Obstacle-avoiding Rectilinear Steiner Tree Construction

Liang Li and Evangeline F. Y. Young
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Email: {lli,fyyoung}@cse.cuhk.edu.hk

Abstract— **In today's VLSI designs, there can be many blockages in a routing region. The obstacle-avoiding rectilinear Steiner minimum tree (OARSMT) problem has become an important problem in the physical design stage of VLSI circuits. This problem has attracted a lot of attentions in research and several approaches have been proposed to solve this problem effectively. In this paper, we will present a heuristic maze routing based approach to solve this OARSMT problem. It is commonly believed that maze routing based approaches can only handle small scale problems and there is a lack of an effective multi-terminal variant to handle multi-pin nets in practice. We will show in this paper that maze routing based approaches can also handle large scale OARSMT problems effectively. Our approach is based on the searching process as in maze routing and can handle multi-pin nets very well in both solution quality, running time and memory space usage. We have compared our results with those of the previous works and can show that we can out-perform the best previous results on this problem [15] by giving an OARSMT with 2.01% less wire length on average and can make a 27.04% improvement in wire length in comparison with a lower bound of the optimal solution on average, while the running times are all very short and comparable to those in [15]. Besides, due to the flexibility of maze routing, we can handle different kinds of obstacles with different convex or concave rectilinear shapes directly without a need to partition each blockage into a set of rectangular sub-blockages, which will increase the size of the problem.**

I. Introduction

Construction of rectilinear Steiner minimum tree (RSMT) is an important problem in VLSI physical design. It is useful for the detailed and global routing steps, and is important for congestion, wire length and timing estimations during the floorplanning or placement step. This classical problem has long been shown to be NP-complete and has attracted a lot of attentions in research. The original RSMT problem assumes no obstacles in the routing region. In today's VLSI designs, there can be many routing blockages however, like macro cells, IP blocks and pre-routed nets. Therefore, the RSMT problem with blockages, called obstacle avoiding RSMT (OARSMT), has become an important problem in practice and is an interesting problem theoretically. Since the RSMT problem is NP-complete, the OARSMT problem is also NP-complete as it is a generalized version of RSMT.

Clarkson *et al.*[1] considered only 2-pin nets and presented an $O(n(\lg n)^2)$ time algorithm to compute a rectilinear shortest path between two pins avoiding polygonal obstacles, where n is the number of pins and obstacle boundaries. Wu *et al.*[2] introduced a sparse connection graph, called track graph, to reduce the search space for computing the shortest path between two points avoiding obstacles. Ganley *et al.*[3] proposed an algorithm to construct an optimal 3-terminal or 4-terminal OARSMT. Heuristics, called G3S, G4S and B3S, were developed for the cases with less than 20 terminals. Zheng *et al.*[4] proposed an algorithm to find obstacle-avoiding shortest paths using an implicit connection graph approach. Yang *et al.*[5] presented a complicated 4-step heuristic to the OARSMT problem and their approach works well when the terminal number is less than seven and when the obstacles are convex.

More recently, Hu *et al.*[6] developed an efficient hierarchical heuristic called FORst, that partitions all pins into subsets, then connects pins in each subset, and finally constructs an OARSMT using a connection graph [7] based approach. Their method can tackle large scale problems efficiently. Based on an ant colony optimization, Hu *et al.*[8] proposed another non-deterministic local search heuristic, called An-OARSMan, to handle small-scale OARSMT problems with complex obstacles of both concave and convex shapes. This approach can achieve shorter wire length than FORst when the number of terminals is less than 100. Although An-OARSMan is flexible in handling complex obstacles, it takes extremely long running time for large scale designs. CDCTree proposed by Shi *et al.*[11] is based on a current driven circuit model and it can achieve shorter wire length than An-OARSMan. Shen *et al.*[9] proposed a connection graph based approach to solve the OARSMT problem, using the spanning graph in [10] which did not consider obstacles. In their approach, an obstacle-avoiding spanning graph is first constructed and then transformed into an OARSMT. Feng *et al.*[12] proposed a method to construct obstacle-avoiding Steiner tree in an arbitrary λ-geometry by Delaunay triangulation. Its running time complexity is just $O(nlogn)$, where n is the total number of terminals and obstacles.

