2024 IEEE International Symposium on Circuits and Systems (ISCAS) | 979-8-3503-3099-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCAS58744.2024.10558430

A Rule-Based High Efficient Obstacle-Avoiding RSMT Algorithm for VLSI Routing

Junhao Guo†, Hongxin Kong‡, Lang Feng†*
†*School of Integrated Circuits, Sun Yat-sen University;* ‡*Advanced Micro Devices, Inc.*
guojh73@mail2.sysu.edu.cn; hongxink@amd.com; fenglang3@mail.sysu.edu.cn;

***Abstract*—For VLSI physical design, the routing problem has attracted attention in recent years due to the emerging manufacturing technologies. Tree generation is one key routing step directly affecting the routing quality, which is to find the rectilinear steiner minimal tree (RSMT) of each net. Ordinary RSMT algorithms such as FLUTE fail to generate valid trees avoiding obstacles. In contrast, current obstacle-avoiding RSMT (OARSMT) algorithms can incur a large runtime overhead compared with FLUTE. To reduce the runtime cost while maintaining the quality, a novel OARSMT algorithm is proposed in this work. By proposing multiple rule-based routing schemes, which are fast while maintaining the awareness of global conditions from mature RSMT solutions, OARSMT solutions with reasonable qualities can be quickly obtained, even for large and complicated cases. Compared with the state-of-the-art works, traded with limited wirelength overhead, the proposed algorithm has ∼10x-2700x and ∼150x-5800x runtime speedup under randomized testcases and standard benchmarks, respectively.**

I. Introduction

Routing is one critical and time-consuming step in very large-scale integration (VLSI) physical design. It involves multiple phases, including tree generation, global routing, detailed routing, etc., within which tree generation is the first phase that has a decisive impact on the qualities of later phases. The tree generation has been researched for many years and mature algorithms with practical runtime and quality trade-off have been proposed. However, with the improvement of manufacturing technologies, more and more obstacles are involved and greatly increase the difficulty of routing. The obstacles may be due to macro cells, IPs, 3D packaging, prerouted nets, power networks, etc. To improve the routing quality, the first critical step is to design a new tree generation algorithm for obstacles.

Tree generation in routing is to find the topological tree structures to connect the pins in each net, which has the minimum cost (typically the total wirelength of edges). In VLSI physical design, this is typically formed as the rectilinear steiner minimal tree (RSMT) problem. Although the RSMT problem is an NP-hard problem, it can be mostly well addressed by FLUTE [1] that leverages the recorded pin patterns and solutions, which are widely applied in modern routers. However, it cannot be directly applied in scenarios with obstacles due to design rule violations. The patterns involved with obstacles have much more complex conditions so that the basic idea of FLUTE cannot be easily migrated to the OARSMT problem.

*The corresponding author. §This work was partially supported by National Natural Science Foundation of China (Grant No. 62204111).

To tackle the above challenges, various researches are proposed, such as leveraging obstacle-avoiding spanning graph (OASG) [2], [3], using the intermediate results during maze routing [4], or applying machine learning [5], etc [6], [7], [8], [9], [10], [11], [12]. Nevertheless, they still suffer from relatively low speed or quality, especially for large cases. For example, the work [2] leverages the OASG and trims the edges and steiner nodes to reduce the design space. This restricts the solution quality. The work [12] uses a guiding solution-based local search method using the escape graph to find and refine the tree structures. Although it also uses greedy rules, they are not as complete as those in the proposed approach, and it focuses more on the complex algorithms that can lead to large runtime costs. The work [4] is the state-of-the-art work that uses the intermediate information of maze routing to reduce the cost of the tree. Its quality is further improved compared with previous works. However, the runtime is still impractical especially for large cases.

