2010

Obstacle-Avoiding Rectilinear Steiner Minimal Tree Construction

Gaurav Ajwani
Iowa State University

Obstacle-avoiding rectilinear Steiner minimal tree construction

by

Gaurav Ajwani

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Chris Chu, Major Professor
Randall Geiger
Zhang Zhao

Iowa State University

Ames, Iowa

2010

DEDICATION

I would like to dedicate this report to my family and my friends without whose support I would not have been able to complete my work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Chris Chu for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank Veerendera Allada, my friend and a fellow graduate student who has helped me a lot with his timely guidance and discussions. I would also like to acknowledge the support given by my labmates at ISU VLSI CAD lab Yue Xu, Yanheng Zhang and Jackey Zijun Yan. Last but not the least I would like to thank few of my special friends who have heard me patiently whenever I was stuck in some problem of my thesis and supported me full heartedly.

ABSTRACT

Obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) construction is becoming one of the most sought after problems in modern design flow. In this thesis we present an algorithm to route a multi-terminal net in the presence of obstacles. Ours 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 we use a novel obstacle-avoiding spanning graph (OASG) algorithm which is a generalization of Zhou's spanning graph algorithm without obstacle [18]. The runtime complexity of our algorithm is $O(n \log n)$. Our experimental results indicate that it outperforms Lin et al. [9] by 2.3% in wirelength. It also has 20% faster run time as compared with Long et al. [11], which is the fastest solution till date.

CHAPTER 1. INTRODUCTION

Following placement, the routing process determines the precise paths for nets on the chip layout to interconnect the pins on the circuit blocks or pads at the chip boundary. As we move in to nanometer era the routing is becoming an extremely complex problem.

For routing of a multi-terminal net i.e. a net connecting more than two pins, one naive approach is to decompose each net in to a set of two pin connections, and then route the connections one by one. The quality of routing obtained with this approach depends upon the decomposition step and often leads to suboptimal solutions. A better and more natural approach is to construct a Steiner tree. Steiner Tree is a tree connecting a set of vertices using extra intermediate vertices in order to reduce the length of tree. This tree can also be termed as a multi-terminal net. Rectilinear Steiner minimal tree (RSMT) is a Steiner tree which contains rectilinear lines and achieves a minimum possible wirelength.

RSMT construction is a fundamental problem that has many applications in VLSI design. In early design stages like physical synthesis, floorplanning, interconnect planning and placement it can be used to estimate wireload, routing congestion and interconnect delay. In global and detailed routing stages, it is used to generate the routing topology of each net. Many previous work have addressed this problem.

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. These fixed blocks are termed as immovable obstacles. Hence, routing of multi-terminal nets in the presence of obstacles i.e. OARSMT construction has become a quintessential part of the design and has been studied by many (e.g., [4, 13, 5, 14, 16, 9, 12, 11, 7, 8, 10]). As pointed out by Hwang [6], in the absence of obstacles multi-terminal net routing corresponds to the RSMT problem

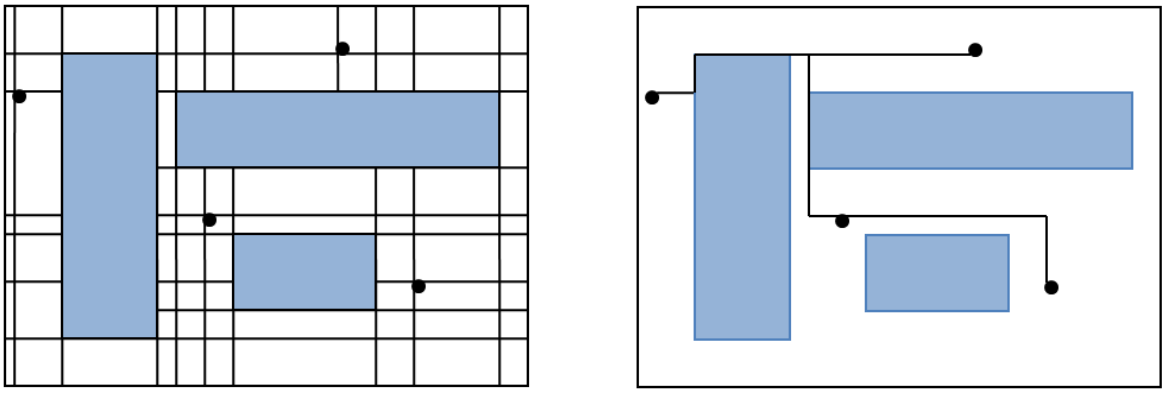
which is NP-complete. The presence of obstacles in the region makes multi-terminal routing problem even harder.

1.1 Related Work

In past several efforts have been made for solving this problem. We can categorize them broadly on the basis of two aspects of their solution for OARSMT construction. First, connectivity information amongst various pins and corners of obstacles is acquired using some kind of a graph. Second, Steiner tree is constructed between pin vertices avoiding obstacles with the help of connectivity graph obtained earlier.

1.1.1 Classification based on Graph Structure

There are three categories based on structure of the graph constructed.