Very recently, Wu *et al.*[13] presented another algorithm for constructing OARSMTs. Their approach will first find a minimum spanning tree of all the terminals by applying a partitioning method. The segments intersecting with the obstacles will be removed, forming a set of sub-trees. An ant colony optimization based approach is then used to connect the sub-trees back into a single tree which is rectilinearized at the end to given an OARSMT. Hentschke *et al.*[14] presented AMAZE, a fast maze routing based algorithm to build steiner trees. A sharing factor and a path-length factor were introduced to trade-off wire length for delay. However, when dealing with multiple paths, the algorithm adopts traditional ways of selecting just one path from multiple paths with a biasing technique. The most recent work on this problem is by Lin *et al.*[15]. They extended the connection graph based approach in [9] by identifying many "essential" edges which can lead to more desirable solutions in the construction of the obstacle-avoiding spanning graph. They have also improved the OARSMT transformation procedure in [9] significantly, and develop an effective refinement scheme for the U-shaped connections in an OARSMT to further reduce the total wire length. The result obtained by this approach is the most updated one among all the others.

It is commonly believed that the maze routing based approaches are suitable for small scale problems only. The major drawback is the lack of a multi-terminal variant to handle multi-pin nets. Besides, its time complexity and memory usage can grow very large as the routing area expands. In this paper, we will show that maze routing based approaches can also handle large scale OARSMT problems effectively. In our algorithm, to handle multi-pin nets, multiple paths between the pins are kept until all the pins are reached, then a MST based method is used to select between those paths to create an OARSMT. A post-processing step is then performed to further reduce the total wire length. The basic maze routing step is implemented efficiently by using a heap data structure and by propagating on the *simplified Hanan grid* which is formed by the original Hanan grid with all the intersection points lying inside an obstacle deleted. This heap data structure has also helped to handle multi-pin nets very efficiently. Experimental results show that we can out-perform the most updated previous work [15] on this problem in terms of wire length in a very short running time. We can achieve 2.01% less wire length on average and a 27.04% improvement in comparison with a lower bound of the optimal wire length, while the running times are all very short and comparable to those in [15].

In this paper, we will first formulate the problem in section II. Our maze routing based approach will be presented in section III. Experimental results will be shown in section IV before the conclusion in section V.

II. Problem Definition

Definition 1: **OARSMT Problem:** Let $B = \{b_1, b_2, \ldots, b_k\}$ be a set of non-overlapping rectilinear blockages in a 2-dimensional space R. Let $P = \{p_1, p_2, \ldots, p_m\}$ be a set of pins for a m-pin net in R such that no pins $p_i \in P$ lie inside an obstacle $b_j \in B$. Construct a rectilinear Steiner minimum tree connecting all the pins in P to achieve a minimum total length (measured by Manhattan distance), while avoiding intersection with any blockages in R.

A. Properties of OARSMT

In the OARSMT problem, a pin cannot lie inside an obstacle but it can be at the corner or on the boundary of an obstacle. An obstacle cannot overlap with another obstacle, but it can point-touch an obstacle at a corner or line-touch at a boundary. The edge of the OARSMT cannot intersect with a blockage, but it can run along the borders of the obstacles.

III. Our Approach

Our heuristic approach is based on the maze routing method, but our maze routing engine can handle directly multi-pin nets effectively. The efficiency of this routing engine can be further improved by the use of a heap data structure and by propagation on the simplified Hanan grid. By using this multi-pin maze routing approach alone, we can already obtain results better than those generated by the most updated previous work [15]. Then, a fast post-processing step will be applied to scan the paths connected to each pin once to further minimize the total wire length. In the following, we will first explain the general approach we employed to handle multi-pin nets in our maze routing engine. Details of the implementation will be explained in the next sub-section, followed by a description of the post-processing step at the end.

A. Handling of Multi-pin Nets

Given a m-pin net $P = \{p_1, p_2, \ldots, p_m\}$, we will pick one pin to start with. In our algorithm, we will choose the pin whose position is the nearest to the boundary. Assume that $p_{\pi(1)}$ is selected as the first pin. Then, we will propagate from $p_{\pi(1)}$ to find the second pin to connect with. Let $p_{\pi(2)}$ be the second pin we found, i.e., $p_{\pi(2)}$ has the shortest Manhattan distance from $p_{\pi(1)}$ in comparison with others. Note that multiple paths of this length will be found and recorded in our maze router. Then, we will propogate from all these paths to find the third pin, again, recording multiple paths of the same shortest Manhattan length to reach this third pin. This process is repeated until all the pins in P are reached. After this process, we will find all the Steiner points lying on the paths found and record the Manhattan distances between the points (pins and Steiner points). From this information, we will perform once a minimum spanning tree algorithm to connect all the points, followed by a garbage collection step to remove dangling connections which are connected to Steiner points only. An example illustrating our approach to handle multiple pins is shown in Fig. 1. The pseudocode is described as follows.

