A Fast and Stable Algorithm for Obstacle-Avoiding Rectilinear Steiner Minimal Tree Construction *

Pei-Ci Wu, Jhih-Rong Gao, and Ting-Chi Wang

Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
g944310@oz.nthu.edu.tw, g944309@oz.nthu.edu.tw, tcwang@cs.nthu.edu.tw

ABSTRACT - In routing, finding a rectilinear Steiner minimal tree (RSMT) is a fundamental problem. Today's design often contains rectilinear obstacles, like macro cells, IP blocks, and pre-routed nets. Therefore obstacle-avoiding RSMT (OARSMT) construction becomes a very practical problem. In this paper we present a fast and stable algorithm for this problem. We use a partitioning based method and an ant colony optimization based method to construct obstacle-avoiding Steiner minimal tree (OASMT). Besides, two heuristics are proposed to do the rectilinearization and refinement to further improve wirelegnth. The experimental results show our algorithm achieves the best wirelength results in most of the test cases and the runtime is very small even for the larger cases each of which has both the number of terminals and the number of obstacles more than 100.

I. INTRODUCTION

Routing plays an important role in the physical design stage of very/ultra large scale integrated circuits (VLSI/ULSI). Routing a net, i.e., finding a rectilinear Steiner minimal tree (RSMT) is a fundamental problem. In fact, today's design often contains rectilinear obstacles, like macro cells, IP blocks, and pre-routed nets. Therefore by taking obstacles into consideration, obstacle-avoiding RSMT (OARSMT) construction becomes a very practical problem.

It has been proved that the RSMT problem is NP-complete [1] and taking obstacles into account even increases the complexity of the problem. Thus there exists no efficient optimal algorithm for finding an OARSMT.

There are many recent works focusing on the OARSMT problem. FORst [2] is a 3-step heuristic which can tackle large-scale problems efficiently. An-OARSMan [3], using ant colony optimization [10] on the track graph [7], can achieve shorter wirelength than FORst when the number of terminals is less than 100. CDCTree [4], based on a current driven circuit model, can achieve shorter wirelength than An-OARSMan when the number of terminals is less than 50. Zion *et al*. [5] propose an effective approach which constructs an OARSMT from a connection graph called spanning graph. Although [3-5] can find an OARSMT with good length performance, they tend to have a long runtime on finding a good solution when the number of terminals is large.

In the experiment of [5], there is an industrial test case where the total number of nets is 47730 has 0.3% nets whose number of terminals is more than 100. Besides, in detailed routing or ECO routing, it is possible that the number of obstacles in a routing region is more than 100 or even more. Therefore in practice, there exist cases which have more than 100 terminals and 100 obstacles. Thus how to find an OARSMT efficiently for a large case having many terminals and obstacles becomes a practical and useful problem. An $O(n\log n)$ algorithm is proposed in [6] where n is the sum of the number of terminals and the number of obstacles. Because of its lower time complexity, [6] has very small runtime in large cases which have many terminals and obstacles. Since there are no other recent works which are able to deal with such large cases, [6] cannot tell whether its performance is good enough. From our experimental results (to be shown in section V), we see that for large cases reported in [6], they still have much room to improve wirelength.

To the best of our knowledge, all existing heuristics for OARSMT construction cannot achieve both small wirelength and small runtime for large cases. Therefore an algorithm which can get the best wirelength and efficiency is needed. In this paper, we present a fast and stable algorithm for constructing OARSMTs. Given a complete graph formed by terminals, our algorithm first applies a partitioning method, which is to find a minimum spanning tree (MST) on the graph, and remove the segments intersecting obstacles. Thus the MST will be partitioned into sub-trees. Our algorithm then uses an ant colony optimization based approach, which is different from the one proposed in An-OARSMan [3], to connect the sub-trees into a single tree and get an obstacle-avoiding Steiner minimal tree (OASMT). Notice that An-OARSMan [3], a connection graph based approach, is applied on track graphs, and we find that if the numbers of terminals and obstacles are large, the size of the track graph becomes huge and significantly increases the runtime. Instead our ant colony optimization based method works on spanning graphs. As indicated in [5], a spanning graph is an efficient and effective connection graph. Although for large cases a spanning graph may lose some information, we show that it can produce an OASMT with good wirelength performance from our experiments. After obtaining an OASMT, our algorithm continues to use a rectilinearization heuristic to generate an OARSMT, and apply a refinement method on it for further improvement and get the final OARSMT.

