extended Dijkstra's algorithm will not make any change in the graph and the extended Kruskal will simply work on a spanning graph.

B. OPMST Construction

We note that a sparse OASG does not always have direct connections between the pin vertices even if one is allowed. This is due to a neighboring corner vertex being nearer than the other pin vertex in the same region. These indirect detour paths are unnecessary and if not taken care of can lead to a significant loss of quality. We note that the algorithm proposed by [8] failed to address this issue. On the other hand, we address this problem by constructing an OPMST from the MTST by removing all the corner vertices and storing detour information as the weight of an edge.

To construct an OPMST, we follow a simple strategy. For any corner vertex v, we find the nearest neighboring pin vertex u. We connect all the pin vertices originally connected with v to u and delete v. We update their weights as their original weight *plus* the weight of $e(u, v)$. This method guarantees that in case we have a major detour between two pin vertices due to an obstacle, the weight of that edge will corroborate this fact. In other words we can say that the edge would be *penalized* for the obstacles in its path.

V. OAST Generation

This step differentiates our algorithm from [2] and [6]–[8]. We exploit the extremely fast and efficient Steiner tree generation capability of FLUTE [15] for low degree nets. In order to embed FLUTE in our problem, we designed an obstacle aware version of FLUTE, OA-FLUTE. As OA-FLUTE is less effective for high degree nets and dense obstacle region, we partition a high degree net into subnets guided by the OPMST obtained from the previous step. The subproblems obtained after partitioning are passed on to OA-FLUTE for obstacle aware topology generation. It is termed as *obstacle-aware* because the nodes of the tree are placed in their appropriate locations considering obstacles around them.

```
Function: Partition(T)
Input: An OPMST T
Output: An OAST

1  if (∃ a completely blocked edge e) then
2      /* Refer to Fig. 6 */
3      e(u, v) is to be routed around obstacle edge e(a, b)
4      Let T = T1 + e(u, v) + T2
5      T1 = T1 + e(u, a)
6      T2 = T2 + e(u, b)
7      T' = Partition(T1) ∪ Partition(T2)
8  else if (|T| > HIGH THRESHOLD) then
10     /* Refer to Fig. 7(a) */
11     Let e(u, v) be the longest edge s.t.
           T = T1 + e(u, v) + T2 with |T1| ≥ 2 and |T2| ≥ 2
12     T' = Partition(T1) ∪ Partition(T2)
13     /* Refer to Fig. 7(b) */
14     Refine T' using OA-FLUTE(N")
15        where N" is a set of pin vertices around e(u, v) in T'
16 else
17     T' = OA-FLUTE(N)
18        where N is set of all vertices in T
19 end if
20 return T'
```

Fig. 5. Pseudocode for the partition function.

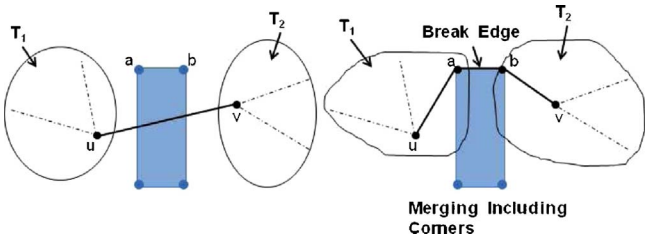


Fig. 6. Example illustrating first criterion for partitioning.

Figs. 5 and 8 describe the pseudocodes for the Partition and OA-FLUTE functions. It is evident that both functions are recursive functions. Let us first explain the Partition function.

A. Partition

The input to the Partition function is an OPMST obtained from the last step and the output is an OAST. An OAST is a Steiner tree in which the Steiner nodes have been placed considering the obstacles present in the routing region to minimize the overall wirelength. The following two criteria are set for partitioning pin vertices. The first criterion is to determine if any edge is completely blocked by an obstacle. The second criterion is to check if the size of OPMST is more than the HIGH THRESHOLD defined.

As can be clearly seen in Fig. 6 that for an overlap free solution, we have to route around the obstacle. Therefore, it seems logical to break the tree at edge (u, v). We know that OA-FLUTE can efficiently construct a tree when the number of nodes is less than the HIGH THRESHOLD value. If the size of the tree is still more than the HIGH THRESHOLD after breaking at the blocking obstacles, we need to break the tree further. In this case, we look for the edge with the largest weight on the tree and delete that edge, refer to Fig. 7(a).

