Journal of Information & Computational Science 1: 3 (2004) 107–116
Available at http://www.joics.com

FORst: A 3-Step Heuristic For Obstacle-avoiding Rectilinear Steiner Minimal Tree Construction ⋆

Yu Hu [a,*], Zhe Feng [a], Tong Jing [a], Xianlong Hong [a], Yang Yang [a], Ge Yu [b], Xiaodong Hu [c], Guiying Yan [c]

[a] *Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, P. R. China*

[b] *School of Information Sci. and Eng., Northeastern University, Shenyang, 110004, P. R. China*

[c] *Institute of Applied Mathematics, Chinese Academy of Sciences, Beijing, 100080, P. R. China*

Abstract

Macro cells, IP blocks, and pre-routed nets are often regarded as obstacles in VLSI routing phase. Obstacle-avoiding rectilinear Steiner minimum tree (OARSMT) algorithms are often used to meet the needs of practical routing applications. However, OARSMT algorithms with multi-terminal nets routing still can not satisfy the requirements of practical applications. This paper presents a 3-step heuristic, named FORst, to tackle the OARSMT problem. In Step1, we partition all terminals into some subsets in the presence of obstacles. Then in Step2, we connect terminals in each connected graph with one or more trees, respectively. In Step3, we connect the forest consisting of trees constructed in Step2 into a completed Steiner tree spanning all terminals while avoiding all obstacles. Two algorithms, called ACO-RSMT and GFST-RSMT, are proposed to construct OARSMT in a connected graph in Step2, which are suitable for different situations. This algorithm has been implemented and tested on cases with typical obstacles. The experimental results show that FORst is with great efficiency and can get good performance. Moreover, it can tackle large scale nets among complex obstacles, such as a net with 1000 terminals in the presence of 100 rectangular obstacles.

Keywords: Rectilinear Steiner Minimal Tree (RSMT); Obstacle-Avoiding; IC CAD; VLSI; Routing

---

1 Introduction

Routing a net, finding a rectilinear Steiner minimum tree (RSMT) for a given terminal set, is one of the fundamental problems in integrated circuit computer-aided design (IC CAD). However, only a few RSMT algorithms take obstacles into consideration. Macro cells, IP blocks, and pre-routed nets are often regarded as obstacles and RSMT construction among obstacles is often

⋆This work was supported in part by the NSFC under Grant No.60373012, the SRFDP of China under Grant No.20020003008, and the Hi-Tech Research and Development (863) Program of China under Grant No.2004AA1Z1050.

*Corresponding author
Email address: matrix98@mails.tsinghua.edu.cn (Yu Hu).

1548–7741/ Copyright © 2004 Binary Information Press
December 2004

used as the estimation for wire length and delay throughout the process of routing. So, we need efficient obstacle-avoiding RSMT (OARSMT) algorithms in IC CAD. [1] proved that the RSMT problem is NP-complete. So, OARSMT problem is more complicated and no polynomial-time algorithm can solve it exactly.

Since it has been paid less attention to multi-terminal net routing among obstacles due to great complexity and difficulty, the VLSI designers always use the multi-terminal variant of maze routing algorithms [2, 3, 4, 5]. Ganley *et al* [6] proposed an algorithm to compute the optimal 3-terminal or 4-terminal RSMT in the presence of obstacles. Then, G3S, B3S, and G4S heuristics [6] are proposed for larger scale cases (but less than 20 terminals). The running time is still long. Zachariasen *et al* [7] proposed an efficient exact algorithm to find an obstacle-avoiding Euclidean Steiner tree with less than 150 terminals. But the complexity of this algorithm is exponential and no rectilinear exact algorithms are reported at present. Zhou *et al* introduced a generalized connection graph [8] and proposed an efficient algorithm to compute OARSMT for a 3-terminal net [9]. The recent progress in OARSMT is an $O(mn)$ 2-step heuristic presented in [10], which works well when the number of terminals is less than 7 and the obstacles are convex polygons. We find that the existing algorithms tackling OARSMT problem are more suitable for small scale and simply cases. That is, it contains only a few terminals and obstacles. The shape of the obstacles is always constrained to be convex polygons.