Algorithm 1 MultiMaze

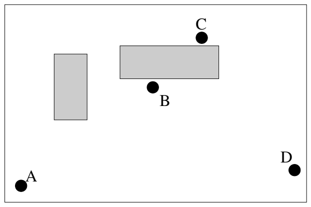
```
1: //Find a route for a $m$-pin net with pins $P = \{p_1, p_2, \ldots, p_m\}$
2: $V = \{p_{\pi(1)}\}$ //Let $p_{\pi(1)}$ be the first pin to start with
3: while some pins are not connected yet do
4:   if this is the first round then
5:     Propagate from the pin in $V$ until reaching another pin $p_j$
6:   else
7:     Continue the propagation until reaching another pin $p_j$
8:   end if
9:   $V = V + \{p_j\}$
10:  Backtrack from $p_j$ to find a set $Q$ of paths with the same
     shortest Manhattan distance connecting to $p_j$
11:  Set the distances of all the points on the paths in $Q$ to zero
12: end while
13: Find a MST $T$ covering all the points in $V$ using some of the
    paths found
14: Remove edges from $T$ until there are no leaf nodes which are
    Steiner points //Garbage collection step
15: Post-process to reduce wire length
```

Our approach is different from traditional maze routing based methods in handling multi-pin nets. Instead of connecting the pins sequentially and decide on the paths once some new pins are reached, multiple paths are kept and path selection is delayed until all the pins are reached and the final route is constructed globally by finding an MST. In order to route fast, the propagation is performed on the simplified Hanan grid and a heap data structure is used such that the current shortest path will always be expanded first. One important technique in our approach to shorten the running time is that after some new paths are found and the distances of the points on them are set to zero, instead of starting the maze routing once again from scratch, we will continue the propagation from the last round. The heap data structure allows us to update the shortest distances at those affected points fast without repeating works which have been done before. We will explain in details the propagation step, the backtracking step and the final post-processing step in the following sub-sections.

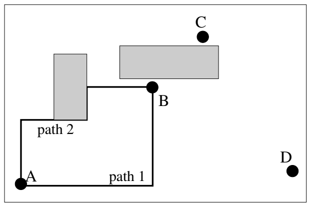
B. Propagation

We can see from the pseudocode of *MultiMaze* that in each round of the propagation process (line 3 to 12), the backtracking step will be performed once and the backtracking result will affect the propagation in the next round. In this section, we will focus on the propagation process, while the backtracking step will be described in the next section. The pseudocode of the propagation process is shown in Algorithm 2.

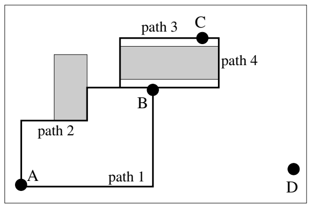
In the propagation process, a heap H is maintained. This heap will store 2-tuples $((x, y), d)$ where (x, y) is the coordinate of a point and d is the distance of a path that can reach (x, y) from some connected points. The heap H will be sorted according to the d values. A 2-dimensional array $dist[\,][\,]$ keeps the shortest distance found so far from a connected point and the value $dist[x][y]$ will be zero if the point (x, y) is connected. The while-loop on line 10 repeats while $min(H) < best$ where $min(H)$ is the shortest distance stored in



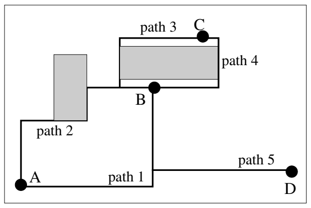
(a) Four pins to connect.



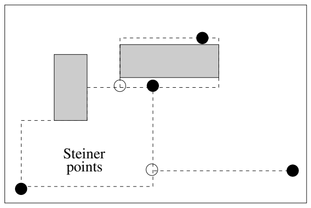
(b) Start from pin A. Connect to pin B and form two paths.



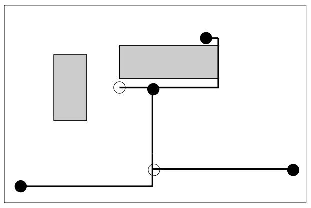
(c) Connect to pin C. Form two more paths.



(d) Connected to D. Form one more path.



(e) Steiners points and path lengths are found.


(f) A route is constructed by finding a MST.

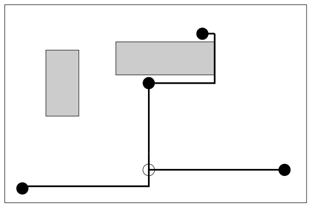

(g) After garbage collection step.