The experimental results indicate that our algorithm obtains

* This work was partially supported by National Science Council under Grant No. NSC-95-2220-E-007-037, and Ministry of Economic Affairs under Grant No. MOEA-95-EC-17-A-01-S1-031.

1-4244-0630-7/07/$20.00 ©2007 IEEE.

the best wirelength results for most of the test cases when compared with the recent works [2-6]. The runtime is just a little worse than [6], and much better than the others [2-5]. For the largest test case which has 1000 terminals and 10000 obstacles, our algorithm only spends 4.2 seconds and reduces 54.37% wirelength as compared with [6], an O($n$log$n$) algorithm whose runtime is 2.8 seconds. As a result, our algorithm is very fast and stable.

We organize the rest of the paper as follows. In section II, we formally define the problem. In section III, the overview of our algorithm is introduced. Section IV describes the details of our algorithm. Section V presents the experimental results and we conclude the paper in Section VI.

II. PROBLEM FORMULATION

Given a set of terminals, $T = \{t_1, t_2, \ldots, t_n\}$, and a set of rectangular obstacles, $O = \{o_1, o_2, \ldots, o_m\}$, the obstacle-avoiding rectilinear Steiner minimal tree (OARSMT) problem is to find a rectilinear Steiner minimum tree (RSMT) which connects all terminals together but does not intersect any obstacle, and the wirelength of the RSMT should be as small as possible. Note that if all boundaries of a non-rectangular obstacle are either horizontal or vertical, we can dissect it into a set of rectangular obstacles. For simplicity, we assume all obstacles in our problem to be rectangular in the rest of the paper.

III. OVERVIEW OF OUR ALGORITHM

Our algorithm contains four steps, which are overviewed as follows:

Step 1: We construct a complete weighted graph formed by all pairs of terminals, and then we construct a minimum spanning tree (MST) connecting all the terminals (see Fig. 1 where grey rectangles represent obstacles, and black points represent terminals). The weight between two terminals is the Manhattan distance between them plus an obstacle penalty which will be defined in section IV-A. After constructing an MST, there might be some edges whose L-shaped segments (including upper and lower L-shaped segments) intersect obstacles. For example, in Fig. 1 there is no L-shaped segment that can connect terminals v_1 and v_2 without intersecting any obstacle. Therefore we remove those intersecting segments from the MST so that the MST can be partitioned into a set of sub-trees, SST (see Fig. 2).

Step 2: A spanning graph of all terminals and obstacles is generated using the idea proposed in [5] (see Fig. 3). This graph will be used in *Step 3*.

Step 3: We merge the sub-trees of SST obtained in *Step 1* using an ant-colony optimization based algorithm modified from [3] to find the connections between sub-trees and transform SST into an obstacle-avoiding Steiner minimal tree (OASMT). In order to reduce the runtime, the ant colony optimization based algorithm is running on a spanning graph rather than a track graph, which means that a path connecting two sub-trees must be a set of edges on the spanning graph (see Fig. 4 where the bold lines represent the path connecting two sub-trees on the spanning graph after using the ant colony optimization based algorithm).

Step 4: After finding the OASMT connecting all terminals, a rectilinearization procedure is applied on the tree to generate an initial OARSMT, called IOARSMT (see Fig. 5). Then we use a refinement method to further improve the wirelength of IOARSMT by removing redundant segments which will be defined in section IV-D. Finally we can get the final OARSMT, called FOARSMT (see Fig. 6).

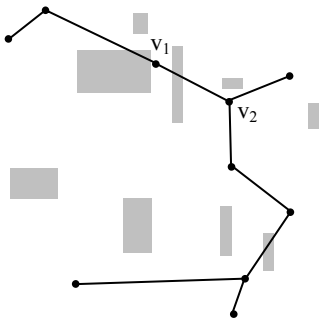

Fig. 1. MST.

