FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction*

Gaurav Ajwani
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
gajwani@iastate.edu

Chris Chu
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
cnchu@iastate.edu

Wai-Kei Mak
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300 R.O.C.
wkmak@cs.nthu.edu.tw

ABSTRACT

Obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) construction is becoming one of the most sought after problems in modern design flow. In this paper we present FOARS, an algorithm to route a multi-terminal net in the presence of obstacles. FOARS is a top down approach which includes partitioning the initial solution into subproblems and using obstacle aware version of Fast Lookup Table based Wirelength Estimation (OA-FLUTE) at a lower level to generate an OAST followed by recombining them with some backend refinement. To construct an initial connectivity graph FOARS uses a novel obstacle-avoiding spanning graph (OASG) algorithm which is a generalization of Zhou's spanning graph algorithm without obstacle [1]. FOARS has a run time complexity of $O(n \log n)$. Our experimental results indicate that it outperforms Lin et al. [2] by 2.3% in wirelength. FOARS also has 20% faster run time as compared with Long et al. [3], which is the fastest solution till date.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance, Theory

Keywords

Physical Design, Routing, Spanning Graph, RSMT

1. 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, etc. 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., [2–12]). As pointed out by Hwang [13], in the absence of obstacles multi-terminal net routing corresponds to the rectilinear Steiner minimal tree problem which is NP-complete. The presence of obstacles in the region makes multi-terminal routing problem even harder.

In this work, we develop a new $O(n \log n)$ time algorithm called FOARS for OARSMT generation by leveraging FLUTE [14]. FLUTE is a very fast and robust tool for the rectilinear Steiner minimal tree problem without obstacle. It is widely used in many recent academic global routers. 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 by such a simple strategy can be far from optimal. A better strategy is to break the Steiner tree produced by FLUTE on overlapping obstacles, recursively call FLUTE for local optimization, and then combine all locally optimized subtrees at the end. However, as the number of pins increases or if the routing region is severely cluttered with obstacles, the quality of the solution produced will degrade because it lacks a global view of the problem. 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 instances that can be effectively handled.

To guide the partitioning algorithm, we propose to use a sparse obstacle-avoiding spanning graph (OASG) to capture the proximity information amongst the pins and corners of obstacles. Three categories of graph have been used to capture the proximity information during OARSMT construction in the past. [4, 6, 7, 11] all use the escape graph. [10] utilizes 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. [2, 3, 5, 8, 9] are based on various forms of obstacle-avoiding spanning graphs. Shen et al. [5] proposed a form of OASG that only contains a linear number of edges which is also adopted in [8]. Later Lin et al. [2] 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

*This work was partially supported by National Science of Council of Taiwan under grant NSC 98-2220-E-007-031, IBM Faculty Award, and NSF under grant CCF-0540998.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISPD'10, March 14–17, 2010, San Francisco, California, USA.
Copyright 2010 ACM 978-1-60558-920-6/10/03 ...$5.00.

large extent. In view of that, Long et al. [3, 9] 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, 3, 5, 9] 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 constructing an OARSMT from an OASG directly is that it tends to follow obstacle boundaries and make detours towards obstacle corners. This makes it easier to lead to congestion when routing many nets in a design. (Adding essential edges as in [2] 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 [3,9] and possesses other desirable properties not found in Long's OASG. For example, our OASG is guaranteed to contain at least one minimum spanning tree in the absence of obstacle while Long's OASG does not have such a guarantee.

The rest of the paper is organized as follows. We first provide an overview of the main steps of our OARSMT construction approach in Section 2. Each main step is described in details in Sections 3 to 7. The experimental results are reported in Section 8. Finally, we give our conclusion in Section 9.

2. OVERVIEW OF THE ALGORITHM

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

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 3 describes the OASG algorithm in detail.

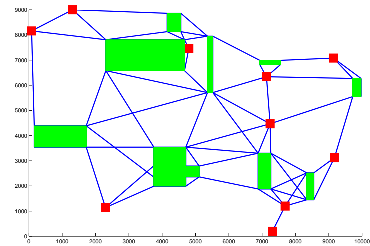
Stage 2: OPMST Generation. Based on the OASG, we construct a minimum terminal spanning tree (MTST) using the approach mentioned in [15] and then obtain an obstacle penalized minimal spanning tree (OPMST) from the MTST. Section 4 talks about OPMST construction in detail.