To further increase the speed while maintaining the quality, a rule-based algorithm is proposed, which uses mature RSMT solutions in the early stage as a global baseline guidance. The detailed contributions are as follows:

- Proposes a basic rule-based OARSMT algorithm that is not only fast, but also able to provide high efficient solutions by considering the guidance of RSMT solutions from mature RSMT solver FLUTE.
- Multiple approaches to extend the rules are provided, which consider more conditions so that it can efficiently explore more opportunities to avoid the local optimum during OARSMT optimization.
- Throughout experiments including the comparisons between multiple works are conducted, which indicate that the proposed algorithm has ∼10x-2700x and ∼150x-5800x speedup for randomized testcases and standard benchmarks, respectively, traded with limited wirelength overhead.

II. Algorithm Design

A. Problem Formulation and Symbols

The OARSMT problem in our work is defined as follows. There are m pins $\{p_0, p_1, p_2..., p_{m-1}\}$ that are connected as a net. There are n obstacles $\{b_0, b_1, b_2..., b_{n-1}\}$ that are all rectangles with width w_i and height h_i for each b_i. The pins and obstacles are located in a xy-plane without overlap. The OARSMT problem is to find a rectilinear steiner tree with minimal cost that connects all pins, with no edge crossing any obstacles and no steiner node located inside any obstacles. The cost is defined as the wirelength with Manhattan distance, and an edge passing through (or a steiner node located at) the boundary of an obstacle is assumed to be legal.

B. Design Rationale and Edge Updating

As FLUTE does not consider obstacles, previous works do not apply it in the early stage. This may leave out the valuable guidance of optimized tree structures, and incur more runtime. The proposed algorithm in this work leverages the solutions of FLUTE as a guidance, and further updates the solutions to optimize and legalize the layout with obstacles.

The basic procedure to perform the solution updating is to find all the straight-line edges from a FLUTE solution that cross obstacles, and update them one by one. Using maze routing or recording the patterns of obstacles in a look-up table might be two approaches, but they either suffer from large runtime or tremendous patterns to be recorded. A trade-off approach can be the rule-based edge updating. Two example rules to update the edges from FLUTE are shown in Figure 1(a), including directly detouring the obstacles, and connecting through the obstacle boundaries. However, they can incur large cost or even violations with other obstacles, indicated as red lines in Figure 1(a).

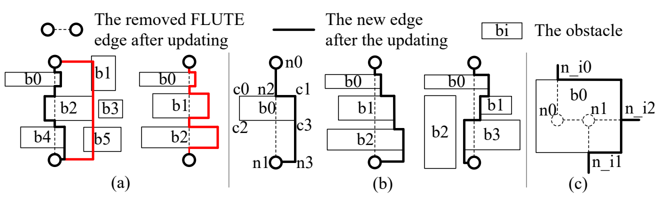

Fig. 1. (a) Examples of directly detouring and connecting through the obstacle boundaries; (b) The examples where the proposed reference line-based edge updating show the optimal solutions; (c) An example where the steiner nodes generated by FLUTE are in an obstacle.

To tackle the above challenges while keeping the fast process, a new rule-based edge updating approach is proposed, which keeps a balance between the two rules in Figures 1(a). In short, the new algorithm iteratively bends the edge line only when it is blocked on its direction, and each time bends to the obstacle corner nearest to the original edge. Taking the left part of Figures 1(b) as an example, the line of the original edge generated by FLUTE is named the *reference line* (such as $[n_0, n_1]$). One of the nodes of the original edge is first selected as the source node (such as n_0), while another is the target node (such as n_1). The direction from the source node to the target node is named the *reference direction*, which is vertically from the top to the bottom in this example. Then, a line is created and extended from the source node to the target node through the reference direction. Once the line is blocked by an obstacle's boundary (e.g., the line extended from n_0 is blocked at n_2 by the top boundary of b_0), it is bent and extended to a corner of this boundary (e.g., the extended line segment after bending is $[n_2, c_1]$). The direction of bending is to the corner that is closer to the reference line (e.g., corner c_1 in this case). Once the bent line is extended and reaches the obstacle's corner, it is then bent again and extends through the reference direction again (such as $[c_1, n_3]$). Once the line is extended to have the same coordinate as the target node (e.g., n_3 has the same y-coordinate as n_1), the line is bent again towards the target node and connects it.