The main contribution of this paper is the 3-step heuristic, called FORst, for obstacle-avoiding rectilinear Steiner minimum tree construction. FORst can handle complex cases such as many terminals and concave polygon obstacles, keep the high performance, and take short running time. Two algorithms, called ACO-RSMT and GFST-RSMT, are proposed to construct OARSMT in a connected graph in Step2, which are suitable for different situations. The FST (fulsome Steiner tree) construction, ant colony optimization (ACO) technology, and detour method are used in our algorithm, which makes it be more practical for many cases.

The rest of this paper is organized as follows. In Section 2, we give the outline of our FORst algorithm. In Section 3, the algorithm is described in detail. Section 4 shows the experimental results and Section 5 concludes the whole paper.

2 Outline of FORst Algorithm

The OARSMT problem is described as follows.

Definition 1 ***(OARSMT)*** *Given a set T of n points, called terminals, and a set O of m rectilinear obstacles in the plane, find a set S of additional points, called Steiner points, such that the length of a rectilinear minimum spanning tree of $T \cup S$, which avoids all obstacles in O, is minimized.*

Our FORst algorithm is a 3-step heuristic. In Step1, we partition all the terminals into some subsets in the presence of obstacles by some rules. After Step1, connected components, regarded as hyper-graphs in Section 3, are generated. Then in Step2, we connect terminals in each connected graph with one or more trees, respectively. As a result, the separated sub-trees are constructed after this step. In Step3, we connect the forest consisting of the trees constructed in Step2 into a completed Steiner tree spanning all terminals while avoiding all obstacles. Two kinds of algorithms, called ACO-RSMT and GFST-RSMT, are proposed to construct OARSMT in a

connected graph in Step2, which are suitable for different situations. The flow chart and an illustration of FORst are shown in Fig.1 and Fig.2, respectively.