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 5 talks about the partitioning and OA-FLUTE in more detail.

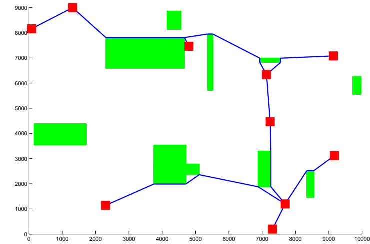
Stage 4: OARSMT Generation. In this step, we rectilinearize the pin-to-pin connections avoiding obstacles to construct an OARSMT. Section 6 discusses OARSMT construction.

Stage 5: Refinement. To further reduce the wirelength, we perform V-shape refinement on the OARSMT. Details for it can be found in Section 7.

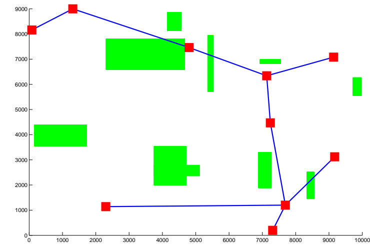
Fig. 1 depicts the outputs after various stages of the algorithm.



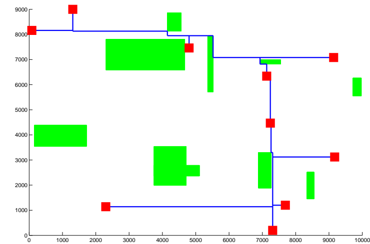
(a) OASG



(b) MTST



(c) OPMST



(d) OARSMT

Figure 1: Outputs at various stages for benchmark RC01

3. OASG GENERATION

3.1 Previous Approaches

Zhou et al. [1] described a spanning graph generation algorithm with $O(n)$ edges in the absence of obstacles. We prove that their approach can be seen as a special case of our obstacle-avoiding spanning graph generation algorithm. Here we start with a few definitions.

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 run time 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. [5] 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. [9] recently described a novel quadrant approach which is a modified version of [1] for OASG generation with a linear number of edges in $O(n \log n)$ time. 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, similar to [1]. After scanning any vertex v, they search 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.

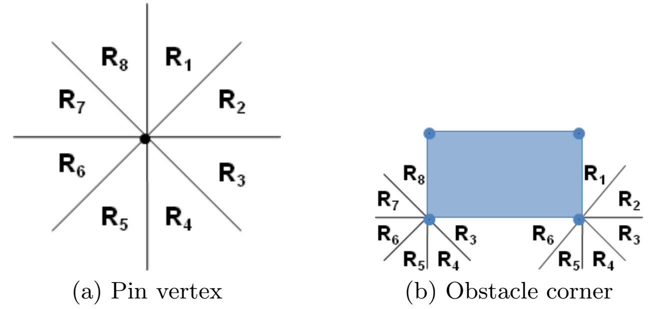
We found that their algorithm has certain shortcomings. First, their algorithm is not symmetric, i.e., the nearest neighbor for any vertex in a quadrant is contingent upon the direction of scanning which means they have to scan along all four quadrants of a vertex in order to capture its connectivity information. Second, unlike [1] in the absence of obstacles, their algorithm cannot guarantee the presence of at least one MST in the plane. Third, their algorithm cannot handle abutting obstacles due to minor mistakes in the inequality conditions.

3.2 Our Approach for OASG

Looking at the above mentioned issues we conceived that rather than modifying Zhou et al's [1] approach, it will be best to simply build on their idea. Therefore, we propose an algorithm based on *octant partition* exhibiting *uniqueness property* similar to their algorithm. We reiterate the definition given in their paper. The notation $||pq||$ represents rectilinear distance between p and q.