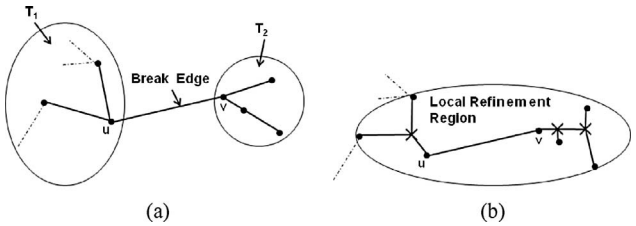


Fig. 7. Example illustrating second criterion for partitioning. (a) Partitioning. (b) Local refinement.

Function: OA-FLUTE(N)
Input: A set of nodes N
Output: An OAST

1 T' = FLUTE(N)
2 **if** ($\exists$ a completely blocked edge e) **then**
3 $e(u, v)$ is to be routed around obstacle boundary $e(a, b)$
4 /* Refer to Fig. 9 */
5 Let $N = N_1 \cup N_2$
6 $N_1 = N_1 \cup \{a\}$
7 $N_2 = N_2 \cup \{b\}$
8 T' = OA-FLUTE(N_1) $\cup$ OA-FLUTE(N_2)
9 **else if** ($\exists$ Steiner Node S that falls on any obstacle) **then**
10 /* Refer to Fig. 10 */
11 Let $a_1, a_2, \ldots, a_D$ be the intersection points with the obstacle ordered in anti-clockwise direction
12 Let $N = N_1 \cup N_2 \cup \ldots \cup N_D \cup \{S\}$
13 Let (a_u, a_v) be the segment with largest weight
14 **for** ($i = a_v$ to a_u in anti-clockwise order) **do**
15 $N_i = N_i \cup$ corner vertices along the path next to N_i
16 **end for**
17 T' = OA-FLUTE(N_1) $\cup$ ..... $\cup$ OA-FLUTE(N_D)
18 **end if**
19 **return** T'

Fig. 8. Pseudocode for the OA-FLUTE function.

Based on the above-mentioned criteria, if we break an obstacle edge, we simply include corner vertices in the tree and divide the two trees as shown in Fig. 7. Else, if we break at the edge with largest weight, we delete that edge and make sure that it does not contain any leaf of the tree as shown in Fig. 7(a).

After breaking an edge, we make recursive calls to the Partition function using two subtrees. When the size of the tree becomes less than the HIGH THRESHOLD, we pass the nodes of the tree to OA-FLUTE function. The OA-FLUTE function returns an OAST. After returning from OA-FLUTE in Partition, if the partition was performed on an obstacle edge, we simply merge two Steiner trees using the same obstacle edge. In case the partition was performed on the longest edge, we explore an opportunity to further optimize wirelength. We merge the two trees on the longest edge and then search the region around the longest edge to extract neighboring pin vertices, refer to lines 12–15 in Figs. 5 and 7(b). This refinement is the same as the local refinement proposed in [15]. We pass this set of nodes to OA-FLUTE for further optimization.

B. OA-FLUTE

The purpose of OA-FLUTE function is to form an OAST. It begins by calling FLUTE on the set of input nodes. FLUTE constructs a Steiner tree without considering obstacles. This tree can have two kinds of overlap: 1) an edge completely blocked by an obstacle, and 2) a Steiner node falling on any obstacle. We handle both of these cases differently.

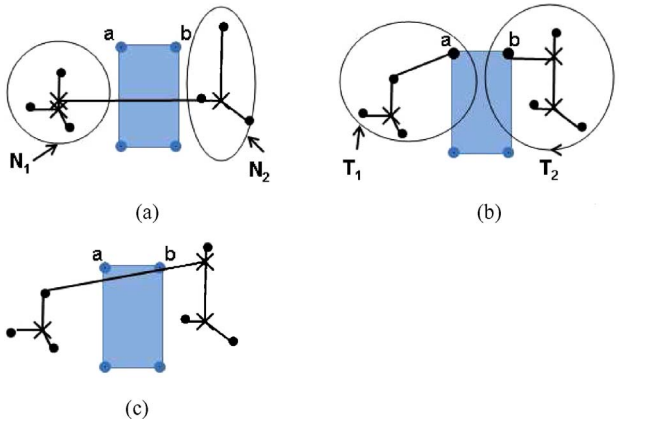


Fig. 9. OA-FLUTE: handling an edge completely blocked by an obstacle. (a) Completely blocked edge. (b) Subtrees before merging. (c) Merging excluding corners.