Except the edge itself, the steiner nodes from FLUTE on edges also need to be updated when they are located inside the obstacles, such as the example in Figure 1(c). This can be fixed by regenerating the edges shown as the black lines in the figure, and removing the invalid edges and nodes. The black lines are generated as the shortest edges placed around the obstacle that can connect all the intersection points (n_{i0}-n_{i2}).

C. Rule Enhancements

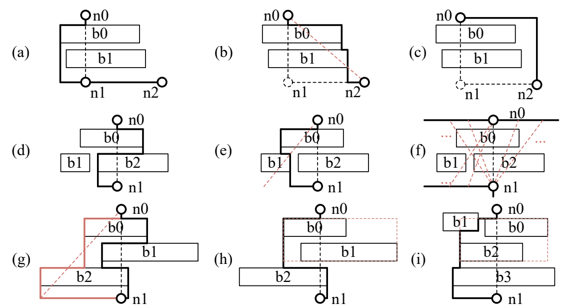

Fig. 2. (a)-(c) An example of better-updating L-shape connections. (d)-(f) An example of the benefit by sloping the reference line. (g)-(i) An example of the benefit by merging obstacles. (The reference line is shown as the red dotted line if it is not overlapped with the original edge to be updated, and the red solid lines are the its corresponding updated edges, if illustrated.)

1) L-Shape Optimization: Although the edge updating performs well for the cases in Figure 1, there are still some cases that are difficult with only the basic edge updating. For example, In Figure 2(a), 3 steiner nodes from FLUTE are shown. Edge updating needs to update $[n_0, n_1]$, and the solution is Figure 2(a) as the left corner of $b0$ is nearer to $[n_0, n_1]$. However, if n_0 connects n_2 by the edges on the right of Figure 2(b), after $[n_0, n_1]$ and a segment of $[n_1, n_2]$ are removed, the optimal solution can be obtained. This can be realized by setting the reference line as $[n_0, n_2]$. Besides, if we change the L-shape direction as Figure 2(c), another optimal solution can also be obtained. Inspired by these findings, given an edge $[n_i, n_j]$ needs to be updated, if the degree of n_i or n_j is 2 (which means there exists an L-shape), the reference line of edge updating is the diagonal of this L-shape. Besides, two types of L-shapes are both tried to perform the edge updating, and the solution with a smaller wirelength is kept.

2) Reference Line Sloping: Except for L-shapes, there are also other cases that can be explored. For example, Figure 2(d) shows the solution of the proposed edge updating for $[n_0, n_1]$, but a better solution is shown in Figure 2(e). This can be obtained by setting the reference line with a slope to the left. Therefore, when updating $[n_i, n_j]$, all its orthogonal half-lines (at most 4) starting from n_i (or n_j) are considered to build the reference lines between the source node n_j (or n_i) and a node on the half-lines, respectively. k_l evenly distributed reference lines within the bounding box are tried for each half-line, where k_l is set by users. An example is shown in Figure 2(f). After updating edge $[n_0, n_1]$ by multiple reference lines, the solution with the smallest wirelength is kept.