Definition 3 [1] *Given a point s, a region R has the uniqueness property with respect to s if for every pair of points p, $q \in R$, $||pq|| < max(||sp||, ||sq||)$. A partition of space into a finite set of disjoint regions is said to have the uniqueness property if each of its regions has the uniqueness property.*

Fig. 2(a) and Fig. 2(b) describes octant partition for a pin vertex and an obstacle corner, respectively. It is proved in [1] that octant partition exhibits the *uniqueness property*. Imagine three points s, p and q such that $||sp|| < ||sq||$ where points p and q lie in R_i of s. As R_i has the *uniqueness property*, it implies $||pq|| < ||sq||$. Since the longest edge



(a) Pin vertex (b) Obstacle corner

Figure 2: Octant partition for a pin vertex and an obstacle corner

of any cycle should not be included in a MST, we can still guarantee that a MST exists in an OASG that does not include edge (s, q).

Another interesting 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 $x+y = c$; in regions R_3,R_4,R_7,R_8, they are described by the form $x - y = c$. Now this property can be exploited when we generate an obstacle-avoiding spanning graph.

Algorithm: **OASG generation for R_1**

1 $A_{active} = A_{bottom} = A_{left} = \emptyset$
2 **for** all $v \in V$ in increasing $(x + y)$ order
3 $\quad S(v) = \emptyset$
4 $\quad$ **for** all $u \in A_{active}$ which have v in their R_1 **do**
5 $\quad\quad$ Add u to $S(v)$
6 $\quad$ **end for**
7 $\quad$ Connect v to the nearest point $u^* \in S(v)$ such that $e(u^*, v)$ is not completely blocked by obstacle boundaries in A_{bottom} and A_{left}
8 $\quad$ Delete all points in $S(v)$ from A_{active}
9 $\quad$ **if** v is a bottom left corner **then**
10 $\quad\quad$ Add the bottom boundary containing v to A_{bottom} and the left boundary containing v to A_{left}
11 $\quad$ **else if** v is a top left corner **then**
12 $\quad\quad$ Delete the left boundary containing v from A_{left}
13 $\quad$ **else if** v is a bottom right corner **then**
14 $\quad\quad$ Determine the bottom boundary B containing v
15 $\quad\quad$ Delete B from A_{bottom}
16 $\quad\quad$ Delete from A_{active} all points which are completely blocked by B
17 $\quad$ **end if**
18 $\quad$ Add v to A_{active}
19 **end for**

Figure 3: Pseudo code for OASG generation algorithm

The pseudo code 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 our algorithm is exactly symmetrical as it does not depend on the direction of scanning. If any point v is the nearest neighbor of u in R_1, it implies that u is the nearest neighbor of v in R_5 which reduces our sweep iterations. 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 to 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 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).

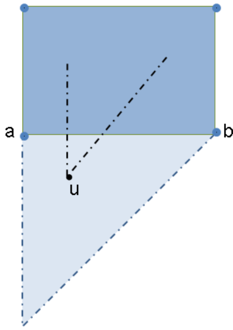


Figure 4: Completely blocked vertices

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.

Lemma 1 *Zhou et al's algorithm [1] is a special case of our OASG generation algorithm*

If we consider a case which has no obstacle, then we can simply ignore the blockage check in line 7 and lines 9 to 17 from the algorithm in Fig. 3. The resulting algorithm would be exactly the same as the algorithm in [1].

3.3 An $O(n \log n)$ Implementation

We first show how to perform the following fundamental operations in the OASG generation algorithm in $O(\log n)$ time: 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 A_{active} which have a given point in their R_1, we need the following lemma.

Lemma 2 **[1]** *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 $x_p - y_p \leq x_q - y_q$.*

We arrange A_{v1} in increasing order of x. Utilizing Lemma 2, to find the subset of points which have v in their R_1, we first find largest x such that $x \leq x_v$. We then proceed in decreasing order of x until $x - y < x_v - y_v$. 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. We use balanced binary search tree to implement A_{active} in order to have $O(\log n)$ query operation.

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 present in the list A_{bottom}. We use the balance binary search tree data structure for A_{bottom} with the y-coordinate of a boundary as a key value. Every attempt to search for an obstacle boundary between y_u and y_v in A_b takes $O(\log n)$ time. 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 regions 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.

The loop from line 2 to line 19 will repeat n times. Lines 2–6 and 8–18 can all be performed in $O(\log n)$ time. To analyze the total run time of line 7, note that each $u \in V$ will only be added to some $S(v)$ at most once in line 5. Then it will be removed from future consideration in line 8. Corresponding to each u added, the blocking of one edge needs to be checked in line 7. Hence totally n edges are checked. In conclusion, the total run time of the algorithm is $O(n \log n)$.