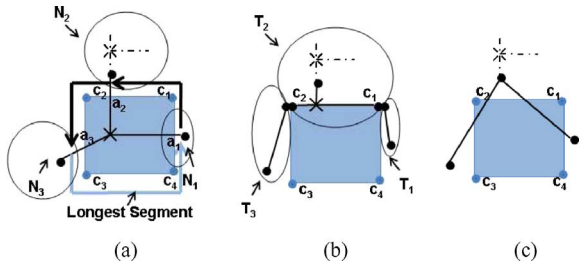


Fig. 10. OA-FLUTE: handling Steiner node falling on an obstacle. (a) Removing longest segment. (b) Subtrees before merging. (c) Merging excluding corners.

To handle the first case, refer to Fig. 9, we break the Steiner tree into two subtrees including corner points of the obstacle as in Fig. 9(b) and make recursive calls to OA-FLUTE. We selectively prune the number of recursive calls based on the size of the tree in order to strike a balance between run-time and quality.

To handle the second case, we devised a special technique. We pick an obstacle which has a Steiner node on top of it. For every boundary of this obstacle intersecting with the Steiner tree, we extract a set of nodes N_i which includes the pin vertices in the tree near to that boundary. In Fig. 10(a), we have a single Steiner node inside the obstacle intersecting at a_1, a_2, and a_3, with the right, top, and left boundary of the obstacle, respectively. We extract three set of pin vertices N_1, N_2, and N_3 from the original Steiner tree for the right, top, and left boundary, respectively. The points a_1, a_2, and a_3 divide the obstacle outline into three segments as shown in Fig. 10(a). We then find the longest segment [the light shaded segment (a_3, a_1) in Fig. 10(a)]. We then traverse from one endpoint of the longest segment to the other endpoint via other segments in an anti-clockwise direction, for example, from a_1 to c_1 to a_2 to c_2 to a_3 in Fig. 10(a). While moving along the other segments, we keep adding corner vertices to the corresponding N_i's, e.g.,

c_1 gets added to both N_1 and N_2 and c_2 gets added to both N_2 and N_3 in Fig. 10(b). We then recursively call OA-FLUTE for all N_i's thus formed.

As our goal with OA-FLUTE is to determine befitting locations for Steiner nodes we exclude all corner vertices when we finally merge the subtrees. Figs. 9(c) and 10(c) show the final Steiner trees after excluding corners while merging. The reason for not adding corner vertices in this step is twofold. First, it is not desirable to further restrict the solution when we already did once in Partition function. Second, we want our OA-FLUTE to be a generic function which can preserve the number of pin-vertices provided to it, adding corner vertices would increase them.

C. Fast Implementation with Obstacle Tree Data Structure

To create an OAST, OA-FLUTE performs two major tasks. It tries to remove all completely blocked edges and removes all Steiner nodes falling on any obstacle. In order to perform these tasks, it requires to check each edge and Steiner node of the tree against all obstacles if we use a naive list data structure to represent the obstacles. Our experiments in [16] were performed based upon this simple approach.

In this paper, we propose an efficient obstacle tree (OBTree) data structure. An OBTree is a balanced binary tree in which we bin obstacles in their enclosing regions. We start with a region which encloses all obstacles. After that, depending upon the dimensions of the region, we slice the region either vertically or horizontally into two parts. After dividing the region into two halves we create two similar problems as before. We use recursion to further split these regions until there is only one obstacle inside each bin.

Fig. 11 describes the pseudocode used by FOARS to create an obstacle tree. The procedure CreateOBTree is provided with a list of obstacles. In case the region enclosing the obstacles has larger width than height, we divide the obstacles into two parts based upon the x-coordinates of their bottom left corners. Otherwise, the y-coordinates of bottom left corners of obstacles are used.

OBTree is extremely efficient for searching if an edge is completely blocked by any obstacle or if a Steiner node falls on any obstacle. The pseudocodes in Figs. 12 and 13 are recursive procedures to perform above-mentioned tasks.

```
Function: CreateOBTree(OB, K)
Input: A set of obstacles OB = {O_1, O_2, ..., O_K}
       Number of obstacles K
Output: An obstacle tree OBTree

1  OBTree → OB = OB
2  OBTree → x_MIN = Leftmost boundary of OB
3  OBTree → x_MAX = Rightmost boundary of OB
4  OBTree → y_MIN = Bottommost boundary of OB
5  OBTree → y_MAX = Topmost boundary of OB
6  OBTree → C_1 = NULL
7  OBTree → C_2 = NULL
8  if (K > 1) then
9      H = ⌈K/2⌉
10     if (OBTree → x_MAX − OBTree → x_MIN
           > OBTree → y_MAX − OBTree → y_MIN) then
11         OB_L = {first H obstacles in increasing order of x}
12         OB_R = OB − OB_L
13         OBTree → C_1 = CreateOBTree(OB_L, H)
14         OBTree → C_2 = CreateOBTree(OB_R, K − H)
15     else
16         OB_B = {first H obstacles in increasing order of y}
17         OB_T = OB − OB_B
18         OBTree → C_1 = CreateOBTree(OB_B, H)
19         OBTree → C_2 = CreateOBTree(OB_T, K − H)
20     end if
21 end if
22 return OBTree
```