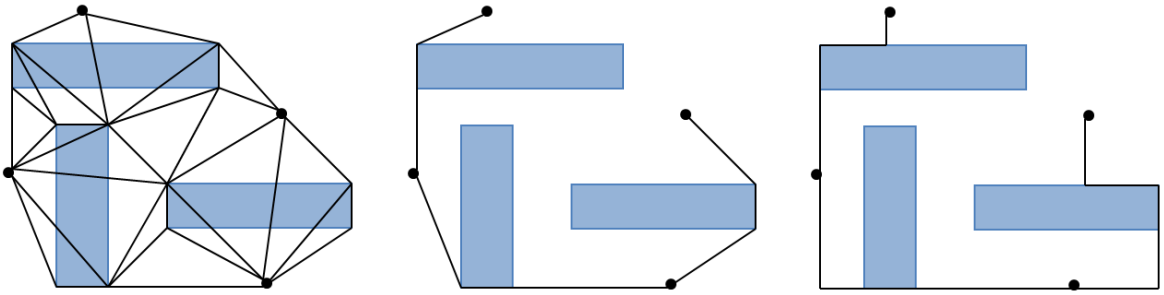


(a) (b)

Figure 1.1 An illustration of OARSMT construction using Escape Graph: (a) Escape Graph (b) OARSMT constructed from Escape Graph

1. **Escape Graph:** Ganley et al. [3] introduced a strong connection graph called Escape Graph. Escape graph is a graph obtained by extending the horizontal and vertical segments intersecting at any pin vertex until the boundary of bounding box. Fig. 1.1(a) depicts an example illustrating an Escape Graph. [3] also proved that all Steiner points in the optimal solution can be found in this graph. It is evident from Fig. 1.1(b) all the

Steiner points lie on escape graph shown in Fig. 1.1(a). Shi et al. [14], Hu et al. [5] [4] and Liang et al. [8] used escape graph to capture connectivity information. The problem with this technique is that the number of edges in this graph tends to be of the order of $O(n^2)$ which makes them slower than other algorithms.



(a) (b) (c)

Figure 1.2 OARSMT construction using Delaunay Triangulation: (a) Delaunay Triangulation Graph (b) Obstacle-avoiding minimum spanning tree (OAMST) (c) OARSMT constructed from OAMST

2. **Delaunay Triangulation:** A Delaunay triangulation DT(P) for a set of P vertices in a plane is a triangulation such that no vertex in P is inside the circumcircle of any triangle in DT(P). There is an interesting property associated with delaunay triangulation, a minimum spanning tree is a subset of delaunay triangulation for a set of points P. Jiang et al. [7] exploited this property to obtain connectivity graph using delaunay triangulation. In presence of obstacles a DT graph can also lead to $O(n^2)$ edges which creates the same difficulty as escape graph.

3. **Obstacle Avoiding Spanning Graph:** [13, 9, 16, 12, 11] are based on various forms of obstacle-avoiding spanning graphs. Shen et al.[13] proposed a form of OASG that only contains a linear number of edges which is also adopted in [16]. The construction of OASG is discussed in detail in later chapter. The property of OASG that makes it useful for creating connectivity graph is that an OASG must contain at least on minimum spanning tree. Later Lin et al.[9] 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. [12, 11] 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.

1.1.2 Classification based on Steiner Tree Construction

We can also categorize our antecedants by the second aspect in their solution mentioned above.

1. **Ant Colony Optimization:** Hu et al. [5] [4] and Wu et al. [16] use ant colony optimization (ACO) running on Escape Graph or Track Graph to form steiner tree. The basic motivation of ACO is to mimic the cooperative behavior of ants to achieve complex computations which consists of multiple iterations. In each iteration one or more ants are allowed to execute a movement, leaving behind a pheromone trail for other to follow. An ant traces out a single path, probabilistically selecting only one edge at a time (in a graph), until an entire solution is obtained. Each separate path can be assigned a cost metric and the sum of all the individual costs defines the function to be minimized by ACO. Heuristics indicate that this approach is quite slow in terms of CPU runtime and does not yield good quality in terms of wirelength as compared with other methods.

2. **Maze Routing:** Maze routing based approaches to find obstacle-avoiding path for a 2-pin net are very common. The multi-terminal variant of this approach is also there but considered very slow in terms of complexity. Recently Liang et al. [8] proposed a maze routing approach running on simplified Hannan grid which is similar to escape graph. However their results indicate that CPU runtime as obtained by them is still very slow as compared with other approaches.

3. **Minimum spanning tree:** Shen et al. [13] and Lin et al. [9] propose refining obstacle-avoiding minimum spanning tree (OAMST) to obtain Steiner tree. To create OAMST they suggest building complete graph between pin vertices and then replacing the edges by

their shortest path obtained from OASG. Although this methodology appeals in general, constructing complete graph results in higher order complexity, which is discouraging. Long et al. [12] improved their approach by using the extended version of Dijkstra and Kruskal's algorithm on OASG to obtain Minimum Terminal Spaning Tree (MTST) and later refining it to form Steiner Tree. Their approach has been the fastest till now but has worst wirelength than [9] which seems to be a concern.

1.2 Our Contribution

In this work, we develop a new $O(n \log n)$ time algorithm for OARSMT generation by leveraging FLUTE[2]. 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. In this thesis, we present a novel octant approach to generate an $O(n)$-edge OASG with more desirable properties.

Different from [13, 9, 12, 11] 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 [9] 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[12, 11] 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.

CHAPTER 2. ALGORITHM

Our OARSMT construction algorithm can be distinctly divided in five stages.