Fig. 1. An example illustrating our approach

H and $best$ is the shortest distance found so far. This is to ensure that *all* the paths of the same shortest Manhattan distance connecting to the next pin p_j will be taken out from the heap and processed. Note that nothing is done if the distance stored on the heap top is larger than the distance recorded in $dist[x][y]$ (line 12), since in this case, we had already found a shorter path reaching (x, y) in some previous round and no updates are needed. An important step in our approach to improve the efficiency is that in each round of the outer while-loop (line 6), some new paths are found and the $dist[\,][\,]$ value of all the points on these paths will be set to zero for the next around, and instead of starting the maze routing once again from scratch, we can continue with the propagation from the last round after inserting some new elements into H. The heap data structure allows us to update the distances of the affected points rapidly without repeating works which have already been done . Imagine a point whose distance value is independent of the newly found paths, its information in H and $dist[\,][\,]$ is kept and will be useful, while for those points that can get a shorter distance because of the newly found paths, their information in $dist[\,][\,]$ will be updated automatically after inserting the 2-tuples $((x, y), 0)$ into H (line 8) for all the points (x, y) lying on those newly found paths. A simple example is shown in Fig. 2 to illustrate this propagation process.

Algorithm 2 Propagate

```
1: //Propagate to reach all the pins $P = \{p_1, p_2, \ldots, p_m\}$ in a $m$-pin net
2: $S = \{(x_{p_{\pi(1)}}, y_{p_{\pi(1)}})\}$ //$p_{\pi(1)}$ is the starting point and $(x_{p_i}, y_{p_i})$ is the coordinate of pin $p_i$.
3: Initialize an empty heap $H$
4: Initialize all elements in $dist[\,][\,]$ as $\infty$ //$dist[\,][\,]$ is a 2-dimensional array with the same dimensions as the simplified Hanan grid
5: $V = \{p_{\pi(1)}\}$
6: while some pins are not connected yet do
7:    Set $dist[x][y] = 0$ for all $(x, y) \in S$
8:    Insert into $H$ 2-tuples $((x, y), 0)$ for all $(x, y) \in S$ //Note that $H$ is sorted according to the second element in the 2-tuples
9:    Initialize $best = \infty$
10:   while $(min(H) < best)$ do
11:      $((x, y), d) = delete\_min(H)$
12:      if $(d \leq dist[x][y])$ //the distance is correct then
13:         if $(d \neq 0)$ and $(x, y)$ is the position of some pin $p_j$ in $P$ //it is a pin point but not connected yet then
14:            $dist[x][y] = d$, $best = d$
15:         end if
16:         Let $(x_e, y_e)$, $(x_s, y_s)$, $(x_w, y_w)$ and $(x_n, y_n)$ be the east, south, west and north neighbor of $(x, y)$
17:         for each neighbor $(x_c, y_c)$ of $(x, y)$ where $c \in \{e, s, w, n\}$ do
18:            if $(d + d_c < dist[x_c][y_c])$ and $(x_c, y_c)$ is not blocked, where $d_c$ is the shortest Manhattan distance from $(x, y)$ to $(x_c, y_c)$ //a shorter distance is found then
19:               $dist[x_c][y_c] = d + d_c$
20:               Insert $((x_c, y_c), dist[x_c][y_c])$ into $H$
21:               if $(x_c, y_c)$ is the position of a pin $p_j$ in $P$ then
22:                  if $(best > dist[x_{p_j}][y_{p_j}])$ then
23:                     $best = dist[x_{p_j}][y_{p_j}]$ //update the current shortest distance
24:                  end if
25:               end if
26:            end if//$(d + d_c < dist[x_c][y_c])$ and ...
27:         end for//each neighbor of $(x, y)$
28:      end if//$(d \leq dist[x][y])$
29:   end while//$(min(H) < best)$
30:   Backtrack from $p_j$ to find a set $Q$ of paths with the same shortest Manhattan distance connecting to $p_j$
31:   $S = \{(x, y) | (x, y)$ on some paths in $Q\}$
32:   $V = V + \{p_j\}$
33: end while//Some pins are not connected yet
```