Fig. 11. Pseudocode for creating an OBTree.

```
Function: CheckSteinerNode(α, π)
Input: Steiner node α; obstacle tree node π
Output: Set of obstacles under π on which α falls

1  Let R =region enclosed by π
2  if (α does not lie inside R) then
3      return ∅
4  else if (π contains a single obstacle O) then
5      return {O}
6  else
7      Let π_1 and π_2 be children of π
8      OB_1 = CheckSteinerNode(α, π_1)
9      OB_2 = CheckSteinerNode(α, π_2)
10     OB = OB_1 ∪ OB_2
11     return OB
12 end if
```

Fig. 12. Pseudocode for checking if a Steiner node falls on any obstacle.

VI. OARSMT Generation

The OAST obtained from last step does not guarantee that rectilinear path for a pin-to-pin connection is obstacle free. In this step, we rectilinearize every pin-to-pin connection avoiding obstacles to generate an OARSMT. For every Manhattan connection between two pins we can have two L-shape paths. On the basis of the obstacles inside the bounding box formed by an edge, we can divide all the possible scenarios into four categories: 1) both L-paths are clean; 2) both L-paths are blocked by the same obstacle; 3) only one L-path is blocked; and 4) both L-paths are blocked but not by the same obstacle. We discuss these scenarios one by one in the following paragraphs.

For the first case, even though we can rectilinearize using any L-path, we instead create a slant edge at this stage to leave the scope for improvement in V-shape refinement. For the second case, we have no option but to go outside the bounding box and pick the least possible detour.

For the third case, we route inside the bounding box, since there exists a path. We break the edge into two sub problems on the corner of an obstacle along the blocked L-path. We recursively solve these sub problems to determine an obstacle-avoiding path. If the wirelength of this path is the same as the Manhattan distance between the pins, we accept the solution, else we route along the unblocked L-path. It is noteworthy that for this case we could have directly accepted

Function: CheckEdge(e, π)
Input: Edge e; obstacle tree node π
Output: Set of obstacles under π completely blocking e

```
1  Let R =region enclosed by π
2  if (e cannot completely blocked by any rectangle in R) then
       /* i.e., e can completely avoid π without detour */
3      return ∅
4  else if (π contains a single obstacle O) then
5      return {O}
6  else
7      Let π1 and π2 be children of π
8      OB1 = CheckEdge(e, π1)
9      OB2 = CheckEdge(e, π2)
10     OB = OB1 ∪ OB2
11     return OB
12 end if
```

Fig. 13. Pseudocode for checking if an edge is completely blocked.

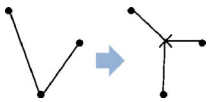


Fig. 14. V-shape refinement case and refined output.

the unblocked L-path. In order to create more slant edges, and hence, further scope for V-shape refinement, we searched for a route along the blocked L-path avoiding obstacles. For the last case where both L-paths are blocked but not by the same obstacle, we determine obstacle-avoiding routes using the same recursive approach as mentioned above for both L-paths and pick the shortest one.

VII. Refinement

We perform a final V-shape refinement to improve total wirelength. This refinement includes movement of Steiner node in order to discard extra segments produced due to previous steps. The concept of refinement is similar to the one that determines a Steiner node for any three terminals. The coordinates of the Steiner node are the median value of the x-coordinates and median value of the y-coordinates. Fig. 14 illustrates a potential case for V-shape refinement and output after refinement. This refinement comes handy in improving the overall wirelength by 1% to 2%.

VIII. Experimental Results

We implemented our algorithm in C. The experiments were performed on a 3 GHz AMD Athlon 64 X2 Dual Core machine. We requested binaries from Long *et al.* [8], Lin *et al.* [6], Liang *et al.* [10], Liu *et al.* [11], [12], and ran them on our platform. Five industrial testcases (IND1–IND05), 12 circuits from [6] (RC01–RC12), five randomly generated benchmark circuits (RT01–RT05) [6], and five large benchmark circuits (RL01–RL05) generated by [8].

A. OARSMT Experimental Results

Tables I and II show wirelength and runtime comparison on benchmarks containing obstacles. We determined experimentally that HIGH THRESHOLD value of 20 works the best.