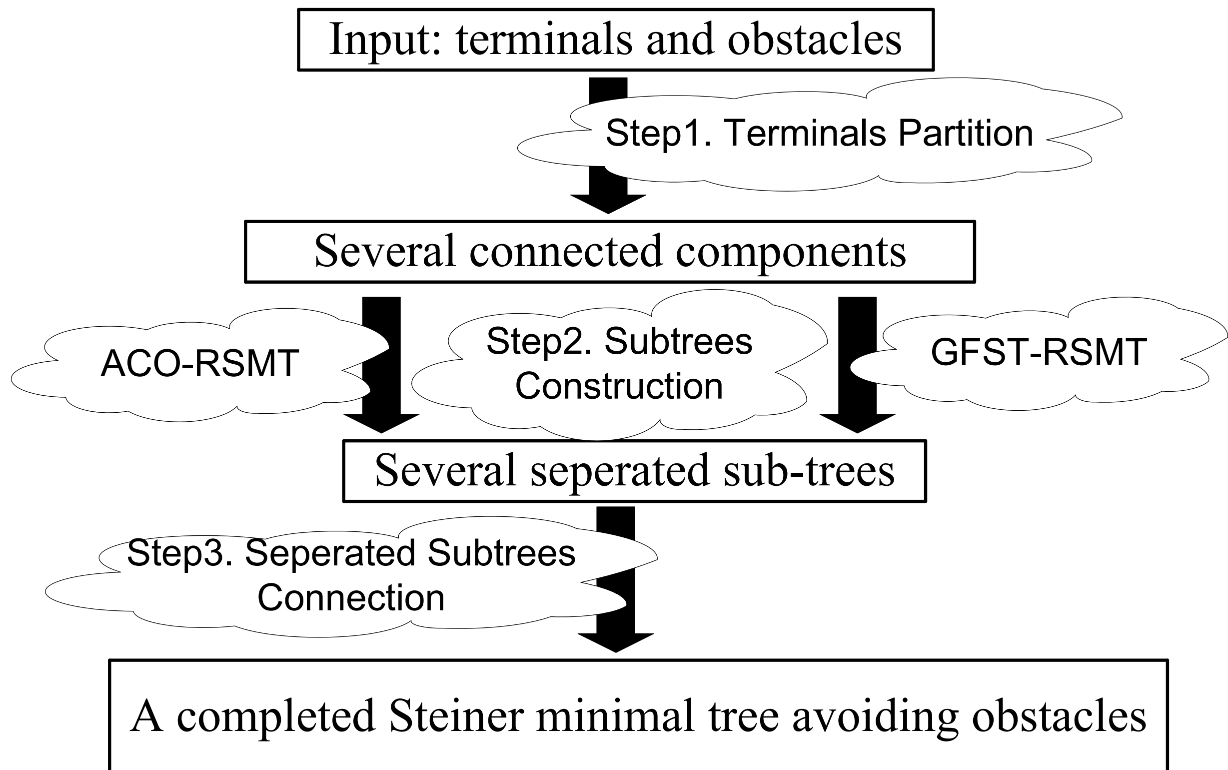

Fig. 1: The flow chart of FORst algorithm

3 Details of the 3-step

3.1 Step1: Terminals partition

Definition 2 ***(FST)*** *A rectilinear Steiner tree is called a full Steiner tree (FST) if every terminal is a leaf of the tree.*

Definition 3 ***(compatible and incompatible)*** *FST f is compatible with SMT s, if f and s share one and only one terminal, and they can appear simultaneously in any SMT. f and s are incompatible, if f and s share more than one terminals or share partial edges.*

We try to translate OARSMT into RSMT to simplify the problem in this step. We partition the terminals into some subsets and construct a graph for each subset. Here, we construct the FSTs by Hwang's theorems [1] and the rules of Warme [11] in $O(n^2)$ time [16]. Then we delete those FSTs that intersect with obstacles. After these operations, there are some FSTs survived, which is called survival set denoted by F. Consider the hypergraph $H = (T, F)$ with the set of terminals T as its vertices and the survival set as its hyperedges. Obviously, the hypergraph is partitioned into some connected components, such as $H_1, H_2, \cdots$, and H_n, where every pair of vertices are connected by hyperedges in H_i. After the partition, the set T and set F is partitioned