3) Obstacle Merging: Another approach for rule enhancement is to merge multiple nearby obstacles, and the bounding box of these merged obstacles is regarded as a new obstacle when applying edge updating. In Figure 2(g), the original edge updating incurs large cost due to lots of bends, and the sloped reference line on the left can also incur large cost. In contrast, the solution in Figure 2(h) becomes more optimized where b_0 and b_1 are merged. Note that sometimes merging obstacles can cause overlaps, such as b_1 in Figure 2(i). Considering the runtime, if the edge updating is blocked by the overlapped obstacles, the edge extends on the boundaries of the overlapped obstacles, through the shortest path until able to perform the edge extension without the overlaps. Finally, similar to the reference line sloping approach, when updating each edge $[n_0, n_1]$, k_m merging strategies are tried, where k_m is set by users. More specifically, if there are n' obstacles that cross $[n_0, n_1]$, the merging strategies of merging each 1, n'/k_m, $2n'/k_m$, $k_m \times n'/k_m$ obstacles are tried, and the solution with the smallest wirelength is kept.

D. The Rule-Based OARSMT Flow

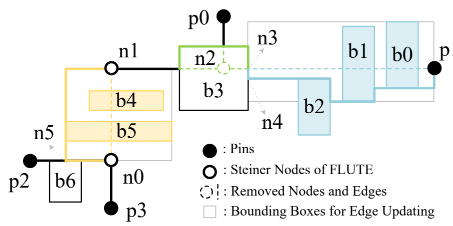

Fig. 3. An example of the solution from the proposed algorithm. (Different colors stand for different elements considered in different processes)

Based on the edge updating rule, the proposed rule-based OARSMT flow can be separated into 4 steps:

(1)RSMT Generation; (2)Steiner Nodes Legalization; (3)Recursive Edge Updating; (4)Post Processing.

For the first step, the FLUTE solution is obtained. This solution usually only includes the connections between nodes, and thus a rectilinear tree first needs to be built. The edge between two connected nodes from FLUTE is constructed by an L-shape pattern, and the direction is basically randomly selected in the basic algorithm. The second step is to legalize the steiner nodes inside the obstacles as Figure 1(c). An example is also shown as the green parts in Figure 3. The critical step is edge updating, which is recursively performed. Each time the an edge e with violations is selected, the set of obstacles S_b that block e is obtained. Then, a bounding box B_b is heuristically introduced to greatly reduce the complexity. The bounding box is a rectangle that just surrounds e and all obstacles in S_b, and the candidate obstacles S_{candb} are selected as the obstacles overlap B_b. For example, in Figure 3, when updating edge $[n_0, n_1]$, B_b is built based on $[n_0, n_1]$, b_4 and b_5. In this case, $S_{candb} = \{b_4, b_5\}$. In contrast, when updating edge $[p_1, n_3]$, although B_b is not built based on b_2, but b_2 is included in S_{candb} since B_b overlap b_2. Next, the edge updating by all combinations of approaches introduced in Sections II-B and II-C is performed to e, with the consideration of the obstacles in S_{candb}. Note that all updated new edges will be further checked after edge updating of e, as they might also cross obstacles, so finally all edges are legal. Finally, the redundant edges (The edge connecting a steiner node with degree 1) are removed, and the overlapped edges are merged. This can be finished quickly by checking the edges connected to each steiner node or pin.

E. Data Structure Improvement and Time Complexity

The data structure is built based on the Hanan grid of the pins and obstacle corners (and the obstacle center if the obstacle is not crossed by grid lines). The grid lines can cover the bounding boxes and updated edges with practical grid size. For each grid line, a set recording the sorted obstacles' coordinates crossed the line is kept. When checking if an edge crosses any obstacles during edge updating, the edge coordinates can be checked with the recorded sorted coordinates of the corresponding grid line, and finished in $O(log(n))$ time. In this case, it can be proved that the time complexity of the proposed algorithm is $O(m \times (m+n) \times log(n))$. This is less than the state-of-the-art previous work [4]. Besides, in most cases, an edge crosses only a small portion of obstacles (Avg. $\sim$4 for the largest random cases in our experiments), and the n in the time complexity is much smaller than the number of all obstacles. This further leads to fast speed in practice.