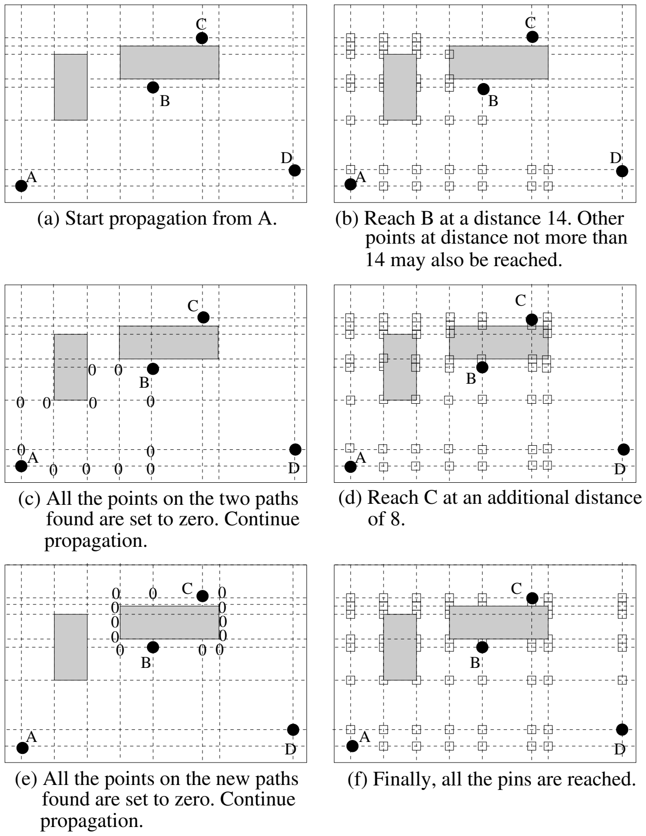
C. Backtrack

After a next pin p_j is found by propagation, backtracking will be performed to find all the paths connecting to p_j with the same shortest Manhattan distance. Basically, our backtracking strategy will try to move without turning as much as possible, unless meeting an obstacle, or seeing an obstacle on one side of the path. At the end, we will find multiple paths of the same length, and the paths found will be "moving around" the obstacles. An example is shown in Fig. 3. The pseudocode of this backtracking step is shown in Algorithm 3.

In the pseudocode of *Backtrack*, a stack K is used to keep all the possible branching points on the paths (points at which a path can change in direction to form another feasible path). The elements

Algorithm 3 Backtrack

```
1: //Backtrack from pin $p_j$ to find a set $Q$ of new paths of the same shortest Manhattan distance connecting $p_j$ to some points which are already connected
2: Initialize a stack called $path$ with $p_j$ //has only 1 point at the beginning
3: Initialize a stack $K$
4: Let $(x_e, y_e)$, $(x_s, y_s)$, $(x_w, y_w)$ and $(x_n, y_n)$ be the east, south, west and north neighbor of $(x_{p_j}, y_{p_j})$
5: for each neighbor $(x_c, y_c)$ where $c \in \{e, s, w, n\}$ do
6:     if $dist[x_c][y_c] + d_c = dist[x_{p_j}][y_{p_j}]$ and $(x_c, y_c)$ is not blocked, where $d_c$ is the shortest Manhattan distance from $(x_{p_j}, y_{p_j})$ to $(x_c, y_c)$ //$(x_{p_j}, y_{p_j})$ is propagated from $(x_c, y_c)$ then
7:         Push into $K$ a 3-tuple $((x_c, y_c), dist[x_c][y_c], (x_{p_j}, y_{p_j}))$ where the third element is called the *father* of the first element in the 3-tuple
8:     end if
9: end for
10: while $K$ is not empty do
11:     Pop from $K$ a 3-tuple $((x, y), d, (x_f, y_f))$
12:     Pop from $path$ until $top(path) = (x_f, y_f)$ //delete the tail part of the previously found path and we are going to construct a new path from the branching point $(x_f, y_f)$
13:     Push $(x, y)$ onto $path$ //the next point on the new path
14:     while $dist[x][y] \neq 0$ //a connected point not found yet do
15:         Let $(x_e, y_e)$, $(x_s, y_s)$, $(x_w, y_w)$ and $(x_n, y_n)$ be the east, south, west and north neighbor of $(x, y)$
16:         for each neighbor $(x_c, y_c)$ of $(x, y)$ where $c \in \{e, s, w, n\}$ do
17:             if $dist[x_c][y_c] + d_c = dist[x][y]$ and $(x_c, y_c)$ is not blocked, where $d_c$ is the shortest Manhattan distance from $(x, y)$ to $(x_c, y_c)$ //$(x, y)$ is propagated from $(x_c, y_c)$ then
18:                 Label this neighbor by 1
19:             else
20:                 Label this neighbor by 0
21:             end if
22:         end for//each neighbor of $(x, y)$
23:         Let $r \in \{e, s, w, n\}$ be the direction from $(x_f, y_f)$ to $(x, y)$ //$r$ is the direction the path is going
24:         if $(x_r, y_r)$ is labeled 1 //the original direction not blocked then
25:             for each neighbor $(x_c, y_c)$ of $(x, y)$ with a label 1, where $c \in \{e, s, w, n\}$ do
26:                 if all neighbors of $(x_c, y_c)$ are not blocked then
27:                     change the label of $(x_c, y_c)$ to 0 //minimize turning unless there are blockages
28:                 end if
29:             end for
30:         end if//$(x_r, y_r)$ is labeled 1
31:         for each neighbor $(x_c, y_c)$ of $(x, y)$ with a label 1, where $c \in \{e, s, w, n\}$ do
32:             Push a 3-tuple $((x_c, y_c), dist[x_c][y_c], (x, y))$ onto $K$ //they are the branching points
33:         end for
34:         Pop from $K$ a 3-tuple $((x, y), d, (x_f, y_f))$ //pick one of the neighbors to continue with the backtrack first, others will be considered when they are popped out from $K$ some time later
35:         Push $(x, y)$ onto $path$
36:     end while//$dist[x][y] \neq 0$
37:     Copy $path$ to a path set $all\_path$
38: end while//$K$ is not empty
```



(a) Start propagation from A.