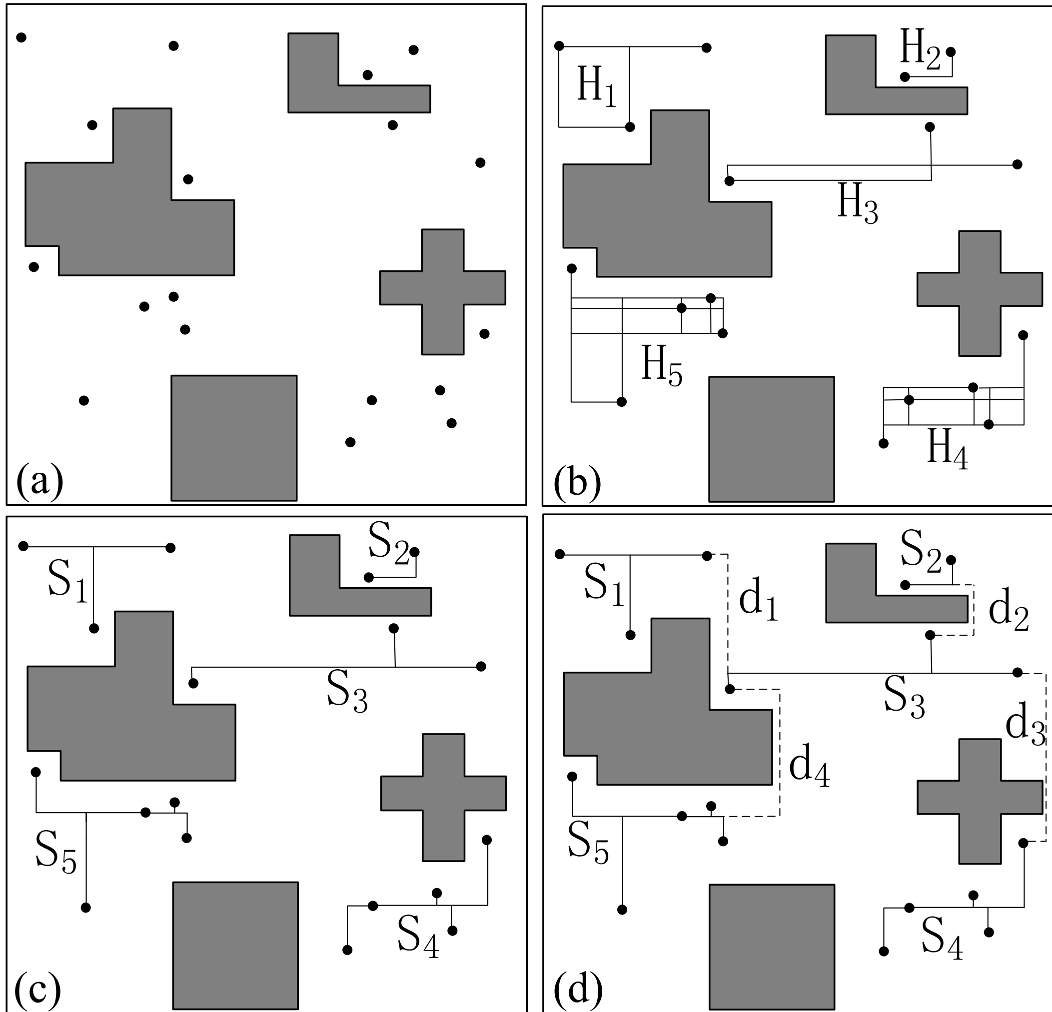

Fig. 2: An illustration of FORst algorithm, (a) the input of the problem: some terminals and four obstacles, (b) the result after *Step*1 (terminal partition), (c) the result after *Step*2 (sub-tree constructions), (d) the result after Step3 (separate sub-tree connection)

into some corresponding subsets, such as $T_1, \cdots, T_n$, and $F_1, \cdots, F_n$, where T_i is the vertex set of H_i and F_i is the edge-set of H_i. The set of hyperedges in H_i, i.e. F_i, forms a connection graph C_i, in which the set of all terminals and Steiner points in all FSTs forms the set of vertices and the line segments of the edges. We can see this process shown in Fig.2(b), in which the hypergraph H is partitioned into five connected components, $H_1, H_2, \cdots$, and H_5, and the sets T and F are partitioned into five subsets correspondingly.

3.2 Step2: Subtrees construction

In this step, the main task is to connect the terminals in each connected components H_i with one completed Steiner tree or several separated Steiner trees. In Fig.2(c), RSMTs are constructed in each of the connected components, which are $S_1, S_2, \cdots$, and S_5, respectively.

We now propose two methods to construct an OARSMT in a given connected component, named ACO-RSMT and GFST-RSMT, respectively. Each has its own advantages, which enables

us to archive better performances in different situations.

3.2.1 ACO-RSMT

The ACO-RSMT is an algorithm based on ant colony optimization [13]. It finds a near-optimal RSMT in the connection graph C_i, which spans all terminals in T_i. Because the connection graph C_i does not intersect with any obstacles, the RSMT is obstacle-avoiding.

In the connection graph, we place an ant in each terminal that needs to be connected. An ant will determine a new vertex by some rules and move to that vertex via an edge in the connection graph. Each ant maintains its own tabu-list, which records the vertices already visited to avoid revisiting it. When ant A meets ant B, ant A dies, and add the vertices in the A's tabu-list into B's. After every movement, an ant will leave some trail in the edge just passed, and the trail will evaporate in a constant rate. The process of ant moving repeats until all terminals in the graph are connected to make a Steiner tree. We record the current best solution, i.e. the minimum wire-length tree. Then the whole procedure iterates until the wire-length of the Steiner tree can not be decreased any more or the times of iterations is more than $MAXLOOP$, where $MAXLOOP$ is a constant.

An ant determines its next vertex that it wants to move stochastically, but biased on a higher value $p_{i,j}$, which is a trade-off between the desirability and the trail intensity. Given an ant m in vertex i, and the desirability of vertex j (j must be the neighbor of i in the connection graph) is defined as

$$\eta_j^m = \frac{1}{c(i,j) + \gamma \cdot \psi_j^m} \tag{1}$$

where γ is a constant, and ψ_j^m is the shortest distance from vertex j to all the vertices in the tabu-list of other ants, which makes the current ant join into others as quickly as possible.

We borrow the definition of $p_{i,j}$ and the trail updating strategy proposed from [14]. The pseudo-code of ACO-RSMT algorithm is shown in Algorithm 1.