III. Evaluations

A. Experiment Setup

The experiments are conducted on a server with Intel Xeon Gold 6348 CPU. There are two kinds of benchmarks: Randomized testcases and standard benchmarks. For randomized testcase generation, 3 factors are set: The number of pins and obstacles, and the obstacle density on the layout. Other parameters are randomly generated. Each setting is tested 50 times to get the average result. Besides, we also test all the standard benchmarks IND, RC, and RT [13] involved in the compared works [4], [2], [12]. $k_l = 5$ and $k_m = 2$ in the experiments. For comparison, the state-of-the-art work [4], a recent work [12] and a traditional work [2] are involved, with works [4], [2] reproduced. Similar to work [5], our experiments also apply the algorithm in work [4] by 2D mode that can be faster than 3D mode.

B. Wirelength and Runtime Analysis of Randomized Testcases

The wirelength difference and speedup of the proposed algorithm is shown in Table I, compared with the state-of-the-art work [4]. In Table I, each row stands for a setting of the number of pins, and each column stands for a setting of the number of obstacles. Each group contains 4 runtime results corresponding to different obstacle densities. One can notice that the proposed algorithm has $\sim$10x - 2700x speedup from small cases to large cases. The absolute values of the runtime for the proposed algorithm are also tested, and are 0.63ms for small cases and only 0.19s for large cases. This not only shows the fast speed, but also proves the scalability.

Besides, Table I also shows the wirelength difference, and negative results mean that the proposed algorithm has a smaller wirelength. The wirelength difference ranges from -5.76%

TABLE I
THE COMPARISONS WITH THE STATE-OF-THE-ART PREVIOUS WORK [4] ON THE RANDOMIZED TESTCASES.