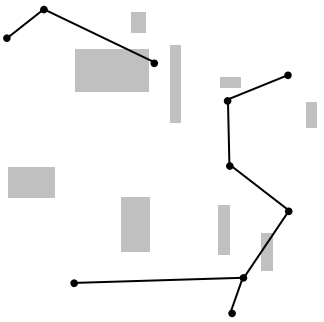
Fig. 2. Partition MST into sub-trees.

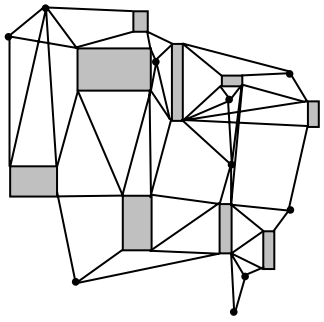
Fig. 3. Spanning graph.

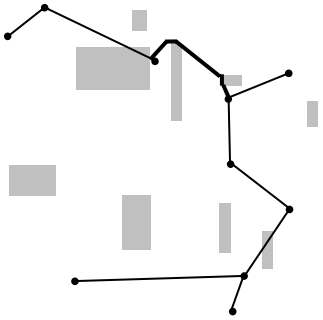
Fig. 4. Merge.

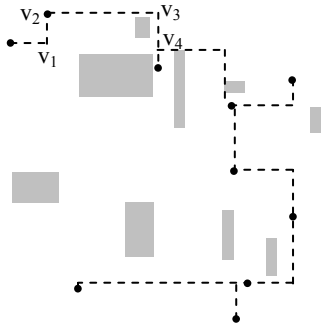

Fig. 5. IOARSMT after rectilinearization.

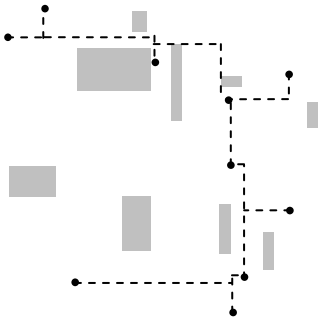
Fig. 6. FOARSMT after refinement.

IV. DETAIL OF OUR ALGORITHM

A. Partitioning Terminals into a Set of Sub-trees

The goal of the partitioning method is to obtain a set of sub-trees, SST. Our algorithm only needs to merge them into a Steiner tree in a later step, and the SST will be the partial

part of the Steiner tree. Therefore doing partitioning first can save a lot of runtime but it may cause worse wirelength because of losing some solution space.

Our partitioning method is based on an MST. We first construct a complete weighted graph $G_c = (V_c, E_c)$, where V_c is the set T of terminals, and E_c is the set of edges connecting all pairs of terminals. In E_c, the weight of an edge e connecting v_i and v_j is defined below:

weight(e) = Manhattan distance(e) + obstacle penalty(e)

Manhattan distance(e) = $| x_i - x_j | + | y_i - y_j |$ where v_i has coordinate (x_i, y_i) and v_j has coordinate (x_j, y_j).

Obstacle penalty (OP) is the estimation of additional distance if both L-shaped segments of an edge intersect obstacles. An edge connecting two terminals has two L-shaped segments (including upper and lower L-shaped segments). If an L-shaped segment intersects obstacles, OP is the sum of the lengths of the intersecting boundaries of the obstacles, otherwise it is zero. But we do not check all obstacles to find intersections with respect to an edge because it is too time consuming. We just check the nearest obstacles from a terminal in the four directions (top, down, left, right). Because an edge may have two OPs corresponding to two L-shaped segments of the edge, we choose the smaller of these two as the OP of the edge. We take Fig. 7 as an example to describe the definition of OP in detail. In Fig. 7(a), the upper L-shaped segment of edge e intersects two obstacles, obstacle o_1 from vertex v_1 in the top direction and obstacle o_2 from vertex v_2 in the left direction. The OP of the upper L-shaped segment is w_1+h_2. Similarly, the OP of the lower L-shaped segment is h_3. Then the OP of the edge e is h_3 which is the smaller one. In Fig. 7(b), because the upper L-shaped segment of edge e does not intersect any obstacle, the OP of edge e is 0.