4. OPMST GENERATION

4.1 MTST Generation

After capturing the initial connectivity amongst pin vertices, the next logical step is to extract a minimum terminal spanning tree (MTST) from the OASG that connects all pin vertices and avoid obstacles. Shen et al. [5] and Lin et al. [2] both use an indirect approach for this step. They first construct 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 use 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 80's, Wu et al. [15] 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. [3] 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 [3]. 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 trees, m being the number of pin vertices. The root of every 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 [3] which can be used for linking the trees.

Definition 4 [3] *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.

4.2 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 [3] failed to address this issue. On the other hand, we address this problem by constructing an obstacle penalized minimal spanning tree (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.

5. OAST GENERATION

This step differentiates our algorithm from [2, 3, 5, 9]. We exploit the extremely fast and efficient Steiner tree generation capability of FLUTE [14] 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 efficient 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 location considering obstacles around them.

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

1  If(∃ a completely blocked edge e)
2      /* Refer to Fig. 7 */
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)
10     /* Refer to Fig. 8(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. 8(b) */
14     Refine T' using OA-FLUTE(N") where,
15     N" is a set of pin vertices around e(u,v) in T'
16 Else
17     T' = OA-FLUTE(N) where,
18     N is set of all pin vertices in T'
19 Return T'
```

Figure 5: Pseudo code for the Partition function

Fig. 5 and Fig. 6 describe the pseudo codes for the *Partition* and *OA-FLUTE* functions. It is evident that both functions are recursive functions. Let us first explain the *Partition* function.

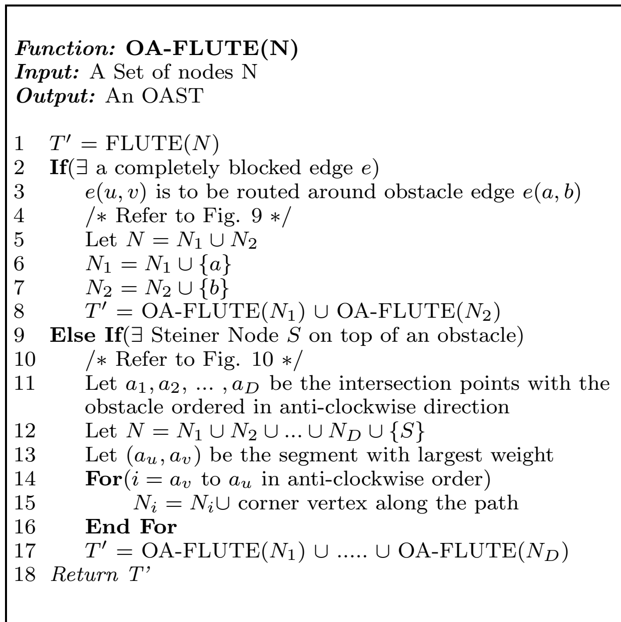
5.1 Partition

The input to the *Partition* function is an OPMST obtained from the last step and the output is an obstacle-aware Steiner tree (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. 7 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. 8(a).

Based on the above mentioned criterion, 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. 8(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



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

1 $T' = \text{FLUTE}(N)$
2 **If**($\exists$ a completely blocked edge e)
3 $e(u, v)$ is to be routed around obstacle edge $e(a, b)$
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 $T' = \text{OA-FLUTE}(N_1) \cup \text{OA-FLUTE}(N_2)$
9 **Else If**($\exists$ Steiner Node S on top of an obstacle)
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)
15 $N_i = N_i \cup$ corner vertex along the path
16 **End For**
17 $T' = \text{OA-FLUTE}(N_1) \cup \ldots.. \cup \text{OA-FLUTE}(N_D)$
18 *Return T'*

Figure 6: Pseudo code for the OA-FLUTE function

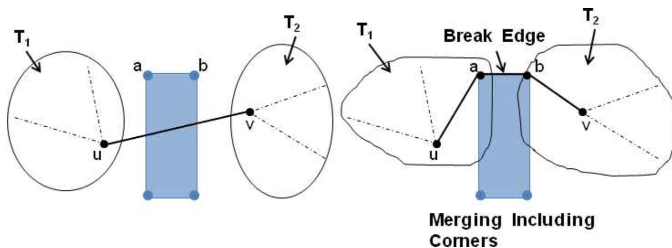


Figure 7: An example illustrating first criterion for partitioning

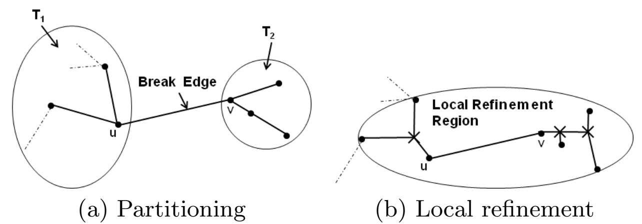