(b) Reach B at a distance 14. Other points at distance not more than 14 may also be reached.

(c) All the points on the two paths found are set to zero. Continue propagation.

(d) Reach C at an additional distance of 8.

(e) All the points on the new paths found are set to zero. Continue propagation.

(f) Finally, all the pins are reached.

Fig. 2. An example illustrating our propagation step

stored in K are 3-tuples $((x, y), d, (x_f, y_f))$ meaning that this step is going from (x_f, y_f) to (x, y) and the shortest distance of (x, y) from a connected point is d, i.e., $dist[x][y] = d$. From each branching point, a feasible path will be constructed step by step until reaching a point which is already connected, i.e., $dist[\ ][\ ] = 0$ (line 14 to 36). A variable $path$ is used to remember the current path under construction. On line 12, it is popped until getting to (x_f, y_f) which is a branching point, and we are going to construct another feasible path starting from that point. In the while-loop that computes a feasible path (line 14 - 36), a variable $label$ is used to remember whether a point is a branching point and to be stored in K. We will try to move in the same direction without bending if no blockages are seen (line 26 to 28). If there are blockages, we will find all possible paths moving around the blockages. After a new path is found, it will be stored in the path set all_path. At the end, all feasible paths of the same shortest Manhattan distance will be stored in all_path.

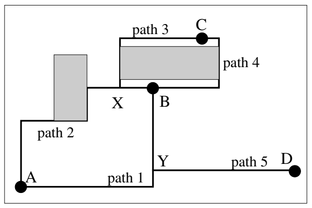
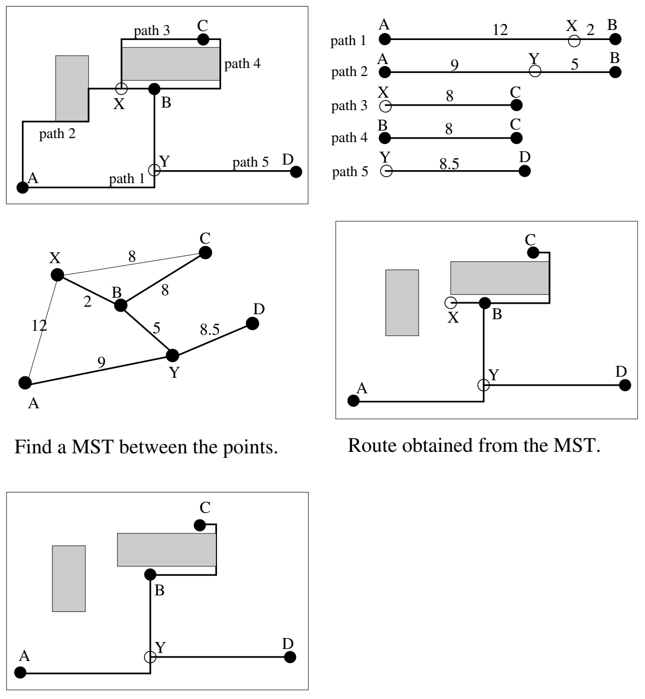


Fig. 3. A number of paths moving around the obstacles will be found.

D. *Finding MST*

After finding a set of all paths, we can look at the ending points of each path. Since they are either a pin or a Steiner point, we can identify the positions of all the Steiner points immediately after this step. Then, we need to find the path lengths between the set of pins and Steiner points if they are connected. These distances can be computed very efficiently by, again, tracing each path once. When we are tracing a path p, we can identify the positions of the Steiner points or the pin points, and we can tell the distances between them easily. A Steiner tree T will be constructed based on these distances and paths to connect all the points. It may happen that in T, some paths are just connected to some Steiner points only. We can remove these edges by removing recursively those leaf nodes which are Steiner points. After this, the tree give a reasonable route connecting all the pins and avoiding the obstacles. An example of this MST construction step is shown in Fig. 4.



Find a MST between the points.

Route obtained from the MST.

Remove dangling connection.