	Density	#Obs=10	#Obs=50	#Obs=100	#Obs=500	#Obs=1000	#Obs=2000
Speedup							
#Pin=10	10%	34.56x	57.03x	133.74x	965.18x	1709.18x	2711.79x
	30%	28.02x	42.13x	65.05x	436.41x	1037.00x	1255.36x
	50%	20.61x	23.86x	47.62x	277.52x	863.57x	1052.37x
	70%	19.97x	18.41x	31.61x	261.89x	949.72x	725.38x
#Pin=20	10%	50.07x	99.56x	143.85x	1073.18x	1713.21x	2332.41x
	30%	37.29x	63.26x	85.33x	402.39x	793.79x	1159.35x
	50%	37.37x	40.62x	55.80x	251.26x	501.56x	790.10x
	70%	37.41x	33.13x	45.35x	198.97x	361.32x	563.91x
#Pin=30	10%	174.33x	109.19x	179.57x	1075.40x	1718.38x	2443.06x
	30%	59.79x	72.72x	99.10x	420.68x	681.07x	763.08x
	50%	64.34x	54.72x	71.49x	256.15x	399.53x	643.97x
	70%	58.02x	43.52x	59.27x	166.03x	308.12x	493.75x
#Pin=50	10%	102.39x	157.40x	205.02x	1040.27x	1647.71x	1876.00x
	30%	114.99x	99.48x	135.85x	449.44x	602.45x	690.97x
	50%	112.97x	89.93x	100.86x	259.32x	380.30x	433.12x
	70%	106.00x	72.53x	81.37x	188.36x	267.95x	315.92x
#Pin=100	10%	206.47x	230.86x	282.12x	1073.62x	1656.79x	1916.65x
	30%	212.87x	185.91x	214.37x	492.41x	668.77x	619.63x
	50%	214.97x	168.15x	190.66x	338.00x	436.84x	441.54x
	70%	206.01x	151.89x	157.39x	252.90x	287.52x	261.67x
#Pin=200	10%	429.27x	383.08x	445.77x	1042.32x	1457.32x	1779.90x
	30%	398.38x	353.72x	372.01x	621.75x	757.48x	646.16x
	50%	389.05x	355.30x	352.46x	441.75x	521.98x	442.24x
	70%	342.13x	331.91x	310.92x	348.23x	378.06x	316.04x
Wirelength Difference (<0 Stands for Improvements)							
#Pin=10	10%	-2.51%	-2.42%	-2.81%	-5.00%	-4.26%	-5.76%
	30%	-0.76%	-1.68%	-2.89%	-4.65%	-4.44%	-4.54%
	50%	0.56%	-0.57%	-2.65%	-2.83%	-3.83%	-4.54%
	70%	-0.31%	-4.86%	-2.54%	-3.53%	-1.56%	0.53%
#Pin=20	10%	-1.46%	-1.53%	-1.83%	-3.40%	-3.50%	-4.47%
	30%	-0.27%	-0.89%	-1.73%	-2.69%	-3.84%	-4.48%
	50%	0.47%	0.72%	-0.29%	-2.00%	-2.96%	-3.86%
	70%	1.20%	-1.34%	-0.18%	-1.74%	-2.30%	-0.74%
#Pin=30	10%	-0.13%	-0.76%	-0.98%	-1.46%	-3.37%	-3.17%
	30%	0.37%	-0.04%	-0.50%	-1.33%	-2.91%	-2.38%
	50%	1.58%	1.46%	0.39%	-0.98%	-2.55%	-2.69%
	70%	1.77%	2.68%	2.14%	-0.01%	-3.53%	-3.58%
#Pin=50	10%	-1.29%	-1.02%	-1.36%	-2.29%	-3.18%	-3.47%
	30%	-0.85%	-0.22%	-0.19%	-2.08%	-3.21%	-3.79%
	50%	-0.61%	0.72%	-0.11%	-1.64%	-2.75%	-2.99%
	70%	-0.22%	1.89%	1.59%	-0.66%	-3.31%	-1.77%
#Pin=100	10%	-1.02%	-1.01%	-1.16%	-1.59%	-2.13%	-2.51%
	30%	-0.65%	-0.40%	-0.27%	-1.28%	-2.23%	-2.62%
	50%	-0.87%	0.57%	0.86%	-0.43%	-1.66%	-2.60%
	70%	-1.39%	1.29%	1.51%	-0.13%	-1.37%	-1.70%
#Pin=200	10%	-1.15%	-0.98%	-1.02%	-1.16%	-1.56%	-1.85%
	30%	-1.15%	-0.23%	-0.10%	-0.51%	-1.21%	-1.98%
	50%	-1.21%	-0.18%	0.35%	0.23%	-0.63%	-1.29%
	70%	-0.91%	-0.05%	1.03%	1.01%	0.06%	-0.92%

to 2.68%. It is shown that the proposed algorithm can also outperform work [4] in the wirelength for various cases, and the wirelength improvement is 1.14% on average. For relatively large designs, the proposed algorithm shows >3% wirelength improvement. The experiments indicate that the proposed algorithm has better wirelength for randomized testcases compared with the previous work, but also spends orders of shorter runtime.

Moreover, we evaluate the wirelength improvement of the enhanced rules as shown in Figure 4, which shows the differences between the runs with and without using the enhanced rules. The algorithm with rule enhancements shows an average -1.88% wirelength improvement. For the cases with fewer pins (#pins$\leq$30) or a large number of obstacles (#obstacles$\geq$500), when obstacles each edge passes become complicated, it shows a bigger -2.22% and -2.69% wirelength improvement, respectively. The results prove that the rule enhancements, by exploring more design space, provide more opportunities to avoid local optimal solutions and achieve better wirelength.