OA-FLUTE in *Partition*, if the partition was performed on an obstacle edge, we simply merge two Steiner trees using the same obstacle edge. In case the partition was performed on the longest edge, we explore an opportunity to further optimize wirelength. We merge the two trees on the longest edge and then search the region around the longest edge to extract neighboring pin vertices, refer to lines 12-15 in Fig. 5 and Fig. 8(b). This refinement is same as the local refinement proposed in [14]. We pass this set of nodes to *OA-FLUTE* for further optimization.

5.2 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, 2) a Steiner node on top of an obstacle. We handle both of these cases differently.

To handle the first case, refer to Fig. 9, we break the Steiner tree into two subtrees including corner points of the obstacle 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.



(a) Partitioning (b) Local refinement

Figure 8: An example illustrating second criterion for partitioning

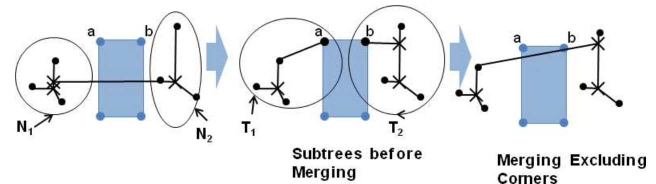


Figure 9: OA-FLUTE: An edge completely blocked by an obstacle

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 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. We then find the longest segment (the light shaded segment (a_3, a_1) in Fig. 10). 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. 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. 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 while merging1. Fig. 9 and Fig. 10, indicate Steiner tree 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.

6. 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)

Benchmark	m	k	Wirelength					Run time(s)			
			Lin [2]	Long [3]	Liang [11]	Liu [12]	Ours	Lin [2]	Long [3]	Liang [11]	Ours
RC01	10	10	27790	26120	25980	26740	25980	0.00	0.00	0.01	0.00
RC02	30	10	42240	41630	42010	42070	42110	0.00	0.00	0.02	0.00
RC03	50	10	56140	55010	54390	54550	56030	0.00	0.00	0.00	0.00
RC04	70	10	60800	59250	59740	59390	59720	0.00	0.00	0.01	0.00
RC05	100	10	76760	76240	74650	75430	75000	0.00	0.00	0.01	0.00
RC06	100	500	84193	85976	81607	81903	81229	0.10	0.08	0.50	0.03
RC07	200	500	114173	116450	111542	111752	110764	0.18	0.09	0.60	0.04
RC08	200	800	120492	122390	115931	118349	116047	0.31	0.15	1.16	0.07
RC09	200	1000	117647	118700	113460	114928	115593	0.40	0.22	1.53	0.09
RC10	500	100	171519	168500	167620	167540	168280	0.20	0.03	0.18	0.02
RC11	1000	100	237794	234650	235283	234097	234416	0.74	0.06	0.83	0.04
RC12	1000	10000	803483	832780	761606	780528	756998	55.09	3.80	186.3	2.65
RT01	10	500	2289	2379	2231	2259	2191	0.03	0.06	0.19	0.01
RT02	50	500	48858	51274	47297	486884	48156	0.05	0.06	0.55	0.02
RT03	100	500	8508	8554	8187	8347	8282	0.10	0.06	0.21	0.03
RT04	100	1000	10459	10534	9914	10221	10330	0.22	0.23	0.37	0.09
RT05	200	2000	54683	55387	52473	53745	54598	0.96	0.66	3.18	0.26
IND1	10	32	632	639	619	626	604	0.00	0.00	0.00	0.00
IND2	10	43	9700	10000	9500	9500	9500	0.00	0.00	0.00	0.00
IND3	10	59	632	623	600	600	600	0.00	0.00	0.00	0.00
IND4	25	79	1121	1130	1096	1095	1129	0.00	0.00	0.00	0.00
IND5	33	71	1392	1379	1360	1364	1364	0.00	0.00	0.00	0.00
RL01	5000	5000	492865	491855	481813	-	483027	106.66	3.58	27.14	3.01
RL02	10000	500	648508	638487	638439	-	637753	159.09	1.27	29.45	1.07
RL03	10000	100	652241	641769	642380	-	640902	153.95	1.08	23.35	1.04
RL04	10000	10	709904	697595	699502	-	697125	195.25	0.97	22.00	1.39
RL05	10000	0	741697	728585	730857	-	728438	217.88	0.96	33.64	1.5