TABLE I
Wirelength and Runtime Comparison Between FOARS [16] and Current Results

			Wirelength		Runtime (s)	
Benchmark	m	k	FOARS [16]	FOARS	FOARS [16]	FOARS
RC01	10	10	25 980	25 980	0.00	0.00
RC02	30	10	42 110	42 110	0.00	0.00
RC03	50	10	56 030	56 030	0.00	0.00
RC04	70	10	59 720	59 720	0.00	0.00
RC05	100	10	75 000	75 000	0.00	0.00
RC06	100	500	81 229	81 229	0.03	0.03
RC07	200	500	110 764	110 764	0.03	0.03
RC08	200	800	116 047	116 047	0.06	0.05
RC09	200	1000	115 593	115 593	0.08	0.06
RC10	500	100	168 280	168 280	0.02	0.02
RC11	1000	100	234 416	234 416	0.04	0.03
RC12	1000	10 000	756 998	756 998	2.04	1.19
RT01	10	500	2191	2191	0.01	0.00
RT02	50	500	48 156	48 156	0.02	0.02
RT03	100	500	8282	8282	0.03	0.03
RT04	100	1000	10 330	10 330	0.07	0.06
RT05	200	2000	54 598	54 634	0.21	0.15
IND1	10	32	604	604	0.00	0.00
IND2	10	43	9500	9500	0.00	0.00
IND3	10	59	600	600	0.00	0.00
IND4	25	79	1129	1129	0.00	0.00
IND5	33	71	1364	1364	0.00	0.00
RL01	5000	5000	483 027	483 027	3.13	1.15
RL02	10 000	500	637 753	637 753	1.36	1.18
RL03	10 000	100	640 902	640 902	1.15	1.13
RL04	10 000	10	697 125	697 125	1.55	1.57
RL05	10 000	0	728 438	728 670	1.66	0.12
			(0.999)	(1)	11.54(1.59)	7.23(1)

Table I shows our newest results as compared with results published in [16]. Using OBTree in FOARS we were able to cut down the runtime by 59% with negligible increase in wirelength. The reason for the slight increase in wirelength is that we decided to disable the refinement step for instances with no obstacle (RL05 in Table I). Based on experiments, we determine that if we remove the refinement step for instances with no obstacles, we gain significantly in runtime with negligible loss in quality (see Table III).

Columns 4 and 5 of Table II show that FOARS outperforms Lin *et al.* [6] by 2.3% and Long *et al.* [8] by 2.7%. Columns 6–8 indicate that FOARS has similar wirelength results as compared with Liang *et al.* [10] and Liu *et al.* [11], [12]. For the runtime in Table II, we are now 84% faster than Long *et al.* [8] on average. We are 46 times faster than [10] and 123 times faster than [6]. Our runtime is slower as compared with Liu *et al.* [11], [12].

B. RSMT Experimental Results

As mentioned before, RSMT can be seen as a special case for OARSMT. In an effort to construct a single solution for OARSMT and RSMT generation, we performed experiments on our existing benchmarks after deleting obstacles. As shown in Table III, we compare our results with Long *et al.* [8] and Liang *et al.* [10]. We could not compare our results with [11] and [12] as their binaries could not run on cases with no obstacles. We also compared our results with FLUTE-2.5