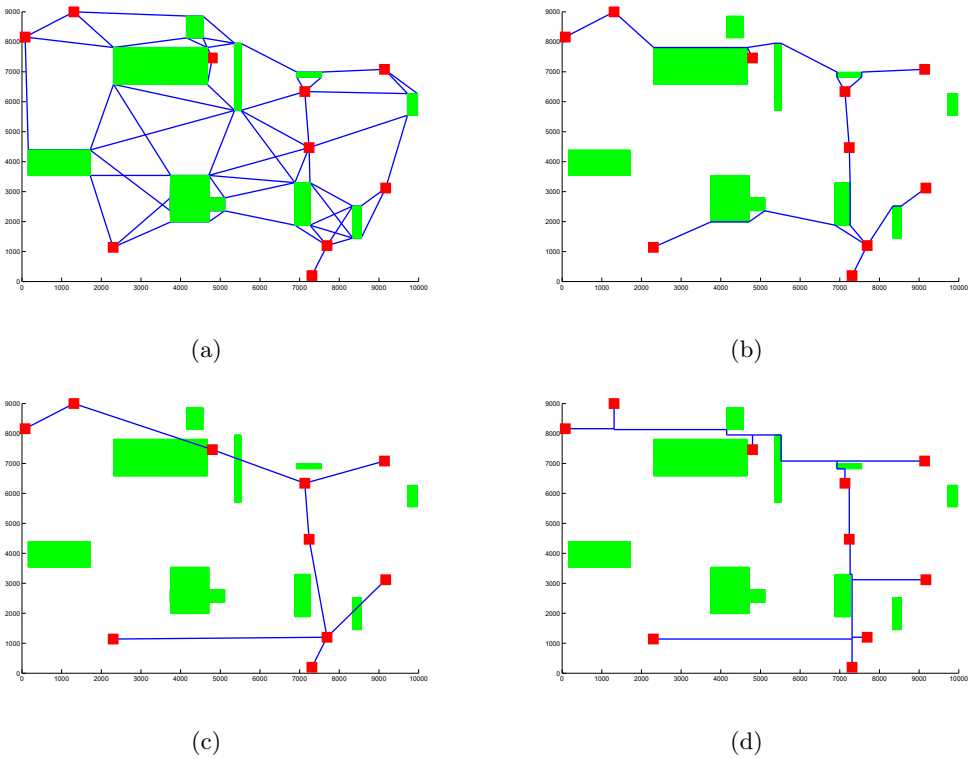
Stage 1: **OASG Generation**. First, we obtain the connectivity information between the pins and obstacle corner vertices using a novel octant OASG generation algorithm. Section 2.1 describes the OASG algorithm in detail.

Stage 2: **OPMST Generation**. Based on the OASG, we construct a minimum terminal spanning tree (MTST) using the approach mentioned in [17] and then obtain an obstacle penalized minimal spanning tree (OPMST) from the MTST. Section 2.2 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 2.3 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 2.4 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 2.5.



(a) (b)

(c) (d)

Figure 2.1 Outputs at various stages for benchmark RC01: (a) OASG (b) MTST (c) OPMST (d) OARSMT.

2.1 OASG Generation

Definition 1 *Given a set of pin vertices P and a set of corners C of obstacles O, an obstacle avoiding spanning graph (OASG) is undirected graph on vertex set $P \cup C$, where no edge intersects with an obstacle in O and it contains at least one minimum spanning tree.*

2.1.1 Previous Approaches

Zhou et al. [18] 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 ev-*

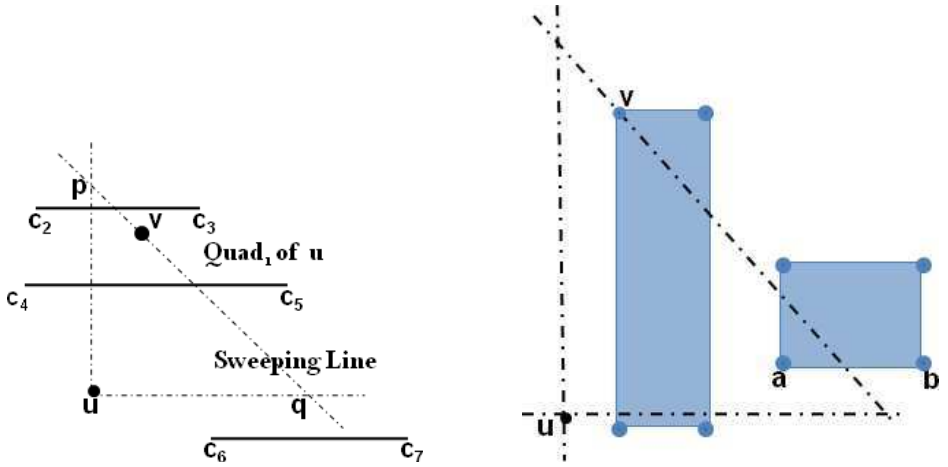
ery monotonic Manhattan path connecting u and v intersects with a boundary of b.

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. [13] 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. [12] recently described a novel quadrant approach which is a modified version of [18] for OASG generation with a linear number of edges in $O(n \log n)$ time. They suggested scanning along $\pm 45^\circ$ 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 [18]. 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.

According to Lemma 1 in [12] the Manhattan connection between vertex (u, v) is considered completely blocked by an obstacle boundary (a, b) if y-coordinate of (a, b) lies between y-coordinates of v and u and (a, b) is present in active obstacle boundary list. In other words we can say that (a, b) lies inside $\triangle pqu$, refer Fig.2.2(a) However, we found that there is a fault in this lemma. Consider a scenario described in Fig.2.2(b). Following explanation above edge (u, v) will be blocked by an obstacle (a, b). However, (a, b) does not actually block (u, v). The result of such incorrect blockage would imply that u will not be connected to its nearest neighbor and would remain active i.e., it has not found its neighbor in Quad 1. This fault can cause u to form an edge in further iterations which can overlap with an obstacle violating



(a) (b)

Figure 2.2 Bug in algorithm proposed in [12] (a) Taken from [12] (b) Specific scenario for lemma 1 in [12]

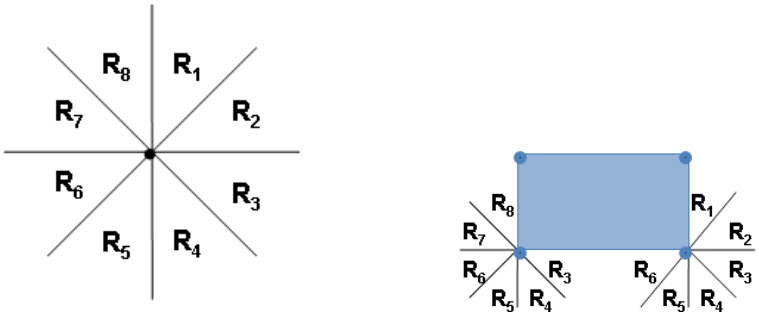
Definition 1. We reported this bug to them and they came up with a fix in [11]. In that they mentioned checking for x-coordinate of Bottom Left corner of obstacle to be on left of u which seems to be correct solution for the above mentioned bug.