			(1.023)	(1.027)	(0.995)	(1.004)	(1)	891.25(78.45)	13.36(1.196)	331.235(30)	11.36(1)

Table 1: Wirelength and run time comparison. 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 run time

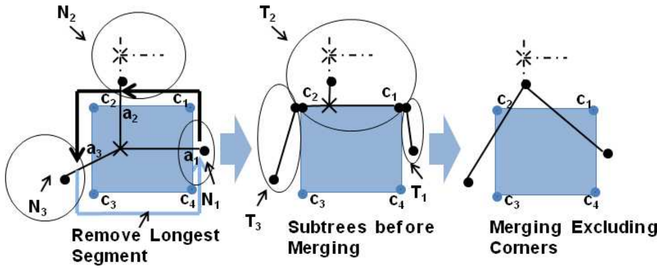


Figure 10: OA-FLUTE: Steiner node is on top of an obstacle

both L-paths are blocked by the same obstacle 3) only one L-path is blocked 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 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 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 smallest one.

7. 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. 11 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%.

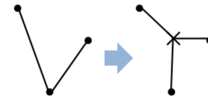


Figure 11: V-shape refinement case and refined output

8. EXPERIMENTAL RESULTS

We implemented our algorithm in C. The experiments were performed on a 3GHz AMD Athlon 64 X2 Dual Core machine. We requested for binaries from Long et al. [3], Lin et al. [2], Liang et al. [11] and ran them on our platform. We could not get binary from Liu et al. [12], which is the most recent work, on time to include in the paper. We report their results as provided in their paper. Table 1 shows Wirelength and CPU run time comparison with them. There

are four sets of benchmarks. Five industrial test cases are from Synopsys(IND1 - IND05), twelve circuits are from [2] (RC01-RC12), five randomly generated benchmark circuits (RT01-RT05) [2] and five large benchmark circuits (RL01-RL05) generated by [3]. We determined experimentally that HIGH THRESHOLD value of 20 works the best.

As shown in last row, column 4, 5 and 7, of Table 1, on an average over all benchmarks, our wirelength results outperform Lin et al. [2] by 2.3% and Long et al. [3] by 2.7% and Liu et al. [12] by 0.4%. But column 6 indicates that our results are 0.5% longer as compared to Liang et al. [11]. This could be attributed to the fact that they use maze routing approach.

Our results also indicate that our algorithm performs better in terms of quality in all higher order (RC07 - RC12)(RL01 - RL05) benchmarks than Liu et al. and we are just 0.1% longer than Liang et al. and 30 times more efficient in CPU run time. We believe that larger benchmarks with more number of pin vertices and more number of obstacles (similar to RC12) are more scalable in industry and we outperform all other existing approaches in these benchmarks due to our highly efficient steiner tree generation tool, OA-FLUTE.

For the run time, we are 20% faster than Long et al. [3] on average. We are 33 times faster than [11] and 88 times faster than [2]. We could not make a direct comparison between the run times of Liu et al. as we could not run their binary on our platform.

We can conclude from the above discussion that existing heuristics improve either run time or wirelength but not both. Our algorithm improves both in terms of quality and run time as compared to the algorithms [5], [2] and [3], of its kind. Also we have the best results both for wirelength and run time for higher-order benchmarks(RC12, RL01-RL05), when compared to [11] and [12] which indicates the applicability of our algorithm to industrial standard circuits.

9. CONCLUSION