Fig. 4. An example illustrating how we find the Steiner points and path lengths

E. *Local Refinement Scheme*

Based on the results generated by the multi-pin maze routing engine, we will perform a post-processing step to further reduce the wire length. In this post-processing step, we will first identify all the pin points, Steiner points and turning points from the paths we computed. The original path information can be easily transformed to these point to point neighboring information by tracing each path once. After transformation from the path information to the point to point information, we will perform our local refinement scheme. In this local refinement scheme, for each point, we will first check whether it forms one of the topologies shown in Fig. 5. If such topologies exist, we will try to form a cycle by adding one straight edge (dotted line) between the two parallel paths. A longest path on this cycle formed will then be deleted to reduce the total wire length. We will consider in details all the possible cases of having pins or Steiner points lying on this cycle. This local refinement scheme is more general than the U-shape pattern refinement in [15]

IV. Experimental Results

We implemented our algorithm in the C programming language and all the experiments were performed in a computer with a 2.2GHz Intel Pentium processor and 4GB memory running in the Linux environment. There are 22 benchmark circuits, five industrial test cases (ind1-ind5) from Synopsys, twelve test cases used in [12] (rc1-rc12), and five randomly generated test cases from [15]. We compared our algorithm with the approach in [15], which gives the most updated results for this problem. The results of [15] shown below are directly cited from [15] and their experiments were performed on a 2GHz AMD-64 machine with 8GB memory under the Ubuntu 6.06 operation system.

Table I lists the total wire length of the Steiner tree constructed by the two algorithms. We can see from the eighth column that without the post processing step we can improve over [15] by 1.73% on average. After applying the post processing step, we can improve over [15] by 2.01% on average. Since the solutions obtained might already be very close to the optimal, it is more meaningful to compare the differences from the optimal. The fourth column of Table I shows the edge length of the rectilinear Steiner minimal tree of all the pins without considering the blockages, which is a lower bound to the optimal solution. We can see that the improvement over the differences from this lower bound of the optimal solution is 27.04%, and we can imagine that the improvement over the differences from the optimal solution will be even larger. Table II compares the CPU time of the two approaches. From this table, we can see that our method is similar to [15] in terms of running time and is sufficiently efficient. For the test case of rc12, there are many blockages and the pins' positions are evenly distributed in the 2-dimensional space, so the running time is longer than that of the other test cases.

Test Cases	CPU Time (second)		Test Cases	CPU Time (second)	
	[15]	Ours		[15]	Ours
ind1	<0.01	<0.01	rc7	0.43	0.65
ind2	<0.01	0.04	rc8	0.83	1.18
ind3	<0.01	<0.01	rc9	0.91	1.49
ind4	<0.01	<0.01	rc10	0.62	0.26
ind5	<0.01	<0.01	rc11	3.15	0.76
rc1	<0.01	0.05	rc12	118.52	147.14
rc2	<0.01	0.10	rt1	0.06	0.16
rc3	<0.01	0.09	rt2	0.11	0.55
rc4	<0.01	0.11	rt3	0.47	0.20
rc5	0.01	0.08	rt4	0.95	0.33
rc6	0.24	0.56	rt5	2.06	2.78

TABLE II
Comparisons on the CPU time.

V. Conclusion

We have proposed a new heuristic method to construct obstacle-avoiding rectilinear Steiner minimum trees (OARSMT). Experimental results show that our approach is effective and efficient and it out-performs the best previous approach on this problem.

References

[1] K.L. Clarkson and S. Kapoor and P.M. Vaidya, "Rectilinear Shortest Paths through Polygonal Obstacles in $O(nlog^2n)$", *Proceedings ACM Symposium on Computational Geometry*, pp.251-257, 1987.

Test Cases	m	k	GeoSteiner (A)	Total Edge-Length			Improvement(%)		
				[15] (B)	Ours (C)	Ours (D)	$(B-C)/B$	$(B-D)/B$	$(B-D)/(B-A)$
ind1	10	32	604	632	619	619	2.06	2.06	46.43
ind2	10	43	9100	9600	9500	9500	1.04	1.04	20.00
ind3	10	50	587	613	600	600	2.12	2.12	50.00
ind4	25	79	1078	1121	1100	1096	1.87	2.23	58.14
ind5	33	71	1295	1364	1367	1360	-0.22	0.29	5.80
rc1	10	10	25290	26900	25980	25980	3.42	3.42	57.14
rc2	30	10	39170	42210	42280	42010	-0.17	0.47	6.58
rc3	50	10	51900	55750	54560	54390	2.13	2.44	35.32
rc4	70	10	54910	60350	59740	59740	1.01	1.01	11.21
rc5	100	10	71260	76330	75040	74650	1.69	2.20	33.14
rc6	100	500	76356	83365	81768	81607	1.92	2.11	25.08
rc7	200	500	105003	113260	111592	111542	1.47	1.52	20.81
rc8	200	800	107416	118747	116168	115931	2.17	2.37	24.85
rc9	200	1000	105698	116168	113642	113460	2.17	2.33	25.86
rc10	500	100	161790	170690	168520	167620	1.27	1.80	34.49
rc11	1000	100	-	236615	236359	235283	0.11	0.56	-
rc12	1000	10000	-	789097	765541	761606	2.99	3.48	-
rt1	10	500	1817	2267	2231	2231	1.59	1.59	8.00
rt2	50	500	44214	48441	47730	47297	1.47	2.36	27.06
rt3	100	500	7579	8368	8222	8187	1.74	2.16	22.94
rt4	100	1000	7634	10306	9917	9914	3.77	3.80	14.67
rt5	200	2000	42608	53993	52664	52473	2.46	2.82	13.35
Average		-	-	-	-	-	1.73	2.01	27.04

A is the edge length of the rectilinear Steiner minimal tree of all the pins constructed by GeoSteiner, m is the number of pins and k is the number of blockages. C is the edge length before applying the local refinement scheme and D is the edge length after applying the local refinement scheme.