To summarize their algorithm has following 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 [18] 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.

2.1.2 Our Approach for OASG

Looking at the above mentioned issues we conceived that rather than modifying Zhou et al's [18] 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 [18] *Given a point s, a region R has the uniqueness property with respect to s if for every pair of points p, q ∈ 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.*



(a) Pin vertex (b) Obstacle corner

Figure 2.3 Octant partition for a pin vertex and an obstacle corner

Fig. 2.3(a) and Fig. 2.3(b) describes octant partition for a pin vertex and an obstacle corner, respectively. It is proved in [18] 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 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.

The pseudo code for OASG generation for R_1 is provided in Fig. 2.4. 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

Algorithm: **OASG generation for** R_1

```
1   A_active = A_bottom = A_left = ∅
2   for all v ∈ V in increasing (x + y) order
3     S(v) = ∅
4     for all u ∈ A_active which have v in their R_1 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
```

Figure 2.4 Pseudo code for OASG generation algorithm

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).

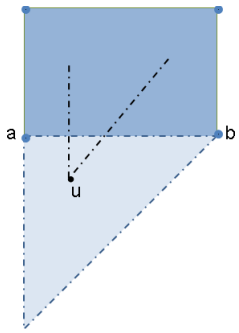


Figure 2.5 Completely blocked vertices

To explain lines 13 to 17, let us refer to Fig. 2.5 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 [18] 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. 2.4. The resulting algorithm would be exactly the same as the algorithm in [18].

2.1.3 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 [18] *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. 2.5). 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 $O(n \log n)$.

2.2 OPMST Generation

Based on OASG generated in previous step, we create an obstacle penalized minimum spanning tree (OPMST). An OPMST is an MST connecting a set of pin vertices, where the weight of any edge between two pins is the length of obstacle avoiding path between them. Constructing an OPMST is a two step process.

2.2.1 MTST Generation

A minimum terminal spanning tree (MTST) is a tree which connects all pin vertices and has the smallest possible length. Since MTST is directly obtained from OASG, it is inherently obstacle avoiding tree. Shen et al. [13] and Lin et al. [9] 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. [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. [11] adopted their approach to solve the problem on the OASG.

Our approach is based on the extended Dijkstra's algorithm and the extended Kruskal's algorithm as defined in [11]. 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.

```
Function: Extended-Dijkstra(G)
Input: A non-negative weighted graph G
Output: A Forest of m trees

1   Heap H_v = φ
2   For Each(vertex u of G)
3       Set u.dist to 0 if u is a pin vertex, +∞
4       H_v.insert(u, u.dist)
5       u.parent = u
6       Make-Set(u)
7   End
8   While H_v is not empty Begin
9       u = H_v.extractMin()
10      Set-Union(u, u.parent)
11      For Each edge e(u, v) of G Begin
12        If v.dist > u.dist + e.length Begin
13          v.dist = u.dist + e.length
14          v.parent = u
15          H_v.decreaseKey(v)
16        End
17      End
18  End
19  For Each vertex u of G Begin
20      u.root = Find-Set(u)
21  End
```

Figure 2.6 Pseudo code for the Extended-Dijkstra [11]

Fig 2.6 describes pseudo code for Extended-Dijkstra Algorithm. This pseudo code is taken from [11] and included here for completion. It is same as Dijkstra algorithm except for a difference in line 3 where every pin vertex is explicitly assigned with 0 value to its distance parameter i.e. making it 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 H_{be} is used to store the weights of all possible edges termed as *bridge edges* in [11] which can be used for linking the trees.

```
Function: Extended-Kruskal(G)
Input: A non-negative weighted graph G
       A Forest of m trees
Output: MTST, T_merge

1  Heap H_be = φ
2  Merging Tree T_merge = φ
3  For Each edge e(u, v) of G
4     If u.root ≠ v.root Begin
5        H_be.insert(e, u.dist + e.length + v.dist)
6     End
7  End
8  While(H_be is not empty Begin)
9     e(u, v) = H_be.extractMin()
10    s_1 = Find-Set(u)
11    s_2 = Find-Set(v)
12    If s_1 ≠ s_2 Begin
13       Connect MTST edge e_MTST(u.root, v.root)
14       s = Set-Union(s_1, s_2)
15       s.edge = e_MTST
16       T_merge.merge(s, s_1, s_2)
17       End
18    End
19 End
```

Figure 2.7 Pseudo code for the Extended-Kruskal [11]

Definition 4 [11] *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. Fig. 2.7 describes pseudo code for extended kruskal's algorithm, it is taken from [11] and included here for completion.

2.2.2 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 [11] 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.

2.3 OAST Generation

This step differentiates our algorithm from [13, 9, 12, 11]. We exploit the extremely fast and efficient Steiner tree generation capability of FLUTE [2] 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.

Fig. 2.8 and Fig. 2.9 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.

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

1  If(∃ a completely blocked edge e)
2      /* Refer to Fig. 2.10 */
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. 2.11(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. 2.11(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 2.8 Pseudo code for the Partition function

2.3.1 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. 2.10 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 heuristically determined that including corner vertex at this stage improves the overall wirelength but we cannot deny that it also restricts the edge to go around the obstacle in one specified direction. We know that *OA-FLUTE* can efficiently construct a tree when the number of nodes is less

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)
3 $e(u, v)$ is to be routed around obstacle edge $e(a, b)$
4 /* Refer to Fig. 2.12 */
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 on top of an obstacle)
10 /* Refer to Fig. 2.13 */
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 $T' =$ OA-FLUTE(N_1) $\cup$ ..... $\cup$ OA-FLUTE(N_D)
18 *Return T'*

Figure 2.9 Pseudo code for the OA-FLUTE function

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. 2.11(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. 2.10. 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. 2.11(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