In this paper, we have presented FOARS, an efficient algorithm to construct OARSMT based on extremely fast and efficient Steiner tree generation tool called FLUTE. We propose a novel OASG algorithm with linear number of edges. We also propose an obstacle aware version of FLUTE, which generates OAST. Our top-down partition approach empowers OA-FLUTE to handle high degree multi-terminal net and dense obstacle region. Our results indicates that our approach is the best tradeoff for quality and run time. Our experiments prove that FOARS obtains good quality solution with excellent run-time as compared with its peers.

10. ACKNOWLEDGEMENT

We acknowledge Lin et. al [2], Liang et. al [11] and Long et. al [3] for sending us their binaries for comparison and clearing our doubts, if any, with respect to the results.

11. REFERENCES

[1] Hai Zhou, Narendra V. Shenoy, and William Nicholls. Efficient minimum spanning tree construction without delaunay triangulation. *In Proc. of ASP-DAC*, pages 192–197, 2001.

[2] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang. Obstacle-avoiding rectilinear Steiner tree construction based on spanning graphs. *In Proc.of IEEE Transactions on CAD of Integrated Circuits and Systems*, 27(4):643–653, 2008.

[3] Jieyi Long, Hai Zhou, and Seda Ogrenci Memik. EBOARST: An efficient edge-based obstacle avoiding-rectilinear Steiner tree construction algorithm. *In Proc. of IEEE Transactions on CAD of Integrated Circuits and Systems*, 27(12), 2008.

[4] Yu Hu, Zhe Feng, Tong Jing, Xianlong Hong, Yang yang Ge, Xiaodong Hu, and Guiying Yan. FORst: A 3-step heuristic for obstacle-avoiding rectilinear Steiner minimal tree construction. *In Proc. of JICS*, pages 107–116, 2004.

[5] Zion Shen, Chris C. N. Chu, and Ying-Meng Li. Efficient rectilinear Steiner tree construction with rectilinear blockages. *In Proc. of ICCD*, pages 38–44, 2005.

[6] Yu Hu, Tong Jing, Xianlong Hong, Zhe Feng, Xiaodong Hu, and Guiying Yan. An-OARSMan: Obstacle-avoiding routing tree construction with good length performance. *In Proc. of ASP-DAC*, pages 630–635, 2006.

[7] Yiyu Shi, Paul Mesa, Hao Yao, and Lei He. Circuit simulation based obstacle-aware Steiner routing. *In Proc. of DAC*, pages 385–388, 2006.

[8] Pei-Ci Wu, Jhih-Rong Gao, and Ting-Chi Wang. A fast and stable algorithm for obstacle-avoiding rectilinear Steiner minimal tree construction. *In Proc. of ASP-DAC*, pages 262–267, 2007.

[9] Jieyi Long, Hai Zhou, and Seda Ogrenci Memik. An O(n log n) edge-based algorithm for obstacle-avoiding rectilinear Steiner tree construction. *In Proc. of ISPD*, pages 126–133, 2008.

[10] Iris Hui-Ru Jiang, Shung-Wei Lin, and Yen-Ting Yu. Unification of obstacle-avoiding rectilinear Steiner tree construction. *In Proc. of SoCC*, pages 127–130, 2008.

[11] Liang Li and Evangeline F. Y. Young. Obstacle-avoiding rectilinear Steiner tree construction. *In Proc. of ICCAD*, pages 523–528, 2008.

[12] Chih-Hung Liu, Shih-Yi Yuan, Sy-Yen Kuo, and Yao-Hsin Chou. An O(n log n) path-based obstacle-avoiding algorithm for rectilinear Steiner tree construction. *In Proc. of DAC*, pages 314–319, 2009.

[13] F. K. Hwang. On Steiner minimal trees with rectilinear distance. *In Proc. of SIAM J. Appl. Math*, 30:104–114, 1976.

[14] Chris Chu and Yiu-Chung Wong. FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *In Proc. of IEEE Transactions on CAD of Integrated Circuits and Systems*, 27(1):70–83, 2008.

[15] Y. F. Wu, P. Widmayer, and C. K. Wong. A faster approximation algorithm for the Steiner problems in graphs. *In Proc. of Acta Informatica*, 23:223–229, 1986.