TABLE II

WIRELENGTH AND RUNTIME COMPARISON

			Wirelength						Runtime (s)					
Benchmark	m	k	Lin [6]	Long [8]	Liang [10]	Liu [11]	Liu [12]	FOARS	Lin [6]	Long [8]	Liang [10]	Liu [11]	Liu [12]	FOARS
RC01	10	10	27 790	26 120	25 980	26 740	26 040	25 980	0.00	0.00	0.01	0.00	0.00	0.00
RC02	30	10	42 240	41 630	42 010	42 070	41 570	42 110	0.00	0.00	0.02	0.00	0.00	0.00
RC03	50	10	56 140	55 010	54 390	54 550	54 620	56 030	0.00	0.00	0.00	0.00	0.00	0.00
RC04	70	10	60 800	59 250	59 740	59 390	59 860	59 720	0.00	0.00	0.01	0.00	0.00	0.00
RC05	100	10	76 760	76 240	74 650	75 440	74 770	75 000	0.00	0.00	0.01	0.00	0.00	0.00
RC06	100	500	84 193	85 976	81 607	81 903	81 854	81 229	0.10	0.08	0.50	0.01	0.02	0.03
RC07	200	500	114 173	116 450	111 542	111 752	110 851	110 764	0.18	0.09	0.60	0.01	0.03	0.03
RC08	200	800	120 492	122 390	115 931	118 349	116 132	116 047	0.31	0.15	1.16	0.02	0.04	0.05
RC09	200	1000	117 647	118 700	113 460	114 928	113 559	115 593	0.40	0.22	1.53	0.02	0.05	0.06
RC10	500	100	171 519	168 500	167 620	167 540	167 460	168 280	0.20	0.03	0.18	0.00	0.01	0.02
RC11	1000	100	237 794	234 650	235 283	234 097	236 018	234 416	0.74	0.06	0.83	0.01	0.02	0.03
RC12	1000	10 000	803 483	832 780	761 606	780 528	762 435	756 998	55.09	3.80	186.3	0.36	1.20	1.19
RT01	10	500	2289	2379	2231	2259	2193	2191	0.03	0.06	0.19	0.01	0.01	0.00
RT02	50	500	48 858	51 274	47 297	48 684	47 488	48 156	0.05	0.06	0.55	0.01	0.02	0.02
RT03	100	500	8508	8554	8187	8347	8231	8282	0.10	0.06	0.21	0.01	0.02	0.03
RT04	100	1000	10 459	10 534	9914	10 221	9893	10 330	0.22	0.23	0.37	0.02	0.04	0.06
RT05	200	2000	54 683	55 387	52 473	53 745	52 509	54 634	0.96	0.66	3.18	0.04	0.12	0.15
IND1	10	32	632	639	619	626	604	604	0.00	0.00	0.00	0.00	0.00	0.00
IND2	10	43	9700	10 000	9500	9700	9600	9500	0.00	0.00	0.00	0.00	0.00	0.00
IND3	10	59	632	623	600	600	600	600	0.00	0.00	0.00	0.00	0.00	0.00
IND4	25	79	1121	1130	1096	1095	1092	1129	0.00	0.00	0.00	0.00	0.00	0.00
IND5	33	71	1392	1379	1360	1364	1374	1364	0.00	0.00	0.00	0.00	0.00	0.00
RL01	5000	5000	492 865	491 855	481 813	483 134	483 199	483 027	106.66	3.58	27.14	0.27	0.63	1.15
RL02	10 000	500	648 508	638 487	638 439	636 097	640 435	637 753	159.09	1.27	29.45	0.23	0.37	1.18
RL03	10 000	100	652 241	641 769	642 380	640 266	644 276	640 902	153.95	1.08	23.35	0.22	0.32	1.13
RL04	10 000	10	709 904	697 595	699 502	696 111	700 937	697 125	195.25	0.97	22.00	0.24	0.29	1.57
RL05	10 000	0	741 697	728 585	730 857	–	–	728 670	217.88	0.96	33.64	–	–	0.12
									891.25	13.36	331.235	1.48	3.19	7.23
Norm			1.024	1.027	0.995	1.004	0.994	1	123.26	1.84	45.81	0.20	0.44	1

m is the number of pin vertices and k is the number of obstacles. The values in the last row are normalized over our results for both wirelength as well as runtime.

TABLE III

WIRELENGTH AND RUNTIME COMPARISON FOR BENCHMARK WITH NO OBSTACLES, I.E. $k = 0$ FOR ALL CASES

			Wirelength				Runtime (s)			
Benchmark	m	k	Long [8]	Liang [10]	FLUTE-2.5 [15]	Ours	Long [8]	Liang [10]	FLUTE-2.5 [15]	FOARS
RC01	10	0	25 290	25 290	25 290	25 290	0.00	0.00	0.00	0.00
RC02	30	0	40 100	40 630	39 920	39 920	0.00	0.00	0.00	0.00
RC03	50	0	52 560	52 440	53 400	53 050	0.00	0.00	0.00	0.00
RC04	70	0	55 850	55 720	57 020	55 380	0.00	0.00	0.00	0.00
RC05	100	0	72 820	71 820	73 370	72 170	0.00	0.00	0.00	0.00
RC06	100	0	77 886	78 068	80 057	77 633	0.00	0.00	0.00	0.00
RC07	200	0	106 591	107 236	109 232	106 581	0.01	0.07	0.00	0.00
RC08	200	0	109 625	109 059	112 787	108 928	0.00	0.03	0.00	0.00
RC09	200	0	109 105	108 101	112 460	108 106	0.01	0.02	0.00	0.00
RC10	500	0	164 940	164 450	170 270	164 130	0.02	0.17	0.00	0.00
RC11	1000	0	233 743	235 284	245 325	233 647	0.06	0.70	0.00	0.00
RC12	1000	0	755 332	764 956	798 742	755 354	0.04	0.75	0.00	0.00
RT01	10	0	1817	1817	1817	1817	0.01	0.00	0.00	0.00
RT02	50	0	44 930	46 109	45 291	44 416	0.00	0.00	0.00	0.00
RT03	100	0	7677	7777	7811	7749	0.00	0.00	0.00	0.00
RT04	100	0	7792	7826	7826	7792	0.00	0.00	0.00	0.00
RT05	200	0	43 335	43 586	44 809	43 026	0.00	0.00	0.00	0.00
IND1	10	0	614	619	604	604	0.00	0.00	0.00	0.00
IND2	10	0	9100	9100	9100	9100	0.00	0.00	0.00	0.00
IND3	10	0	590	590	587	587	0.00	0.00	0.00	0.00
IND4	25	0	1092	1092	1102	1102	0.00	0.00	0.00	0.00
IND5	33	0	1314	1304	1307	1307	0.00	0.00	0.00	0.00
RL01	5000	0	472 392	473 905	501 480	472 818	0.30	11.39	0.05	0.05
RL02	10 000	0	637 131	641 722	674 042	636 895	0.95	32.45	0.25	0.12
RL03	10 000	0	641 289	650 343	674 950	640 580	0.95	33.04	0.26	0.12
RL04	10 000	0	697 712	699 617	740 270	697 239	0.99	32.26	0.25	0.13
RL05	10 000	0	728 595	730 857	778 313	728 670	1.05	34.52	0.26	0.12
			(1.002)	(1.005)	(1.026)	(1)	4.52(7.75)	145.37(249)	1.104(1.89)	0.58(1)