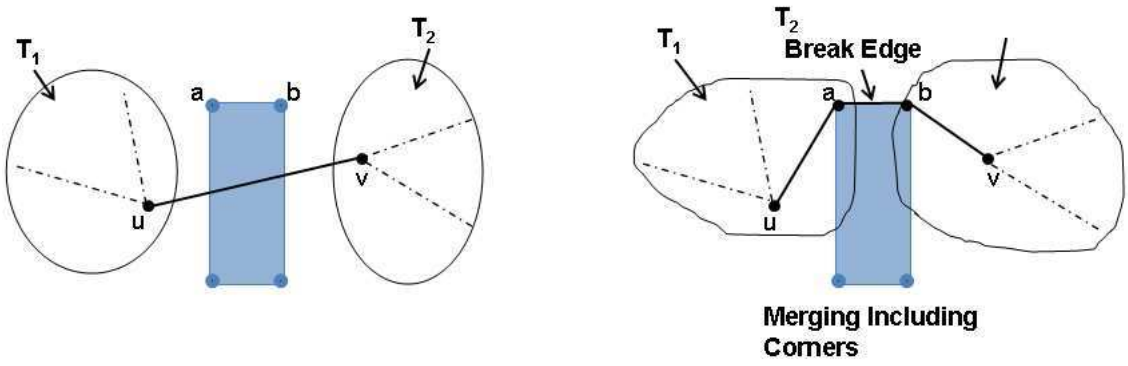
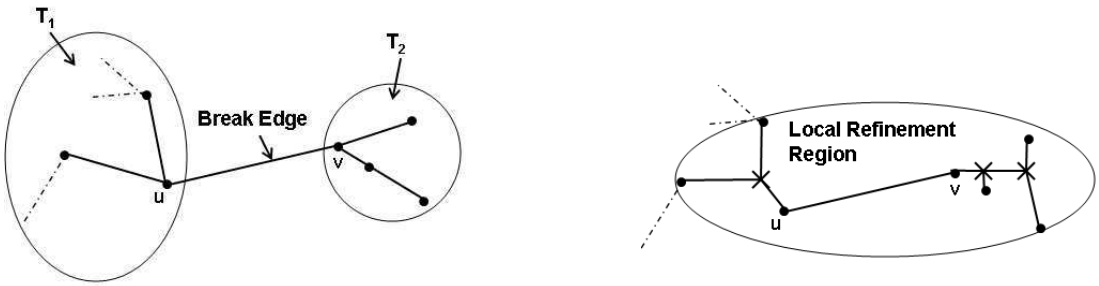


Figure 2.10 An example illustrating first criterion for partitioning



(a) Partitioning (b) Local refinement

Figure 2.11 An example illustrating second criterion for partitioning

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. 2.8 and Fig. 2.11(b). This refinement is same as the local refinement proposed in [2]. We pass this set of nodes to *OA-FLUTE* for further optimization. straightforward to replace the tree obtained after *OA-FLUTE* in the original tree.

2.3.2 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

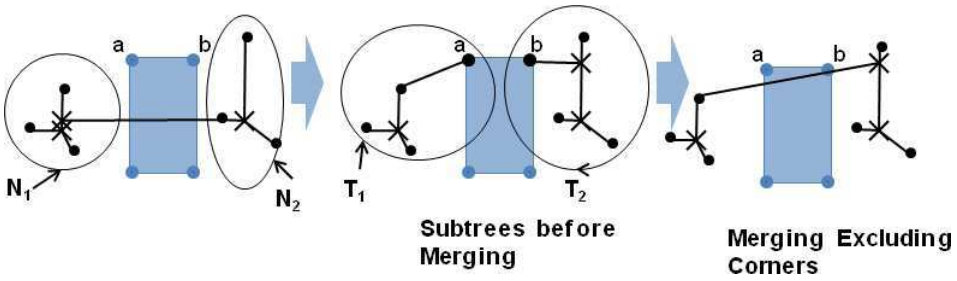


Figure 2.12 OA-FLUTE: An edge completely blocked by an obstacle

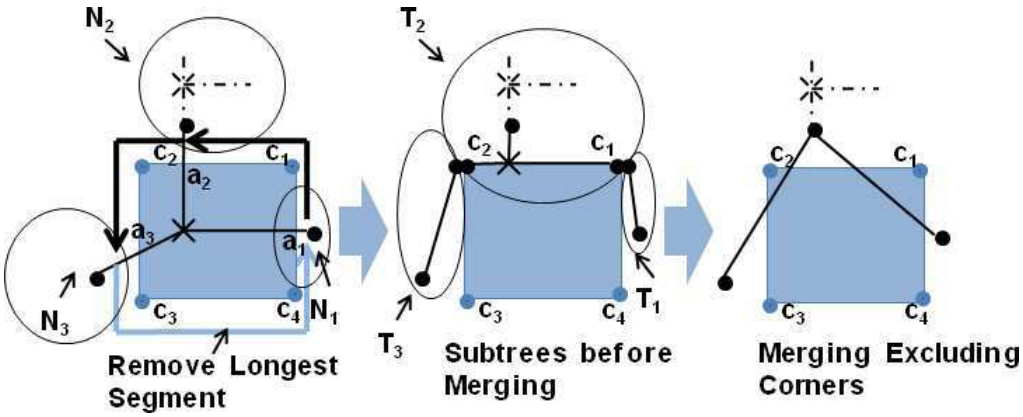


Figure 2.13 OA-FLUTE: Steiner node is on top of an obstacle

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. 2.12, 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. Now it can give an impression that this is exactly the same partition as the one performed in partition function. However, in partition function we included the corner vertices but in OA-FLUTE our goal is to determine the location

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. 2.13 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. 2.13. We then find the longest segment (the light shaded segment (a_3, a_1) in Fig. 2.13). 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. 2.13. 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. 2.12 and Fig. 2.13, 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.