3.2.2 GFST-RSMT

GFST(Greedy FST)-RSMT algorithm selects compatible FSTs in F_i greedily, and constructs several separated SMTs, such as s_{i1}, s_{i2}, $\cdots$, and s_{in}, such that

1. $T(s_{ij}) \cap T(s_{ik}) = \varnothing$, for each pair of j and k ,
2. $T(s_{ij}) \cup T(s_{ik}) \cup \cdots \cup T(s_{in}) T_i$,

where $T(s)$ is the set of terminals spanned by SMT s. Each FST is initialized with the measure κ, and the measure κ of a FST t is defined as follows.

$$\kappa(t) = \frac{(wire\ length\ of\ t)^\lambda}{(terminal\ number\ of\ t)^\omega} \tag{2}$$

where λ and ω are constants. Obviously, a FST with smaller κ indicates that it covers more terminals with a shorter wire-length.

Algorithm 1 ACO-RSMT
Input: Terminal set T_i in H_i
Output: A rectilinear Steiner minimal tree in H_i spanning T_i

1: Initiate the connection graph C_i in partition H_i;
2: Set the intensity in each edge in C_i to be p_0;
3: $LoopNum \leftarrow 0$;
4: $tree \leftarrow \varnothing$;
5: **while** $loopNum < MAXLOOP$ **do**
6: Place an ant on each vertex in the terminal set T_i and put the vertex into its tabu-list;
7: **while** $antNumber > 1$ **do**
8: Select an ant m in vertex i randomly;
9: Compute the p_j of m by equation (1) and trail intensity;
10: Ant m moves to vertex j under the probability of p_j;
11: Add vertex j into m's tabu-list;
12: $tree \leftarrow tree + edge(i, j)$;
13: **if** m meets m' **then**
14: Add vertices in tabu-list of m into that of m';
15: m died;
16: $antNumber \leftarrow antNumber - 1$;
17: **end if**
18: **end while**
19: Update the trail intensity in every edge by the trail updating strategy;
20: $loopNum + +$;
21: **end while**
22: **return** $tree$;

We maintain a triple (D, Q, Y), where Y are the SMTs in the partial solution, and D are candidate FSTs which are compatible with Y, and Q are the terminals that do not covered by Y. In the initialization, Y is set to be empty, and D is set to be F_i, and Q is set to be T.

At the beginning of GFST-RSMT algorithm, we select the FST f with minimal κ from D, and add a new SMT t made up of f into Y. Whenever a FST is added into partial Y, the FSTs incompatible with Y should be removed from D and the terminals covered by this FST should be removed from Q. If there are any FSTs compatible with t in D, the one whose κ is minimal should be selected and added into t. Otherwise, no FSTs are compatible with t. If D is not empty, select the FST f' with minimal κ, and add a new SMT made up of f' into Y. If D is empty, every terminal left in Q makes up of a degenerated SMT, and adds them into Y.

Parameter setting in equation 2 plays an important role. The experimental results under many kinds of testing cases show that the value of ω should be given a higher priority. We set $\lambda = 1$ and $\omega = 4$ in all cases in our program, by which we can archive the best performance.

The pseudo-code of GFST-RSMT algorithm is shown in Algorithm 2.

3.2.3 Comparison between ACO-RSMT and GFST-RSMT

The number of edges and vertices in the connection graph C_i constructed by F_i algorithm is $O(n)$ [11], where n is the number of terminals in T_i. So the times of moving ants for constructing a tree

Algorithm 2 GFST-RSMT
Input: Terminal set T_i in H_i
Output: One or more separated rectilinear Steiner minimal trees in H_i spanning T_i

```
1: Initiate FST set F_i in partition H_i;
2: D ← F_i, Q ← T_i, Y ← ∅;
3: Select f that has the minimum κ from D;
4: Y ← t ← f;
5: Remove those FST incompatible with Y from D;
6: Remove those terminals covered by f from Q;
7: while D ≠ ∅ do
8:    while there are any FSTs compatible with t in D do
9:       Select f that is compatible with t and has the minimum κ from D;
10:      t ← t + f;
11:      Remove those FST incompatible with f from D;
12:      Remove those terminals covered by f from Q;
13:   end while
14:   Select f that has the minimum κ from D;
15:   t ← f;
16:   Y ← Y + t;
17:   Remove those FST incompatible with Y from D;
18:   Remove those terminals covered by f from Q;
19: end while
20: for each terminals n left in Q do
21:   Y ← n;
22: end for
23: return Y;
```