[15] which is the same version of FLUTE as used inside FOARS.

Our result for wirelength is the best among all the algorithms and is 2.6% better as compared with FLUTE-2.5. Compared with FLUTE-2.5, which has been demonstrated to be significantly faster than other RSMT heuristics, FOARS are 89% faster. FOARS are 7.75 and 249 times faster than [8] and [10], respectively. Again FOARS performs much better when we have large number of pin vertices in the benchmark (RC12, RL01–RL05). The improvement in wirelength over FLUTE-2.5 is due to the effective partitioning algorithm for high-degree nets and the application of the local refinement technique as shown in Fig. 7(b).

IX. Conclusion

In this paper, we presented FOARS, an efficient algorithm to construct OARSMT and RSMT based on extremely fast and high-quality Steiner tree generation tool called FLUTE. We proposed a novel OASG algorithm with a linear number of edges. We also proposed an obstacle aware version of FLUTE, which generates OAST. Our top-down partition approach empowers OA-FLUTE to handle high-degree nets and dense obstacle region. Our implementation of OBTree is simple and extremely efficient for checking blockage with obstacles. Our results indicate that our approach is the best tradeoff for quality and runtime for both OARSMT and RSMT construction. Our experiments prove that FOARS obtains good quality solution with excellent runtime as compared with its peers.

Acknowledgment

The authors acknowledge Lin *et al.* [6], Liang *et al.* [10], Long *et al.* [8], and Liu *et al.* [11], [12] for sending us their binaries for comparison and clearing our doubts, if any, with respect to the results.

References

[1] Y. Hu, Z. Feng, T. Jing, X. Hong, Y. Y. Ge, X. Hu, and G. Yan, "FORst: A 3-step heuristic for obstacle-avoiding rectilinear Steiner minimal tree construction," in *Proc. JICS*, 2004, pp. 107–116.

[2] Z. Shen, C. C. N. Chu, and Y.-M. Li, "Efficient rectilinear Steiner tree construction with rectilinear blockages," in *Proc. ICCD*, 2005, pp. 38–44.

[3] Y. Hu, T. Jing, X. Hong, Z. Feng, X. Hu, and G. Yan, "An-OARSMan: Obstacle-avoiding routing tree construction with good length performance," in *Proc. ASP-DAC*, 2006, pp. 630–635.

[4] Y. Shi, P. Mesa, H. Yao, and L. He, "Circuit simulation based obstacle-aware Steiner routing," in *Proc. DAC*, 2006, pp. 385–388.

[5] P.-C. Wu, J.-R. Gao, and T.-C. Wang, "A fast and stable algorithm for obstacle-avoiding rectilinear Steiner minimal tree construction," in *Proc. ASP-DAC*, 2007, pp. 262–267.

[6] C.-W. Lin, S.-Y. Chen, C.-F. Li, Y.-W. Chang, and C.-L. Yang, "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.

[7] 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. ISPD*, 2008, pp. 126–133.

[8] J. Long, H. Zhou, and S. O. Memik, "EBOARST: An efficient edge-based obstacle avoiding-rectilinear Steiner tree construction algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 12, pp. 2169–2182, Dec. 2008.