2.4 OARSMT Generation

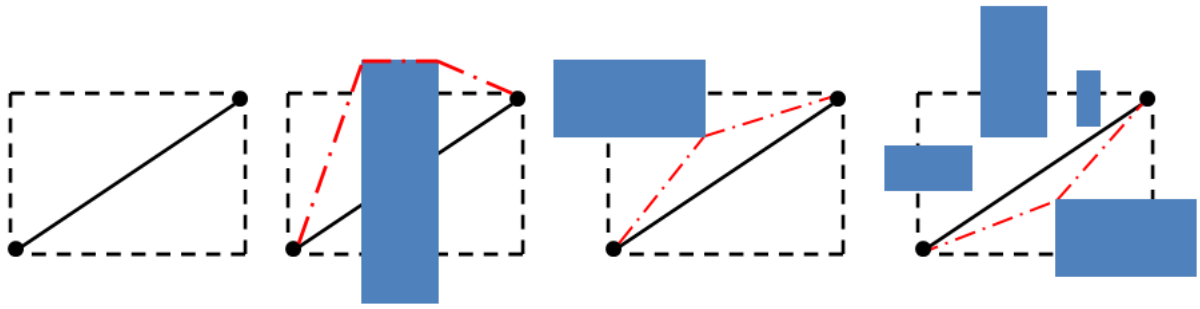


Figure 2.14 Different scenarios for OARSMT generation algorithm

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 4) both L-paths are blocked but not by the same obstacle. We discuss these scenarios one by one in the following paragraphs. The Fig 2.14 illustrates different scenario explained in paragraph above.

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.

2.5 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. 2.15 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%.

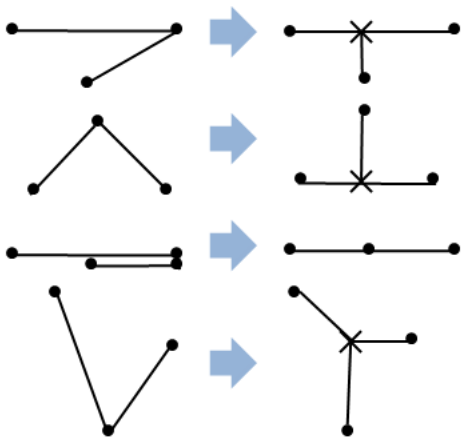


Figure 2.15 V-shape refinement case and refined output

CHAPTER 3. EXPERIMENTAL RESULTS

3.1 Using benchmarks with obstacles

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. [11], Lin et al. [9], Liang et al. [8] and ran them on our platform. We could not get binary from Liu et al. [10], which is the most recent work, on time to include in the paper. We report their results as provided in their paper. Table 3.1 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 [9] (RC01-RC12), five randomly generated benchmark circuits (RT01-RT05) [9] and five large benchmark circuits (RL01-RL05) generated by [11]. We determined experimentally that HIGH THRESHOLD value of 20 works the best. obstacles we used a simpler implementation with a sorted list for obstacles. This could result in a complexity of O(n^2). But, we were still able to achieve fastest results as compared to all approaches mentioned in Section 1.

As shown in last row, column 4, 5 and 7, of Table 3.1, on an average over all benchmarks, our wirelength results outperform Lin et al. [9] by 2.3% and Long et al. [11] by 2.7% and Liu et al. [10] by 0.4%. But column 6 indicates that our results are 0.5% longer as compared to Liang et al.[8]. 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

Benchmark	m	k	Wirelength				
			Lin[9]	Long[11]	Liang[8]	Liu[10]	Ours
RC01	10	10	27790	26120	25980	26740	25980
RC02	30	10	42240	41630	42010	42070	42110
RC03	50	10	56140	55010	54390	54550	56030
RC04	70	10	60800	59250	59740	59390	59720
RC05	100	10	76760	76240	74650	75430	75000
RC06	100	500	84193	85976	81607	81903	81229
RC07	200	500	114173	116450	111542	111752	110764
RC08	200	800	120492	122390	115931	118349	116047
RC09	200	1000	117647	118700	113460	114928	115593
RC10	500	100	171519	168500	167620	167540	168280
RC11	1000	100	237794	234650	235283	234097	234416
RC12	1000	10000	803483	832780	761606	780528	756998
RT01	10	500	2289	2379	2231	2259	2191
RT02	50	500	48858	51274	47297	48684	48156
RT03	100	500	8508	8554	8187	8347	8282
RT04	100	1000	10459	10534	9914	10221	10330
RT05	200	2000	54683	55387	52473	53745	54598
IND1	10	32	632	639	619	626	604
IND2	10	43	9700	10000	9500	9500	9500
IND3	10	59	632	623	600	600	600
IND4	25	79	1121	1130	1096	1095	1129
IND5	33	71	1392	1379	1360	1364	1364
RL01	5000	5000	492865	491855	481813	-	483027
RL02	10000	500	648508	638487	638439	-	637753
RL03	10000	100	652241	641769	642380	-	640902
RL04	10000	10	709904	697595	699502	-	697125
RL05	10000	0	741697	728585	730857	-	728438
			(1.023)	(1.027)	(0.995)	(1.004)	(1)

Table 3.1 Wirelength comparison for OARSMT benchmarks; 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.