is $O(n)$. In each movement, an ant will decide the next vertex in $O(n)$. So the time-complexity of ACO-RSMT is $O(n^2)$. For GFST-RSMT algorithm, we compute the κ for all FSTs in F_i, and sort these FSTs by κ, which requires $O(n\log(n))$ computing time. Then, we select $O(n)$ FSTs one by one greedily. So GFST-RSMT runs in $O(n\log(n))$ time.

On the other hand, ACO-RSMT can always get a complete RSMT spanning all terminals in a given connected component H_i. But GFST-RSMT can only provide several separated RSMTs, which cover all terminals in H_i.

As the trade-off between run-time and performance of solution, we use these two algorithms in different situations. For the terminal number in H_i is larger (more than 20), we use GFST-RSMT, otherwise, we use ACO-RSMT. For the large scale problem (over 100 terminals), the maximum number of H_i whose terminals number is smaller than 20 is $n/20$, and time-complexity for each of these H_i can be view as a constant at this time. So the actual time-complexity of ACO-RSMT is $O(n)$, which suggests the domain complexity of Step2 is $O(n\log(n))$.

3.3 Step3: Separated subtrees connection

After the process of Step1 and Step2, some separate sub-trees are constructed. In Step3, we need to connect these sub-trees into a completed tree avoiding all obstacles. We compute the exact

shortest path of each pair of nodes in each pair of trees, and then choose the minimum ones to connect all the Steiner trees. The dot lines in Fig.2(d) show the results of Step3 for the given instance, and the shortest path we found to connect the trees in Fig.2(c) are d_1, d_2, d_3, and d_4, respectively.

Zheng *et al* [15] introduce the conception of *detour* value, and proved that the shortest path avoiding obstacle is the Manhattan distance plus twice of the detour value.

We compute the shortest path between the source node s and the destination node d in presence of obstacles using *obstacle-avoiding shortest path* (OASP) algorithm. The OASP algorithm is based on the idea of the *detour* technique proposed in [15].

Zheng [15] proved that the time-complexity to compute the shortest path between a node pair with detour algorithm is $O(e \cdot \log(e))$. We must compute every node-pair of all different sub-trees. There are $O(n^2)$ node-pairs totally. So the time-complexity of OASP algorithm is $O(n^2 \cdot e \cdot \log(e))$.

4 Experimental Results

The FORst algorithm has been implemented in C++ language and performed on a Sun V880 fire workstation with Unix operating system. We randomly generate some terminals among several kinds of rectilinear polygon obstacles to test the proposed algorithm. The obstacles are constructed to include all types of convex and concave rectilinear polygons, which includes rectangle, L-shaped polygon, cross-shaped polygon, and more complicated shapes.

We set $\gamma = 1$ and $MAXLOOP = 100$ in ACO-RSMT algorithm and set $\lambda = 1$ and $\omega = 4$ in GFST-RSMT in all experiments, by which the average best performance is archived.

Table 1 shows the performance results (percent improvements over the minimum spanning tree) for our FORst for randomly generated instances containing 10 rectangular obstacles and the indicated numbers of terminals. Since no result is reported in [6] for cases with over 20 terminals, so corresponding entries are marked by "-" in Table 1.

In Table 1, G4S denotes the result of G4S [6]. ACO/CPU and GFST/CPU show the results and CPU-time while only using ACO-RSMT and GFST-RSMT in Step2, respectively. FORst/CPU shows the results and CPU-time by using ACO-RSMT and GFST-RSMT together by the rule given in Sub-Section3.2.3.