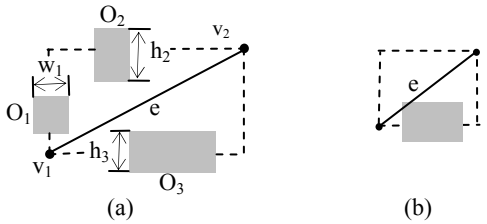

Fig. 7. Obstacle penalty.

We use the Kruskal method [8] to find an MST on the complete graph G_c, and then traverse each tree edge on the MST to check whether both L-shaped segments of an edge intersect obstacles. If both L-shaped segments of an edge intersect obstacles, we remove the edge from the MST. After traversing all segments of the MST, the MST can be partitioned into a set of sub-trees, SST. These sub-trees are the partial part of the MST, and we observe that the MST is a good initial solution for an OASMT. So the SST helps to keep good wirelength performance of the final OARSMT.

Constructing the complete weighted graph G_c needs $O(n^2)$ time where n is the number of terminals. The time complexity of the Kruskal method is $O(m\log m)$ where m is the number of edges in the complete graph G_c. Because the number of edges, m, in the complete graph is $O(n^2)$, the time complexity of the Kruskal method is $O(n^2 \log n)$. And checking if an edge in the MST intersects obstacles can be done in linear time. Therefore the partitioning step can be done in $O(n^2 log n)$ time.

B. Constructing a Spanning Graph

In [3], the ant colony optimization based algorithm runs on a track graph [7]. The size of a track graph is $O(r^2)$, where r is the number of the rectilinear boundaries of all obstacles. Therefore it takes a long time to get a solution. In order to reduce the runtime when running the ant colony optimization based algorithm, we choose spanning graph as our connection graph.

In a spanning graph, every vertex (including terminals and four corners of each rectangular obstacle) is connected to the nearest vertices in its four directions, upper-right, upper-left, lower-right, and lower-left. Fig. 3 is an example of a spanning graph. Considering all given vertices, the connections will form a spanning graph of cardinality $O(n)$ where n is the total number of vertices. In other words, spanning graph is able to describe the relative geometrical relationship between vertices in the plane using $O(n)$ edges. Moreover, by using an $O(n \log n)$ sweep line algorithm, the construction of a spanning graph can be done in $O(n \log n)$ time.

C. Merge: Ant-Colony Optimization Based Algorithm

Das *et al* [10] proposed an ant-colony approach for constructing a Steiner tree in a given graph. Having made some improvements and changes on the basic approach, An-OARSMan [3] constructs an OARSMT on the track graph. We further modify the ant-colony optimization based algorithm used in An-OARSMan and then apply it on the spanning graph to merge the sub-trees in SST into an OASMT. In this section, we will briefly explain the ant-colony approach and focus on the difference between [3] and ours.

The basic ant-colony approach consists of multiple iterations. In each iteration, an ant is to be placed on each terminal which needs to be connected, and then one ant is randomly chosen each time to execute a movement, leaving behind a pheromone trail for others to follow. The ant selects an edge to move at a time, and the selection is determined by some user-defined rules. In our implementation, all edges must be segments in the spanning graph, and we always let an ant move to a location according to a function which is related to the trail intensity, the distance between the ant and other ants, and the distance between the ant and the vertices other ants have ever passed. Once a movement is executed, the chosen edge becomes a branch of the sub-tree formed by the corresponding ant. Each ant maintains its own *tabu-list*, which records the vertices already visited to avoid revisiting them. When ant A meets ant B, ant B dies, and the vertices in the B's *tabu-list* are added into A's. By the movement, the separated sub-trees will be connected together to form an OASMT. One iteration is ended when there is only one ant left, and the OASMT connecting all terminals is also obtained. In our implementation, we set the number of iterations to be 50. From the experiment, it shows 50 is enough to obtain

good solutions and the number of iterations more than 50 just gets minor improvement whereas causes a long runtime.