Benchmark	m	k	Runtime(s)			
			Lin[9]	Long[11]	Liang[8]	Ours
RC01	10	10	0.00	0.00	0.01	0.00
RC02	30	10	0.00	0.00	0.02	0.00
RC03	50	10	0.00	0.00	0.00	0.00
RC04	70	10	0.00	0.00	0.01	0.00
RC05	100	10	0.00	0.00	0.01	0.00
RC06	100	500	0.10	0.08	0.50	0.03
RC07	200	500	0.18	0.09	0.60	0.04
RC08	200	800	0.31	0.15	1.16	0.07
RC09	200	1000	0.40	0.22	1.53	0.09
RC10	500	100	0.20	0.03	0.18	0.02
RC11	1000	100	0.74	0.06	0.83	0.04
RC12	1000	10000	55.09	3.80	186.3	2.65
RT01	10	500	0.03	0.06	0.19	0.01
RT02	50	500	0.05	0.06	0.55	0.02
RT03	100	500	0.10	0.06	0.21	0.03
RT04	100	1000	0.22	0.23	0.37	0.09
RT05	200	2000	0.96	0.66	3.18	0.26
IND1	10	32	0.00	0.00	0.00	0.00
IND2	10	43	0.00	0.00	0.00	0.00
IND3	10	59	0.00	0.00	0.00	0.00
IND4	25	79	0.00	0.00	0.00	0.00
IND5	33	71	0.00	0.00	0.00	0.00
RL01	5000	5000	106.66	3.58	27.14	3.01
RL02	10000	500	159.09	1.27	29.45	1.07
RL03	10000	100	153.95	1.08	23.35	1.04
RL04	10000	10	195.25	0.97	22.00	1.39
RL05	10000	0	217.88	0.96	33.64	1.5
			891.25(78.45)	13.36(1.196)	331.235(30)	11.36(1)

Table 3.2 CPU Runtime comparison for OARSMT benchmarks; The values in the last row are summation for all benchmakrs, and normalized over our results.

Benchmark	m	k	Wirelength				
			Long[11]	Liang[8]	FLUTE-2.5[2]	FLUTE-3.0	Ours
RC01	10	0	25290	25290	25290	25290	25290
RC02	30	0	40100	40630	39920	39920	39920
RC03	50	0	52560	52440	53440	52880	53050
RC04	70	0	55850	55720	57020	55300	55380
RC05	100	0	72820	71820	73370	73220	72170
RC06	100	0	77886	78068	80057	77171	77633
RC07	200	0	106591	107236	109232	106743	106581
RC08	200	0	109625	109059	112787	108495	108815
RC09	200	0	109105	108101	112460	107729	108106
RC10	500	0	164940	164450	170270	169380	164040
RC11	1000	0	233743	235284	245325	231730	233600
RC12	1000	0	755332	764956	798742	748464	755060
RT01	10	0	1817	1817	1817	1817	1817
RT02	50	0	44930	46109	45291	44685	44416
RT03	100	0	7677	7777	7811	7652	7745
RT04	100	0	7792	7826	7826	7827	7792
RT05	200	0	43335	43586	44809	42943	43026
IND1	10	0	614	619	604	604	604
IND2	10	0	9100	9100	9100	9100	9100
IND3	10	0	590	590	587	587	587
IND4	25	0	1092	1092	1102	1102	1102
IND5	33	0	1314	1304	1307	1307	1307
RL01	5000	0	472392	473905	501480	471137	472757
RL02	10000	0	637131	641722	674042	635287	636689
RL03	10000	0	641289	650343	674950	639183	640342
RL04	10000	0	697712	699617	742070	695526	697086
RL05	10000	0	728595	730857	778313	726245	728438
			(1.002)	(1.006)	(1.026)	(0.998)	(1)

Table 3.3 Wirelength comparison for RSMT benchmarks; m is the number of pin vertices; Note that it does not have any obstacles, hence k is 0 for all benchmarks.

benchmarks due to our highly efficient steiner tree generation tool, OA-FLUTE.

For the run time shown in Table 3.2, we are 20% faster than Long et al. [11] on average. We are 33 times faster than [8] and 88 times faster than [9]. 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.s Our algorithm improves both in terms of quality and run time as compared to the algorithms [13], [9] and [11], 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 [8] and [10] which indicates the applicability of our algorithm to industrial standard circuits.

Benchmark	m	k	Runtime(s)				
			Long[11]	Liang[8]	FLUTE-2.5[2]	FLUTE-3.0	Ours
RC01	10	0	0	0	0.000051	0.000012	0.000317
RC02	30	0	0	0	0.000203	0.000076	0.000527
RC03	50	0	0	0	0.000319	0.000322	0.001242
RC04	70	0	0	0	0.000408	0.000446	0.001566
RC05	100	0	0	0	0.000404	0.000625	0.002019
RC06	100	0	0	0	0.000486	0.000664	0.002071
RC07	200	0	0.012	0.02	0.000731	0.001545	0.004096
RC08	200	0	0.004	0.03	0.000684	0.001553	0.005261
RC09	200	0	0.016	0.02	0.000717	0.001523	0.004074
RC10	500	0	0.024	0.17	0.001605	0.006988	0.014066
RC11	1000	0	0.068	0.7	0.003444	0.061217	0.032900
RC12	1000	0	0.044	0.75	0.003498	0.03568	0.032603
RT01	10	0	0.012	0	0.000089	0.000020	0.000442
RT02	50	0	0	0	0.000329	0.000330	0.001399
RT03	100	0	0	0	0.000480	0.000617	0.002495
RT04	100	0	0	0	0.000383	0.000644	0.002182
RT05	200	0	0.04	0.02	0.000715	0.001652	0.004243
IND1	10	0	0	0	0.000900	0.000016	0.000412
IND2	10	0	0	0	0.000830	0.000014	0.000397
IND3	10	0	0	0	0.000096	0.000016	0.000424
IND4	25	0	0	0	0.000170	0.000052	0.000585
IND5	33	0	0	0	0.000198	0.000083	0.000612
RL01	5000	0	0.388	11.39	0.055283	0.463103	0.361255
RL02	10000	0	0.956	32.45	0.259335	1.739664	1.167446
RL03	10000	0	0.956	33.04	0.260298	1.718425	1.149940
RL04	10000	0	0.992	32.26	0.255205	1.003005	1.442818
RL05	10000	0	1.052	34.52	0.257259	1.232832	1.556124
			4.528(0.78)	145.37(25)	6.27(1.082)	1.104(0.19)	5.79(1)

Table 3.4 CPU Runtime comparison for RSMT benchmarks

3.2 Using benchmarks without Obstacles

As our algorithm performs efficiently for all benchmarks with *obstacles* we were curious to compare its performance for benchmarks without *obstacles*. The problem formulates to RSMT construction as mentioed above and has been studied by numerous researchers in the past. The work by Wong et al. [15] shows that FLUTE 3.0 has best results for this problem.