Table 1: Observation results for LSE

Term#	G4S	Ours(%)/CPU(s)					
		ACO	CPU	GFST	CPU	FORst	CPU
5	9.2	7.8	0.02	4.9	< 0.01	7.8	0.02
10	9.1	7.1	0.10	5.2	< 0.01	7.1	0.10
15	9.4	7.4	0.25	3.8	< 0.01	7.4	0.25
20	9.0	7.2	0.38	4.2	< 0.01	7.2	0.38
50	-	7.3	3.22	7.4	0.26	7.1	2.75
100	-	8.9	31.2	7.8	3.12	8.6	3.66
500	-	9.3	1244	8.2	482	9.1	559
1000	-	9.4	1343	8.5	1112	8.9	1240

From Table 1, we find that when the number of terminals is less than 20, most of the FST

are deleted in Step1, which generates many separate terminals and makes the solution quality of GFST-RSMT be worse than ACO-RSMT. When the problem scale becomes larger, more FSTs survive, which makes the solution quality of GFST-RSMT be better. The cooperation of ACO-RSMT and GFST-RSMT is a trade-off between CPU-time and the solution quality. Then the average quality of the algorithm is better than the others.

Fig.3 shows the result of FORst algorithm to route 1000 terminals in the presence of 100 rectangular obstacles.