The major difference between [3] and ours is that in An-OARSMan the ant is placed on each terminal, but in our algorithm we put an ant for each sub-tree because there already exists SST in the spanning graph. Given an ant m_i for sub-tree t_i, the set of vertices on t_i is called Vt_i. When we apply the ant-colony optimization based algorithm, we suppose Vt_i is already visited by its ant m_i. However, if we put all vertices in Vt_i in the *tabu-list* of m_i, the ant-colony approach would spend too much time on traversing the *tabu-list*. Here we have to deal with two problems before applying the ant-colony optimization algorithm. The first problem is to find the initial location of the ant for each sub-tree, and the second one is to determine which vertices should be included in the *tabu-list* of each ant. We propose a greedy method to solve these two problems. In the partitioning step (Step 1), we remove the tree edges of the MST which intersect obstacles so that we can partition the MST into a set of sub-trees. Our greedy method is to take these removed edges to determine the locations and *tabu-lists* of ants. Given a removed edge e_{rem} and its two end points, v_i and v_j with $v_i \in Vt_i$ and $v_j \in Vt_j$, ant m_i adds v_i to its *tabu-list* and equally ant m_j adds v_j to its *tabu-list*. Each ant maintains a variable called *min-weight* which records the minimum weight among the removed edges each of which has at least one end point belonging to the ant's sub-tree. When an end point of a removed edge has smaller weight than its corresponding ant's *min-weight*, we update the ant's *min-weight* and set the location of the ant to the end point. This means that assuming the removed edge e_{rem} has weight d, if d is smaller than ant m_i's *min-weight*, we update m_i's *min-weight* to d and set m_i's location to end point v_i. And ant m_j can do the same action if d is smaller than m_j's *min-weight*. Fig. 8 is an example for the greedy method. Fig. 8(a) is an MST after Step 1, and there exist three removed edges. Fig. 8(b) shows the locations of ants and their *tabu-lists* determined by applying the greedy method. Taking *ant1* for a detailed explanation, end point v_1 of the removed edge e_1 and end point v_2 of the removed edge e_2 both belong to the sub-tree of *ant1*. Thus the *tabu-list* of *ant1* has v_1 and v_2. For determining the location of *ant1*, we compare the weights of e_1 and e_2. Because the weight of e_2 is smaller, we set the location of *ant1* to v_2. Similarly we do the same processes to *ant2*, *ant3* and *ant4*. The *tabu-list* of *ant2* has v_3 and its location is v_3. The *tabu-list* of *ant3* has v_6 and v_7, and its location is v_7. The *tabu-list* of *ant4* has v_8, and its location is v_8.

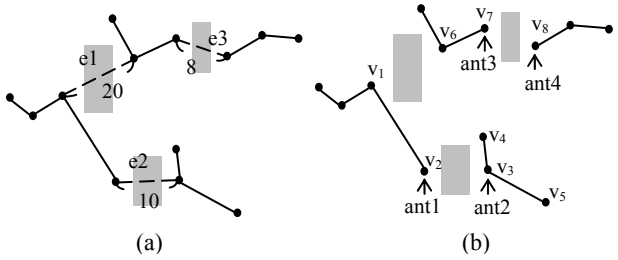

Fig. 8. (a) An MST and dashed lines represent removed edges. (b) The locations of ants and their tabu-lists determined by the greedy method.

Our greedy method can be done as follows. For each edge removed from the MST, we insert the end points of the edge into the *tabu-lists* of the corresponding ants and update the locations of the corresponding ants if needed. Thus the time complexity of the greedy method is linear to the number of the removed edges.

After determining the initial location and *tabu-list* of each ant, we apply the ant colony optimization based algorithm to find an OASMT. In the basic ant colony approach when an ant visits a vertex, the ant adds the vertex to its *tabu-list*, but our algorithm has a constraint for this. Take Fig. 8(b) as an example. Although *ant2* visits v_4 via a path in the spanning graph, the constraint does not allow *ant2* to add v_4 to its *tabu-list* because v_4 already belongs to the sub-tree of *ant2*. The size of the *tabu-list* of each ant would be smaller with the constraint than without the constraint. Therefore it can save runtime on the ant colony optimization based algorithm. From our experimental results, it shows good wirelength performance though the constraint may lose some solution space.

D. Rectilinearization and Refinement

Our goal here is to find an OARSMT. After applying the ant colony optimization based algorithm to find an OASMT, a rectilinearization process is needed to modify all tree edges into either horizontal or vertical segments. We propose a greedy method to rectilinearize these edges. The idea is to share as many segments as possible when generating the rectilinear segments so that the total wirelength will be smaller.