[9] I. H.-R. Jiang, S.-W. Lin, and Y.-T. Yu, "Unification of obstacle-avoiding rectilinear Steiner tree construction," in *Proc. SoCC*, 2008, pp. 127–130.

[10] L. Li and E. F. Y. Young, "Obstacle-avoiding rectilinear Steiner tree construction," in *Proc. ICCAD*, 2008, pp. 523–528.

[11] C.-H. Liu, S.-Y. Yuan, S.-Y. Kuo, and Y.-H. Chou, "An O(n log n) path-based obstacle-avoiding algorithm for rectilinear Steiner tree construction," in *Proc. DAC*, 2009, pp. 314–319.

[12] C.-H. Liu, S.-Y. Yuan, S.-Y. Kuo, and J.-H. Weng, "Obstacle-avoiding rectilinear Steiner tree construction based on Steiner point selection," in *Proc. ICCAD*, 2009, pp. 26–32.

[13] L. Li, Z. Qian, and E. F. Y. Young, "Generation of optimal obstacle-avoiding rectilinear Steiner minimum tree," in *Proc. ICCAD*, 2009, pp. 21–25.

[14] F. K. Hwang, "On Steiner minimal trees with rectilinear distance," *SIAM J. Appl. Math.*, vol. 30, no. 1, pp. 104–114, Jan. 1976.

[15] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[16] G. Ajwani, C. Chu, and W.-K. Mak, "FOARS: FLUTE based obstacle-avoiding rectilinear Steiner tree construction," in *Proc. ISPD*, 2010, pp. 185–192.

[17] Y. F. Wu, P. Widmayer, and C. K. Wong, "A faster approximation algorithm for the Steiner problems in graphs," *Acta Informatica*, vol. 23, no. 2, pp. 223–229, 1986.

[18] H. Zhou, N. V. Shenoy, and W. Nicholls, "Efficient minimum spanning tree construction without Delaunay triangulation," in *Proc. ASP-DAC*, 2001, pp. 192–197.

Gaurav Ajwani (S'09) received the B.S. degree from the Netaji Subhas Institute of Technology, University of Delhi, New Delhi, India, in 2006, and the M.S. degree in computer engineering from Iowa State University, Ames, in 2010.

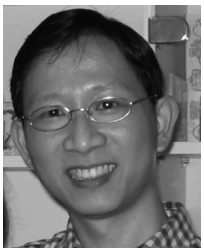
Since March 2010, he has been a Computer-Aided Design Engineer developing flows for Intel, Hillsboro, OR. Before joining Iowa State University, he worked briefly with Freescale, Austin, TX, as a Design Engineer. His current research interests include routing for multi-terminal net in the presence of obstacles.

Mr. Ajwani's work titled "FOARS: FLUTE based obstacle avoiding rectilinear Steiner tree construction" was nominated for the Best Paper Award at the International Symposium for Physical Design in 2010.

Chris Chu received the B.S. degree in computer science from the University of Hong Kong, Pokfulam, Hong Kong, in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Texas, Austin, in 1994 and 1999, respectively.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Iowa State University, Ames. His area of expertise includes computer-aided design of very large scale integration physical design, and design and analysis of algorithms. His current research interests include performance-driven interconnect optimization and fast circuit floorplanning, placement, and routing algorithms.

Dr. Chu received the IEEE Transactions on Computer-Aided Design Best Paper Award in 1999 for his work in performance-driven interconnect optimization, another IEEE Transactions on Computer-Aided Design Best Paper Award in 2010 for his work on routing tree construction, the ISPD Best Paper Award in 2004 for his work on efficient placement algorithms, and the Bert Kay Best Dissertation Award for 1998 to 1999 from the Department of Computer Sciences, University of Texas, Austin.



Wai-Kei Mak (M'03) received the B.S. degree in computer science from the University of Hong Kong, Pokfulam, Hong Kong, in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Texas, Austin, in 1995 and 1998, respectively.

From 1999 to 2003, he was with the Department of Computer Science and Engineering, University of South Florida, Tampa, as an Assistant Professor. Since 2003, he has been with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, where he is currently an Associate Professor. His current research interests include very large scale integration physical design automation, field-programmable gate array architecture, and computer-aided design.

Dr. Mak has served on the program and/or the organizing committees of the Asia South Pacific Design Automation Conference, the International Conference on Field Programmable Logic and Applications, and the International Conference on Field-Programmable Technology. He was the General Chair of the 2008 International Conference on Field-Programmable Technology and was the Technical Program Co-Chair of the same conference in 2006. He has been in the Steering Committee of the International Conference on Field-Programmable Technology since 2009.