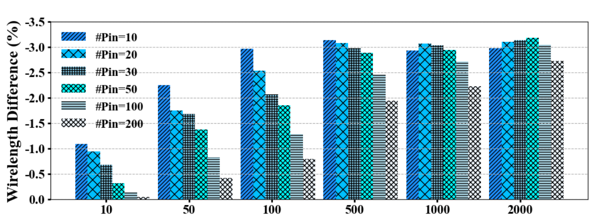

Fig. 4. Wirelength improvement for rule enhancements

Finally, we briefly report the wirelength and runtime comparisons with previous 2D OARSMT work [2]. The speedup of the proposed work compared to work [2] ranges from ∼11x - 7374x, with 1306x on average. Meanwhile, the wirelength difference ranges from -10.04% to -0.47%. The average wirelength improvement among all cases is 5.03%. The previous work [2] has a worse wirelength because it trims some optimized steiner nodes when building an initial tree structure at the early stages. This again indicates the advantages of the proposed algorithm on both quality and speed.

C. Comparisons on Standard Benchmarks

TABLE II
THE COMPARISONS ON THE STANDARD BENCHMARKS [13].

Test cases	#Pin	#Obs	Wirelength [2]	Wirelength GSLS [12]	Wirelength (SOTA) [4]	Wirelength Ours	Runtime (s) [2]	Runtime (s) GSLS [12]	Runtime (s) (SOTA) [4]	Runtime (s) Ours
IND1	10	32	639	604	618	604	0.08	0.01	0.01	0.0002
IND2	10	43	10700	9500	9157	9700	0.10	0.01	0.01	0.0041
IND3	10	50	662	600	600	617	0.18	0.01	0.01	0.0012
IND4	25	79	1165	1086	1115	1165	0.51	0.84	0.06	0.0042
IND5	33	71	1432	1341	1440	1389	0.41	0.57	0.02	0.0063
RC01	10	10	28270	25980	25470	26090	0.02	0.01	0.02	0.0070
RC02	30	10	44540	41350	41017	44790	0.03	0.04	0.04	0.0041
RC03	50	10	58550	54160	52748	60680	0.05	0.45	0.06	0.0089
RC04	70	10	63700	59070	56209	64780	0.08	0.48	0.11	0.0061
RC05	100	10	79630	74070	73825	78860	0.12	8.33	0.21	0.0065
RC06	100	500	86112	79731	78707	82951	27.14	462.88	7.57	0.0637
RC07	200	500	117040	108787	107885	113690	35.11	611.42	10.28	0.0623
RC08	200	800	122444	112601	111255	120378	94.38	475.22	34.34	0.1550
RC09	200	1000	120395	111105	110157	118011	151.69	604.82	48.18	0.1865
RC10	500	100	178120	164406	168390	168970	7.51	739.25	38.20	0.0540
RC11	1000	100	250358	232542	242285	234259	30.35	3260.22	23.57	0.2117
RC12	1000	10000	——	747496	——	749681	——	3398.36	——	0.2279
RT01	10	500	2318	2146	1879	2346	19.61	13.34	0.79	0.0247
RT02	50	500	50311	45852	45039	48087	24.29	23.11	5.69	0.0340
RT03	100	500	8809	7964	8203	8341	25.43	305.71	3.13	0.0437
RT04	100	1000	10777	9694	8566	12058	119.05	154.88	7.58	0.2569
RT05	200	2000	——	51357	45182	60985	——	622.37	82.12	0.4228
Total			1235972	1142589	1144565	1197766	536.16	6661.58	179.88	1.1410
Norm.			1.03	0.95	0.96	1.00	469.90	5838.37	157.65	1.00

• Each result of work [4] is selected by the better one between the reproduction and [4]'s report; The results of GSLS [4] are referred from its report; The results of work [2] are from the reproduction as the benchmarks in its report are the old versions.
• The results with "–" cannot be obtained in neither 3 minutes nor the corresponding report. The total wirelength and runtime are only counted for the testcases that all 4 algorithms have results.