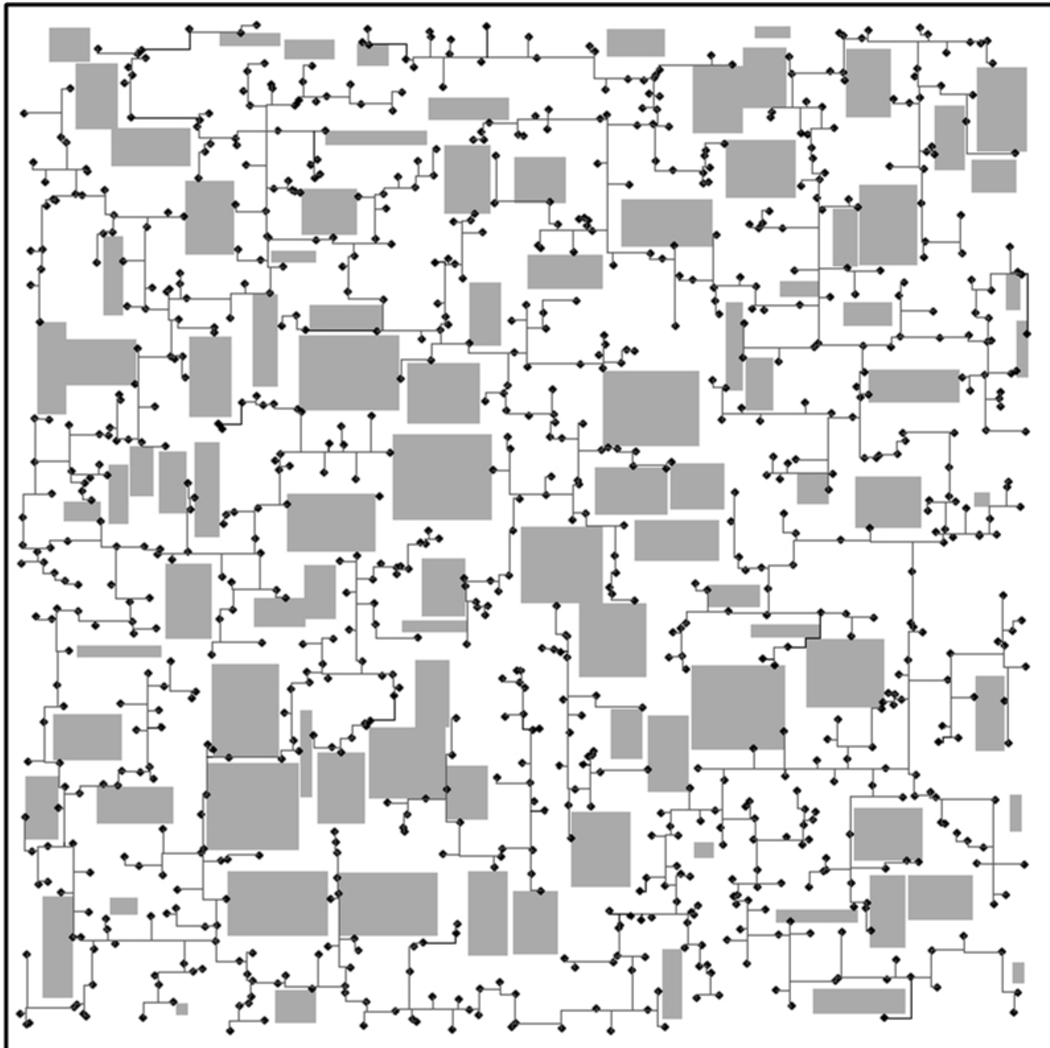

Fig. 3: RSMT for 1000 terminals in the presence of 100 rectangular obstacles

5 Conclusion

In this paper, we propose a 3-step heuristic, called FORst, for OARSMT construction. The experimental results show that FORst can handle large scale cases with a short running time. Two kinds of routing algorithms in Step2 make the FORst work well on the cases with different scales. Furthermore, the performance of FORst algorithm keeps stable in very large scale cases. The time-complexity of the three steps are $O(n^3)$, $O(n \cdot \log(n))$, and $O(n^2 \cdot e \cdot \log(e))$ respectively,

which suggests that the total time-complexity of FORst algorithm is $\max(O(n^3), O(n^2 \cdot e \cdot \log(e)))$, where n is the number of terminals and e is the number of edges of obstacles.

References

[1] M. R. Garey and D. S. Johnson, The rectilinear Steiner tree problem is NP-complete, SIAM Journal on Applied Mathematics, 1977, 32: pp.826-834.

[2] S. B. Akers, A Modification of Lee's Path Connection Algorithm, IEEE Trans. on Electronic Computer,1967, 16(4): pp.97-98.

[3] J. Soukup, Fast Maze Router, In: Proc. of 15th Design Automation Conference, 1978: pp.100-102.

[4] Hadlock, A Shortest Path Algorithm for Grid Graphs, Networks, 1977, 7: pp.323-334.

[5] F. Rubin, The Lee Connection Algorithm, IEEE Trans. on Computer, 1974, 23: pp.907-914.

[6] J. L. Ganley and J. P. Cohoon, Routing a multi-terminal critical net: Steiner tree construction in the presence of obstacles, In: Proc. of IEEE ISCAS, London, UK, 1994, pp.113-116.

[7] M. Zachariasen and P. Winter, Obstacle-avoiding Euclidean Steiner trees in the plane: an exact algorithm, extended abstract presented at the Workshop on Algorithm Engineering and Experimentation (ALENEX), 1999.

[8] Z. Zhou, C. D. Jiang, and L. S. Huang, Finding Obstacle-Avoiding Shortest Path Using Generalized Connection Graph with $O(t)$ Edges, Chinese J. of Software, 2003, 14(2): pp.166-174.

[9] Z. Zhou, C. D. Jiang, and L. S. Huang, On optimal rectilinear shortest paths and 3-steiner tree routing in presence of obstacles, Journal of Software, 2003, 14(9): pp.1503-1514.

[10] Y.Yang, Q.Zhu, T.Jing, and X.L.Hong, Rectilinear Steiner Minimal Tree among Obstacles, In: Proc. of IEEE ASICON, Beijing, China, 2003, pp.348-351.

[11] D. M. Warme, P. Winter, and M. Zachariasen, Exact Algorithms for Plane Steiner Tree Problems: A Computational Study, Technical Report DIKU-TR-98/11, Department of Computer Science, University of Copenhagen, April 1998.

[12] F. K. Hwang, D. S. Richards, and P. Winter, The Steiner Tree Problem, Annals of Discrete Mathematics, Amsterdam, The Netherlands: North-Holland, 1992.

[13] M. Dorigo, V. Maniezzo, and A. Colorni, The Ant System: Optimization by a colony of cooperating agents, IEEE Trans. on Systems, Man, and Cybernetics - Part B, 1996, 26(1): pp.1-13.

[14] S. Das, S. V. Gosavi, and W. H. Hsu, An Ant Colony Approach for the Steiner Tree Problem, In: Proc. of Genetic and Evolutionary Computing Conference, New York City, New York, 2002.

[15] S. Q. Zheng, J. S. Lim, and S. S. Iyengar, Finding obstacle-avoiding shortest paths using implicit connection graphs, IEEE Trans. on Computer Aided Design, 1996, 15(1): pp.103-110.

[16] M. Zachariasen, Rectilinear Full Steiner Tree Generation, Networks, 1999, 33: pp.125-143.