First of all, we choose a 1-degree terminal v_s as the start point and use breadth first search (BFS) to traverse the whole OASMT. Once a non-rectilinear tree edge is traversed, we use either a vertical-horizontal or horizontal-vertical L-shaped segment to replace this edge. For two vertices v_a and v_b, a vertical-horizontal L-shaped segment connecting from v_a to v_b is composed of a vertical sub-segment followed by a horizontal sub-segment, and a horizontal-vertical L-shaped segment is composed of a horizontal sub-segment followed by a vertical sub-segment. If we consider all possible combinations of L-shaped segments of all tree edges, it will be too time consuming. Therefore we try to rectilinearize each non-rectilinear edge using a greedy manner to determine which kind of L-shaped segment we prefer when replacing each edge. In the beginning of BFS, the first traversed edge (v_s, v_i) arbitrarily chooses a vertical-horizontal or horizontal-vertical L-shaped segment to do the rectilinearization when edge (v_s, v_i) is neither a horizontal or vertical edge. For each neighboring vertex v_j of v_i, if the L-shaped segment connecting from v_s to v_i is a horizontal-vertical one, we try to rectilinearize (v_i, v_j) using a vertical-horizontal L-shaped segment. Otherwise a horizontal-vertical L-shaped segment is used. However if the chosen L-shaped segment causes an illegal L-shaped segment which intersects obstacles, then we try the other L-shaped segment (notice that we can easily prove that the other L-shaped segment will not intersect any obstacle).

We use an example to describe the details of the

rectilinearization. In Fig. 9(a), an OASMT is going to be rectilinearized. Assuming v_s is the start point to do BFS, we arbitrarily choose an L-shaped segment to reconnect (v_s, v_1) as shown in Fig. 9(b). Then (v_1, v_2) and (v_1, v_3) are to be considered. Because the L-shaped segment connecting from v_s to v_1 is a horizontal-vertical one, we prefer to use a vertical-horizontal L-shaped segment rather than a horizontal-vertical one to connect v_1 to v_2 or v_3. This is because we want to share segments with the existing segments. It is shown in Fig. 9(c) that (v_1, v_2) is reconnected with a vertical-horizontal L-shaped segment which shares a segment with the existing one. However, in the case where the chosen L-shaped segment we prefer intersects any obstacle, we choose the other one. In Fig. 9(c), the vertical-horizontal L-shaped segment connecting from v_1 to v_3 which is marked by 'x' intersects the obstacle, so we have to choose the other one. Fig. 9(d) shows the final result. Because a refinement step will follow after rectilinearization, we call the current OARSMT as an initial RSMT (IOARSMT).

Now we already have a legal OARSMT solution. Because our rectilinearization method is a greedy one, some "redundant segment" may exist. As a result, we apply a refinement procedure on the IOARSMT to further improve the wirelength. Our idea is to eliminate the 'U' shape connections in the IOARSMT. Fig. 6 is an example after applying refinement on the IOARSMT in Fig. 5. Notice that vertices v_1, v_2, v_3 and v_4 form a U-shaped connection in Fig. 5. Therefore we can move the horizontal segment (v_2, v_3) to the position v_1, and then remove the redundant non-terminal leaf v_3 and the segment connecting to it. By removing all U-shaped connections in the IOARSMT, we can get our final resultant FOARSMT in Fig. 6.

Our rectilinearization method only needs to traverse the OASMT once. Similarly, the refinement method only needs to traverse the IOARSMT once. Therefore the total time complexity in the rectilinearization and refinement step is linear to the sum of the sizes of the OASMT and IOARSMT.

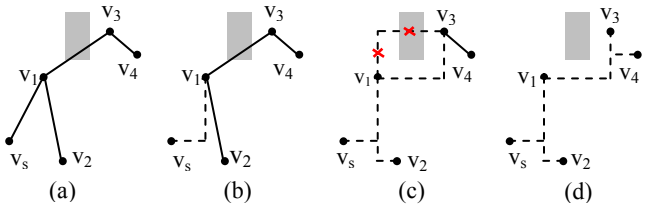

Fig. 9. The process of the rectilinearization. Solid lines represent edges in OASMT, and dashed lines represent segments of IOARSMT.