Besides the throughout randomized testcases and comparisons, we also compare the proposed algorithm on all standard benchmarks [13] adopted in the compared works, including the state-of-the-art work [4], the recent work GSLS [12], and the traditional work [2]. The results are shown in Table II. The speedup of our work ranges from ∼150x to 5800x, with trading limited wirelength overhead compared with the works [4], [12], and with even better wirelength compared with the work [2]. Note that for the state-of-the-art work [4], there is no results for *RC12* in neither the reproduction nor its report, due to the large runtime. In contrast, the proposed algorithm only needs ∼0.2s for *RC12*. Besides, the runtime of the proposed algorithm also shows the scalability as it increases almost linearly with increasing the problem size, and this is in contrast with recent work GSLS [12]. In conclusion, the experiments on standard benchmarks also prove the high efficiency and scalability of the proposed algorithm. This is not only because of the theoretical lower time complexity, but also because of the simpler processing.

IV. CONCLUSION

This work proposes a rule-based OARSMT algorithm. Leveraging the RSMT solution, it not only considers more about the guidance from the global optimal solution, but also provides the opportunities for quick legalization and optimization. Compared with the state-of-the-art works, traded with limited wirelength overhead, the proposed algorithm has ∼10x-2700x and ∼150x-5800x runtime speedup under randomized testcases and standard benchmarks, respectively.

References

[1] C. Chu and Y.-C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2008.

[2] C.-W. Lin, S.-Y. Chen, C.-F. Li, Y.-W. Chang, and C.-L. Yang, "Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Spanning Graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 643–653, 2008.

[3] G. Ajwani, C. Chu, and W.-K. Mak, "FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 2, pp. 194–204, 2011.

[4] K.-W. Lin, Y.-S. Lin, Y.-L. Li, and R.-B. Lin, "A Maze Routing-Based Methodology With Bounded Exploration and Path-Assessed Retracing for Constrained Multilayer Obstacle-Avoiding Rectilinear Steiner Tree Construction," *ACM Transactions on Design Automation of Electronic Systems*, vol. 23, no. 4, pp. 1–26, 2018.

[5] P.-Y. Chen, B.-T. Ke, T.-C. Lee, I.-C. Tsai, T.-W. Kung, L.-Y. Lin, E.-C. Liu, Y.-C. Chang, Y.-L. Li, and M. C.-T. Chao, "A Reinforcement Learning Agent for Obstacle-Avoiding Rectilinear Steiner Tree Construction," *International Symposium on Physical Design*, p. 107–115, 2022.

[6] K. Clarkson, S. Kapoor, and P. Vaidya, "Rectilinear Shortest Paths through Polygonal Obstacles in O(n(Logn)2) Time," *Symposium on Computational Geometry*, pp. 251–257, 1987.

[7] 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 lambda-Geometry Plane," *International Symposium on Physical Design*, pp. 48–55, 2006.

[8] C.-W. Lin, S.-L. Huang, K.-C. Hsu, M.-X. Lee, and Y.-W. Chang, "Multilayer obstacle-avoiding rectilinear steiner tree construction based on spanning graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 11, pp. 2007–2016, 2008.

[9] P. Ghosal, A. Das, and S. Das, "Obstacle Aware RMST Generation Using Non-Manhattan Routing for 3D ICs," *Advances in Computing and Information Technology*, pp. 657–666, 2013.

[10] P. Ghosal, S. Das, and A. Das, "A Novel Algorithm for Obstacle Aware RMST Construction during Routing in 3D ICs," *Advances in Computing and Information Technology*, pp. 649–658, 2013.

[11] C.-H. Liu, C.-X. Lin, I.-C. Chen, D. T. Lee, and T.-C. Wang, "Efficient Multilayer Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Geometric Reduction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1928–1941, 2014.

[12] T. Zhang, Z. Lü, and J. Ding, "Guiding Solution Based Local Search for Obstacle-Avoiding Rectilinear Steiner Minimal Tree Problem," *IEEE Transactions on Emerging Topics in Computational Intelligence*, pp. 1–14, 2023.

[13] 11th DIMACS Implementation Challenge in Collaboration with ICERM: Steiner Tree Problems, https://dimacs11.zib.de/downloads.html, 2016.