TABLE I
COMPARISON ON THE TOTAL EDGE-LENGTH.

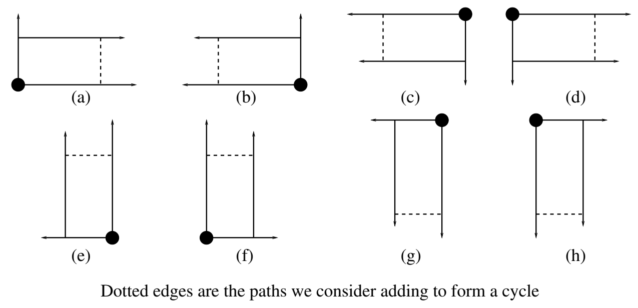

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
Dotted edges are the paths we consider adding to form a cycle

Fig. 5. Local Refinement Scheme

[2] Y.F. Wu and P. Widmayer, M.D.F. Schlag and C.K. Wong, "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles", *IEEE Transaction on Computer-Aided Design*, Vol.36, No.3, pp.321-331, 1987.

[3] J. Ganley and J.P. Cohoon, "Routing a Multi-terminal Critical Net: Steiner Tree Construction in the Presence of Obstacles", *Proceedings International Symposium on Circuits and Systems*, pp.113-116, 1994.

[4] S.Q. Zheng and J.S. Lim and S.S. Iyengar, "Finding Obstacle-avoiding Shortest Paths using Implicit Connection Graphs", *IEEE Transaction on Computer-Aided Design*, Vol.15, No.1, pp.103-110, 1996.

[5] Y. Yang and Q. Zhu and T. Jing and X. Hong and Y. Wang, "Rectilinear Steiner Minimal Tree Among Obstacles", *Proceedings IEEE ASICCON*, pp.348-351, 2003.

[6] H. Wu and Z. Feng and T. Jing and X. Hong and Y. Yang and G. Yu and X. Hu and G. Yan, "FORst: a 3-step Heuristic for Obstacle-avoiding Rectilinear Steiner Minimal Tree Construction", *Journal of Information and Computational Science*, pp.107-116, 2004.

[7] T. Lengauer, "Combinatorial Algorithms for Integrated Circuit Layout", *Wiley*, England, 1990.

[8] H. Wu and T. Jing and X. Hong and Z. Feng and X. Hu and G. Yan, "An-OARSMan: Obstacle-avoiding Routing Tree Construction with Good Length Performance", *Proceedings ASP-DAC*, pp.7-12, 2005.

[9] Z. Shen and C. Chu and Y. Li, "Efficient Rectilinear Steiner Tree Construction with Rectilinear Blockages", *Proceedings ICCD*, pp.38-44, 2005.

[10] H. Zhou, "Efficient Steiner Tree Construction Based on Spanning Graphs", *IEEE Transactions on Computer-Aided Design*, Vol.23, No.5, pp.704-710, 2004.

[11] Y. Shi and T. Jing and L. He and Z. Feng and X. Hong, "CDCTree: Novel Obstacle-avoiding Routing Tree Construction based on Current Driven Circuit Model", *Proceedings ASP-DAC*, 2006.

[12] Z. Feng and Y. Hu and T. Jing and X. Hong and X. Hu and G. Yan, "An $O(nlogn)$ Algorithm for Obstacle-avoiding Routing Tree Construction in the λ-geometry Plane", *Proceedings ISPD*, pp.48-55, 2006.

[13] P.C. Wu and J.R. Gao and T.C. Wang, "A Fast and Stable Algorithm for Obstacle-avoiding Rectilinear Steiner Minimal Tree Construction", *Proceedings ASP-DAC*, pp.262-267, 2007.

[14] R. Hentschke and J. Narasimham and M. Johann and R. Reis, "Maze Routing Steiner Trees with Effective Critical Sink Optimization", *Proceedings ISPD*, 2007.

[15] C.W. Lin and S.Y. Chen and C.F. Li and Y.W. Chang and C.L. Yang, "Efficient Obstacle-avoiding Rectilinear Steiner Tree Construction", *Proceedings ISPD*, 2007.