V. EXPERIMENTAL RESULTS

We have implemented our algorithm in C language and performed it on a Sun Blade2000 workstation with 1200MHz CPU and 8GB memory. We collect the test cases used in [6] and three additional industry test cases, and then compare our results with several state-of-the-art works.

Table 1 and Table 2 show the comparison results of wirelength and runtime with An-OARSMan [3], CDCTree [4], 2-OASMT [6] and the spanning graph based algorithm [5]. In both tables, the first two columns give the numbers of terminals and obstacles in our test cases. The first three cases are from the industry, and the other are the cases used in [6].

The results of [3] and [6] are quoted from the experimental results reported in [6]. And the results of [4] are provided by an author of CDCTree. Because we cannot get the test cases and the program of [5], we implement their algorithm by ourselves and run it in the same platform mentioned above. But we do not get the result for the last test case (i.e., the largest one with 1000 terminals and 10000 obstacles.) because it executes more than 9 hours without termination[1]. [3] and [6] are performed in a Sun V880 fire workstation with 755MHz CPU and 4GB memory, and [4] is performed on a Unix workstation with 2.66G CPU and 1G memory.

From Table 1 we can see that our algorithm achieves the best wirelength results in most of the test cases. The wirelength improvement ratios of our algorithm over [3], [4] and [5] are 5.4%, 3.38% and 1.87% on average, respectively. Besides, when the size of the problem becomes lager, the runtimes of [3], [4], and [5] have great increases. Although [3] and [4] do not run at the same platform as ours, their trends in runtime increase are obviously greater than ours. The long runtime of [5] is because it needs to construct a complete graph among terminals when calculating the shortest paths between all pairs of terminals. As a result, the bottleneck of its runtime is finding the shortest paths between all pairs of terminals.

[6] is the only algorithm which is able to deal with large cases efficiently. However, as the size of a case increases, the wirelength performance of it gets worse. Compared with [6], we can achieve 22.18% wirelength improvement on average and 54.37% improvement at most with a little more runtime. Although [6] can produce the routing result very efficiently, there is still much room to improve its wirelength performance. On the contrary, our algorithm can keep both efficiency and good wirelength performance no matter the size of a case is large or small, which makes our algorithm very practical.

Fig. 10 shows the routing solution obtained from our algorithm for the test case with 500 terminals and 100 obstacles.

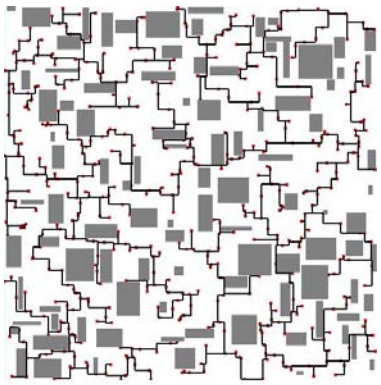

Fig. 10. A routing solution.

[1] [5] cannot get the result for the largest test case because for easy implementation we use a less efficient data structure to implement the Dijkstra algorithm for finding shortest paths. But we believe that [5] still runs slower than our algorithm even it is implemented in a more efficient way.

Term#	obs#	Wirelength								
		[3]	[4]	[6]	[5]	Ours	Improvement ratio over [3] (%)	Improvement ratio over [4] (%)	Improvement ratio over [6] (%)	Improvement ratio over [5] (%)
10	32	-	-	-	644	626	-	-	-	2.80
74	625	-	-	-	1731	1640	-	-	-	5.26
115	1024	-	-	-	3011	2872	-	-	-	4.62
10	10	27840	26970	30410	29320	27250	2.12	-1.04	10.39	7.06
30	10	43090	41700	45640	43400	43220	-0.30	-3.65	5.30	0.41
50	10	63250	62380	58570	57020	56500	10.67	9.43	3.53	0.91
70	10	66310	66560	63340	61910	61090	7.87	8.22	3.55	1.32
100	10	82320	80010	83150	78240	76870	6.62	3.92	7.55	1.75
100	500	-	-	149750	86770	84327	-	-	43.69	2.82
200	500	-	-	181470	118169	115461	-	-	36.37	2.29
200	800	-	-	202741	123360	122574	-	-	39.54	0.64
200	1000	-	-	214850	120567	120017	-	-	44.14	0.46
500	100	-	-	198010	174420	172490	-	-	12.89	1.11
1000	100	-	-	250570	242840	238377	-	-	4.87	1.84
1000	10000	-	-	1723990	N/A	786731	-	-	54.37	-
average							5.40	3.38	22.18	1.87

Table 1. Comparison on wirelength. '-' represents that the result is not available.