One of the goals of our research was also to create a tool which can work efficiently in presence as well as absence of obstacles. In order to excercise this, we removed obstacles from our previous benchmarks and compared our results with results from Liang et al. [8] , Long et al. [11], FLUTE-3.0 [15] and FLUTE-2.5 [2]. Table 3.3 depicts comparison of wirelength with various work listed above, the last row is an average of our result compared with the results in respective column over all the benchmarks. It can be seen that we perform 2.6% better as compared with FLUTE-2.5. Our results are also better as compared with [11] and [8] indicating our approach can be used as a complete solution for Steiner tree construction in

absence as well as presence of obstacles. Column 6 shows our results are slightly worst than FLUTE-3.0 which has the best results for this category.

CPU runtime results in Table 3.4 enumerate that FLUTE-2.5 is much better in terms of execution time as compared with all the approaches, the last row is summation of total runtime over all the benchmarks. Our results in Column 7 as compared with all Liang et al. [8], the maze routing approach, are 25X faster which is not surprising considering that maze routing approach is generally time consuming. We are also slower 22% slower as compared with Long et al. [11]. Since our work is based on FLUTE-2.5, we believe that with code optimization we should be able to achieve same speed as FLUTE-2.5 for benchmarks without obstacles.

CHAPTER 4. CONCLUSION

In this thesis we have presented an algorithm to construct obstacle-avoiding rectiliear Steiner tree (OARSMT) based on extremely fast and efficient Steiner tree generation tool called FLUTE. We propose a novel octant based OASG algorithm with linear number of edges which possess certain desirable properties as compared with previous heuristics. We also propose an obstacle aware version of FLUTE, OA-FLUTE which can be used efficiently for low-degree nets. To handle high degree nets and dense obstacle region we use a top-down partition approach with FLUTE. We also illustrate the capacity of our algorithm to handle benchmarks which does not contain obstacles and observe that it performs well as compared with other algorithms of its kind.

Summarizing we can say that OARSMT problem has become more important then ever for modern nanometer IC designs which need to consider numerous routing obstacles incurred from pre-routed nets, IP blocks etc. It is required for an algorithm to handle Steiner tree construction in presence as well as absence of obstacles with good quality and fast CPU runtime. Our experiments prove that our algorithm obtains a good quality solution with excellent run-time as compared with its peers.

A part of this thesis was submitted for a VLSI CAD conference and it got accepted, the paper is supposed to come out in March 2010 but it can be referred here [1].

CHAPTER 5. FUTURE WORK

Our current solution produces good quality results but there is always a room for enhancement. Due to time restriction while implementing our algorithm we used a very basic data structure to represent obstacle information i.e. a sorted list. This data structure could result in runtime complexity of $O(n^2)$ in worst case while checking an edge to be *completely blocked* by any obstacle in the list. Hence, as a future work we propose using a B-tree data structure to represent obstacle list which will guarantee $O(n \log n)$ complexity for all cases and will reduce runtime for our algorithm. For the benchmarks which does not have obstacles our algorithm should theoretically be as fast as FLUTE 2.5 though our results seems to disagree to this. We believe there is a wide scope for code optimization possible in current implementation which can make this possible.

As modern nanometer IC designs are processed layer by layer it is a new challenge for designers to deal with the multi-layer OARSMT (ML-OARSMT) problem where pins between different layers are connected by *vias*. Apart from total wirelength, for ML-OARSMT construction, router has to also take care of constraints such as number of vias, and the DRC rules. Moreover, most of the pins of standard cell are located in lower layers, while many pins of macro cells are located in higher layers. Therefore the router should be able to connect all the pins of a net, no matter on which layers these pins are. Hence, we see a useful and a strong extension of our algorithm to handle ML-OARSMT problem.

APPENDIX

FLUTE

An excerpt taken from [2]

FLUTE is based on a very fast and accurate lookup table algorithm used for RSMT construction. A *net* of degree n is a set of n pins. In [2] author show that the set of all degree-n nets can be partitioned into $n!$ groups according to relative positions of their pins. For each group, the optimal wirelength of any net can be found based on a few vectors called *potentially optimal wirelength vectors* (POWVs). Each POWV corresponds to a linear combination of distances between adjacent pins. Some of the POWVs are pre-computed and stored into a Lookup table. Associated with each POWV, also stored is corresponding Steiner tree, which is termed as *potentially optimal Steiner tree* (POST). To find the optimal RSMT of a net, we just need to compute the wirelengths corresponding to the POWVs for the group the net belongs to, and then return the POST associated with the POWV with the minimum length. This lookup table idea can optimally and efficiently handle low-degree nets (up to degree 9). For high degree nets, the author's propose a net breaking technique to recursively break a net until the table can be used. The runtime complexity of FLUTE with fixed accuracy iis O($nlogn$) for a net of degree n.

BIBLIOGRAPHY

[1] Gaurav Ajwani, Chris Chu, and Wai-Kei Mak. FOARS: Flute based obstacle-avoiding rectilinear steiner tree construction. *To appear In Proc. of ISPD*, 2010.

[2] 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.

[3] Joseph L. Ganley and James P. Cohoon. Routing a multi-terminal critical net: Steiner tree construction in the presence of obstacles. *In Proc. of IEEE ISCAS*, pages 113–116, 1994.

[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] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Yiu-Chung Wong and Chris Chu. A scalable and accurate rectilinear Steiner minimal tree algorithm. *In Proc. of International Symposium on VLSI Design, Automation and Test*, pages 29–34, 2008.

[16] 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.

[17] 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.

[18] 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.