Term#	obs#	Run time (s)				
		[3]	[4]	[6]	[5]	Ours
10	32	-	-	-	<0.01	<0.01
74	625	-	-	-	14.17	0.1
115	1024	-	-	-	60.69	0.21
10	10	0.164	0.485	0.002	<0.01	<0.01
30	10	1.075	1.034	0.003	<0.01	<0.01
50	10	3.504	8.79	0.004	0.01	<0.01
70	10	10.552	67.62	0.004	0.01	<0.01
100	10	26.974	595.1	0.004	0.02	<0.01
100	500	-	-	0.057	12.49	0.31
200	500	-	-	0.062	28.15	0.36
200	800	-	-	0.095	72.66	1.53
200	1000	-	-	0.129	112.29	1.8
500	100	-	-	0.026	4.14	0.27
1000	100	-	-	0.037	35.34	0.81
1000	10000	-	-	2.823	-	4.2

Table 2. Comparison on runtime.

VI. CONCLUSIONS

In this paper, we present a fast and stable approach for obstacle-avoiding rectilinear Steiner minimal tree construction. We use a partitioning based method to partition an MST into some sub-trees. Then we apply an ant-colony optimization based algorithm on the spanning graph to connect the sub-trees into an obstacle-avoiding Steiner minimal tree. Finally we rectilinearize the OASMT and use a refinement method to further improve the wirelength performance. Compared with state-of-the-art works, our approach has the best wirelength performance in most of the cases and the runtime is very small even for large cases. The high efficiency and good solution quality of our approach makes it extremely practical in the routing process.

VII. REFERENCES

[1] M. R. Garey and D. S. Johnson, "The Rectilinear Steiner Tree Problem is NP-complete," SIAM Journal on Applied Mathematics, 32, 1977.

[2] Y. Hu, Z. Feng, T. Jing, X. Hong, Y. Yang, G. Yu, X. Hu, and G. Yan, "A 3-Step Heuristic for Obstacle-Avoiding Rectilinear Steiner Minimum Tree Construction," in Proc. of ISC&I, 2004.

[3] Y. Hu, T. Jing, X. Hong, Z. Feng, X. Hu, and G. Yan, "An-OARSMan: Obstacle-Avoiding Routing Tree Construction with Good Length Performance," in Proc. of ASP-DAC, 2005.

[4] Y. Shi, T. Jing, L. He, Z. Feng, and X. Hong, "CDCTree: Novel Obstacle-Avoiding Routing Tree Construction based on Current Driven Circuit Model," in Proc. of ASP-DAC, 2006.

[5] Z. Shen, C. C.N. Chu, and Y. Li, "Efficient Rectilinear Steiner Tree Construction with Rectilinear Blockages," in Proc. of ICCD, 2005.

[6] Z. Feng, Y. Hu, T. Jing, X. Hong, X. Hu, and G. Yan, "An O(nlogn) Algorithm for Obstacle-Avoiding Routing Tree Construction in the λ-Geometry Plane," in Proc. of ISPD, 2006.

[7] Y. F. Wu, P. Widmayer, M. D. F. Schlag, and C. K. Wong, "Rectilinear Shortest Paths and Minimum Spanning Trees in the Presence of Rectilinear Obstacles," IEEE Trans on Computers, 1987.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms 2nd Edition, The MIT Press, 2001.

[9] H. Zhou, N. Shenoy, and W. Nicholls, "Efficient Spanning Tree Construction Without Delaney Triangulation," Information Processing Letter, 2002.

[10] S. Das, S. V. Gosavi, W. H. Hsu, and S. A. Vaze, "An Ant Colony Approach for the Steiner Tree Problem," Proc. of GECCO, 2002.